Video Format Compiler Programmer´s Guide

Document Number 007-3402-003

CONTRIBUTORS

Written by Gregory Eitzmann
Edited by Beverley Talbott
Production by Carlos Miqueo
Engineering contributions by Rob Wheeler, Jeffrey Chung, and Ed Hutchins.
St. Peter’s Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.

Video Format Compiler Programmer’s Guide
Document Number 007-3402-003

Contents

List of Examples

List of Figures

List of Tables

Introduction

A video format is the set of electrical and timing characteristics that drive a monitor (or any video output device). In the context of this book, a video format usually refers to the video format source language you use to describe the video format itself.

This guide contains instructions you will need to write a video format. There is more than one way to do create one.

There is no need to use a ten-pound hammer for a one-pound task. Most carpenters know this. How complicated is your video format problem? Why are you writing a video format, anyway?

If you just need to assemble a quick little format for a more-or-less normal monitor to a resolution similar to the standard Silicon Graphics formats, see "Simple Video Formats Using Templates" on page xv.

On the other hand, if you have a special monitor that requires special timing, see "Detailed Video Formats Using Native Compiler Language" on page xvi.

Compiler Functions

The compiler takes the video format source language file and produces a video format object file. You can use this produced file to load video generation hardware. (Some hardware architectures require an intermediate step that combines multiple video format object files.)

Rules-Based Operation

The compiler produces video format object files for a single graphics architecture that you specify. You name the target architecture by specifying which hardware rules (supplied by Silicon Graphics) the compiler should follow when constructing the video format object file. A *rule* is a description of the operation of the hardware, written and provided by Silicon Graphics for each hardware architecture.

The compiler uses two rules: a *board* rule, which describes the circuit board driving the video output and a“*chip* rule, which describes the behavior of the integrated circuit producing the timing. Together, they define signals that drive video output boards. You specify both rules on the command line when you run the compiler. For more information on board definition rules files, see Chapter 7, “Board Hardware Definition.” Chip definition files are not described in this book.

Sample Rules

The examples in this book use sample rules files, one for the chip and one for the board: *samplechip.def* and *sampleboard.def*, respectively. They are installed when the video format compiler is installed. You can use these rules to work along with the examples; however, these rules do not represent real hardware and will not make a format that allows you to do anything beyond the examples.

Legitimate Hardware Rules

The rules that accompany actual hardware allow you to build formats targeted to that hardware. Note that the architecture of the hardware may be more restrictive than that of other hardware or that of the sample rules files: not all hardware is created equal!

Caution: Do not modify the source of the hardware rules that Silicon Graphics supplies. The rules describe the generation of many different signals and special scalar values and have been generated with some care. If you change the rules, you will likely get unpleasant results: the format will not run correctly; another part of the system will operate in an erratic or incorrect way or you may actually damage monitor or display hardware and invalidate your warranty.

Specifying Rule Sets

You can specify the rule set on the command line when executing the video format compiler (all the command-line options are described in the man page [reference page]). To specify the board and chip options, use the **-c** option of *vfc*. For example, to specify the sample rules (see “Sample Rules” on page xiv), use this command:

```
/usr/bin/vfc -c board-sampleboard.def,chip=samplechip.def filename.vfs
```

The chip and board files, not edited by users, are installed in a standard directory where the compiler knows to find them.

Simple Video Formats Using Templates

Use templates if you do not need the complexity of writing video formats in the native compiler language and your format is simple enough that you can write it using one of the standard templates. Your monitor should not require precise timing or it does can be driven by a generic format (such as VESA-compliant monitors).

The value of templates is that you need not know much about video formats and need know nothing at all about the native language of the compiler.

For general information on templates, read Chapter 1, “Running the Compiler With a Template.” For formats for most multi-sync monitors, read Chapter 2, “Using the Block Sync Template.”

Detailed Video Formats Using Native Compiler Language

You write video formats the hard way—using the native language—when the specification of your video format is too complicated or specific to be expressed using templates.

To begin, you need to know about the nomenclature Silicon Graphics uses for video formats—see Chapter 3, "Building Blocks of a Video Format." This chapter also serves as a tutorial for understanding the overall architecture of a video frame.

In Chapter 4, "Compiling Native Language Video Formats," you can find instructions for running the video format compiler.

The full native language for writing formats is explained in Chapter 5, "Native Compiler Language." This chapter contains the details of expressing a format to the compiler.

Before you begin writing, it is easier to start with an example of a video format and work from there. Chapter 6, "Examples of Native Compiler Language," shows annotated examples of different styles of video format.

Chapter 1

Running the Compiler With a Template

Templates are fairly simple, and all run in the same general way: you run the compiler with a predefined format whose values you supply on the command line. The compiler replaces the variables in its template format with the values you supply and builds the format.

For Users of InfiniteReality

Before continuing: have you considered hardware-based static resizing? With InfiniteReality graphics hardware, you may be able to save the trouble of using a template or creating a new video format. Instead, it may be possible to use a standard output format.

If you need only to reduce the resolution (the size of the format in lines and pixels) from one of the standard formats, you can instead use the *ircombine*(1G) tool to select a frame buffer size different from the monitor displays. The hardware takes care of the rest, magnifying the pixels in the frame buffer to the resolution of the monitor.

The *ircombine* online book has a great example of how you can use this feature.

Template Overview

A template is a normal video format source file whose most important values have been left blank. When you use the template, you specify those values through a parameter on the command line when you run the compiler. On the command line, you use the name of the parameter, assigning the value to it.

The Way Templates Work

There is no formal support in the video format compiler at all for templates: everything is performed via *cpp*(1) substitution. Because the compiler uses *cpp* to process its files and because the compiler allows you to specify *cpp* arguments on its command line, you simply perform normal cpp assignments, such as

```
-DLINES_PER_FRAME=1024
```

The command-line option for passing arguments to cpp is **-p**. You will read more about this in "Compiler Options" on page 3.

Template Location

You can use the standard system template that is shipped with the video format compiler (see Chapter 2, "Using the Block Sync Template") or you can write your own template. Standard templates are stored in the directory */usr/gfx/ucode/common/vfc/vfs/*.

Writing Your Own Template

If you will be writing many formats of the same kind, a template may be a shortcut for you:

- You can eliminate simple cut-and-paste errors by using one debugged format into which you can replace variables.
- You can control a master template that others can use from a simple *Makefile*.

How to Use the Compiler With Templates

You execute the video format compiler via a standard UNIX shell command line. The compiler has several options which are described in full in the man page (reference page) vfc(1).

Compiler Options

Besides specifying parameters, the most important options specify the board and chip configuration for which you are targeting the output. For information on the board and chip specification, see “Specifying Rule Sets” on page xv.

Passing Parameters to the Template

To specify parameters that are passed to the template, use the **-p** option to specify an option to be passed directly to *cpp* (see “How to Use the Compiler With Templates” on page 3 for information on parameters). For example, to pass the **-DLINES_PER_FRAME=1024** option to *cpp*, enter the following:

```
/usr/sbin/vfc -p "-DLINES_PER_FRAME=1024" ...
```

Be certain to surround the text with quotes; this keeps *vfc* from interpreting the option instead of passing it to *cpp*.

Example

The individual template is the best example of its own use. However, for a contrived example, presume the following:

- The name of the template is *GoodTemplate.vfs*.
- The rules files are *sampleboard.def* and *samplechip.def*.
- The template has two variables: *VBL_A* and *VBL_B*. These are assigned the values 3 and 4, respectively.
- The output object file (specified with the `-o` *vfc* option) will be *MyFile.vfo*.

To create the file, enter the following:

```
/usr/sbin/vfc -p "-DVBL_A=3 -DVBL_B=4" \
   -c chip=samplechip.def,board=sampleboard.def \
   -o MyFile.vfo \
   /usr/gfx/ucode/common/vfc/vfs/GoodTemplate.vfs
```

Although this example is available as part of the compiler installation and works as written, it will not create an object file that you can use with hardware—the examples are provided solely to permit practice of syntax. To use real hardware, you must use the rules files shipped with that hardware; you must specify the name of a legitimate template to create a meaningful format.

Chapter 2

Using the Block Sync Template

The block sync template creates the simplest format of all the templates, and if you are using a typical multi-synchronous monitor, you are in luck. The sync signal this template creates runs on most monitors that can accept a variety of sync signals because the vertical sync signature is a simple block.

Of course, your format must fall within the range of what the monitor can run–everything has its limits–but imagine of the block sync template with a cheerful, sunny manner that makes most monitors giddy at the prospect of running a format you create with this template.

Parameters

You can specify the parameters in Table 2-1 with the block sync template.

Table 2-1 Block Sync Template Parameters

Parameter Name	Definition	Default
ACTIVE_LINES	The number of lines in the active (visible) portion of the video format.	1024
ACTIVE_PIXELS	The number of pixels in each line of the active (visible) portion of the video format.	1280
FPS	The number of video frames per second the format should generate.	60

About the Created Format

The template is located in */usr/gfx/ucode/common/vfc/vfs/BlockSync.vfs*. See that file for details that may have changed since this document was published.

In general, the block sync template creates a simple horizontal blanking region with a sync pulse on every line that makes a downward transition at the beginning of the line, staying low for a small percentage of the line. The vertical blanking region has a short back porch, a single sync pulse that continues for several lines and terminates with a horizontal sync pulse, and a multiple line back porch. The durations of the total and the components of both horizontal and vertical blanking regions are variable, scaling themselves to the speed of the specified active region.

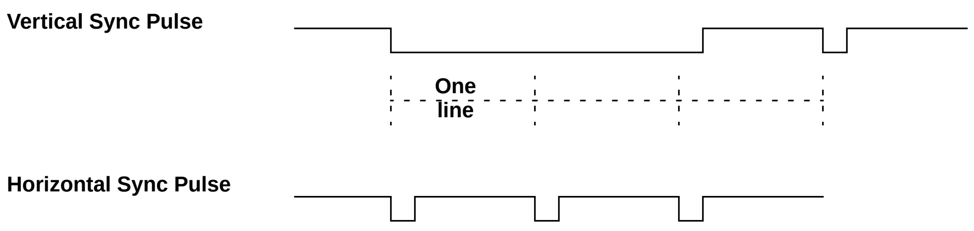

Figure 2-1 Block Sync Patterns

In Figure 2-1, you can see the layout of the sync pulses of the block sync template. This is only a prototype; the durations of the horizontal and vertical sync pulses depend on the size of the active pixel area you specify.

Examples of Template Use

You can use the following examples as a guide to using the block sync template. All examples in this section use the *sampleboard.def* and *samplechip.def* rules; for information on the sample rules, see “Sample Rules” on page xiv. Also, each example creates a file called *MyFormat.vfo* via the **-o** option. For a comprehensive list of all the *vfc* options, see the vfc man page (reference page).

Example 2-1 shows a high-resolution output that has 1200 lines and is 1500 pixels wide. Its frame rate is 72 Hz.

Example 2-1 1500 x 1200 at 72 Hz

```
# /usr/sbin/vfc \
    -p "-DACTIVE_LINES=1200" \
    -p "-DACTIVE_PIXELS=1500" \
    -p "-DFPS=72" \
    -c chip=samplechip.def,board=sampleboard.def \
    -o MyFormat.vfo \
    /usr/gfx/ucode/common/vfc/vfs/BlockSync.vfs
```

The format in Example 2-2 has 680 lines with 960 pixels per line. The frame rate is 50 Hz.

Example 2-2 960 x 680 at 50 Hz

```
# /usr/sbin/vfc \
    -p "-DACTIVE_LINES=680" \
    -p "-DACTIVE_PIXELS=960" \
    -p "-DFPS=50" \
    -c chip=samplechip.def,board=sampleboard.def \
    -o MyFormat.vfo \
    /usr/gfx/ucode/common/vfc/vfs/BlockSync.vfs
```

The format in Example 2-3 has 480 lines with 640 pixels per line. The frame rate is 60 Hz; note the frame rate is not specified because the default is already 60 Hz (see Table 2-1).

Example 2-3 640 x 480 at 60 Hz

```
# /usr/sbin/vfc \
    -p "-DACTIVE_LINES=480" \
    -p "-DACTIVE_PIXELS=640" \
    -c chip=samplechip.def,board=sampleboard.def \
    -o MyFormat.vfo \
    /usr/gfx/ucode/common/vfc/vfs/BlockSync.vfs
```

These are merely examples!

Although these examples are available as part of the compiler installation and work as written, they will not create an object file that you can use with hardware—the examples are provided solely to permit practice of syntax. To use real hardware, you must use the rules files shipped with that hardware (see the user guide for your hardware for the name); you must specify the name of a legitimate template to create a meaningful format.

Chapter 3

Building Blocks of a Video Format

Everyone has a unique way of describing the different parts of the format. To standardize the nomenclature of the different formats, the video format compiler declares names and properties for the different portions of video timing. In this chapter, you will find information on the following topics:

- Architecture of a Video Frame
- The compiler's definition of The Horizontal Line
- The compiler's definition of The Vertical Interval
- The differences in the Level of Sync
- How The Field of the Format is assembled and how it differs from the frame
- Definitions of Components of the frame
- The Silicon Graphics definition and use of The Pixel-to-Clock Ratio

This chapter is very much a tutorial. If you are familiar with the nomenclature Silicon Graphics uses in video formats, you can skip this chapter, using it only as reference.

Architecture of a Video Frame

What else is in a frame of video besides the pixels on the screen? Plenty!

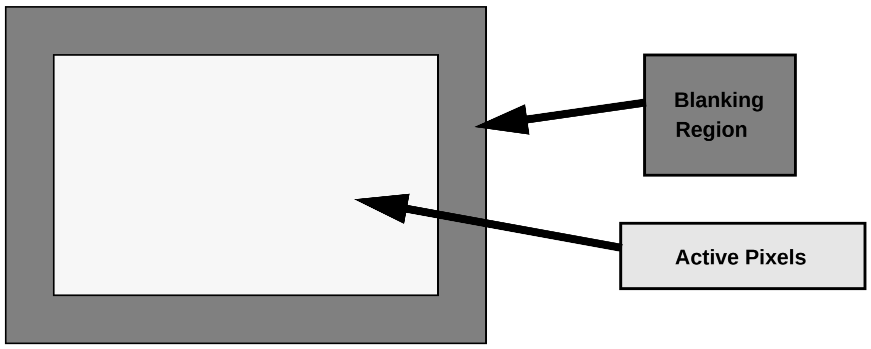

Figure 3-1 Active Pixels and Blanking Region

Figure 3-1 shows the geometric relationship between the *active pixels*—the part of the frame containing the picture you see on the screen—and the blanking region of a video frame. Blanking commonly consumes as much as 25 percent of a video frame, quite a lot for something you never see! So why include it?

We need the blanking region because of our terrible master: the cathode-ray tube (CRT). (Throw in a dash of convention, a bit of backward compatibility, and a slice of history, and you are up to 25 percent.) Old cathode-ray tubes, as well as many contemporary versions, need time and signaling information to manipulate the direction of the spray of electrons on the phosphorescent surface of the screen. Let us explain with a picture of the screen.

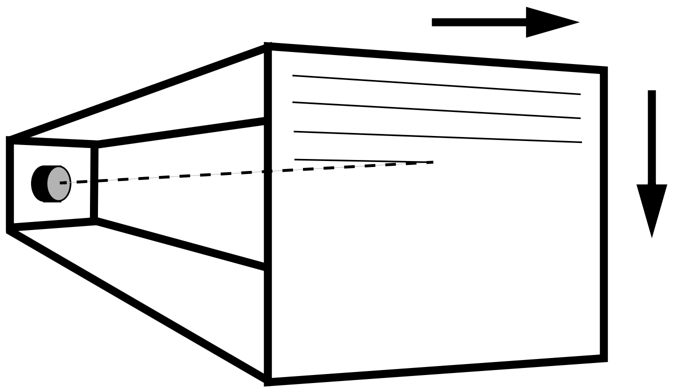

Figure 3-2 Painting the Screen With a CRT

The screens of most cathode ray tubes are drawn in a series of lines, left to right and top to bottom. When the monitor finishes drawing one line and reaches its right-most excursion, the beam is turned off (it is *blanked*) while the monitor returns the beam to the left side of the screen. A similar thing happens when the last line on the screen is finished drawing: the beam traverses to the top of the frame, ready to repeat its Sisyphusian task.

In the video format, what triggers the beam to return to the left side and to the top? It is the magic of synchronization signals. Synchronization signals are special pulses in the blanking region that tell the monitor the position at the beginning of the line; they also provide the frame with some geometric stability by lining up the left side of every line. When you write a video format, you specify the location and character of synchronization pulses.

The Horizontal Line

Each line of the video frame consists of well-defined regions. The most interesting region is that containing the active pixels (the picture drawn on the screen), but the blanking region is necessary to define the beginning and ending of each line.

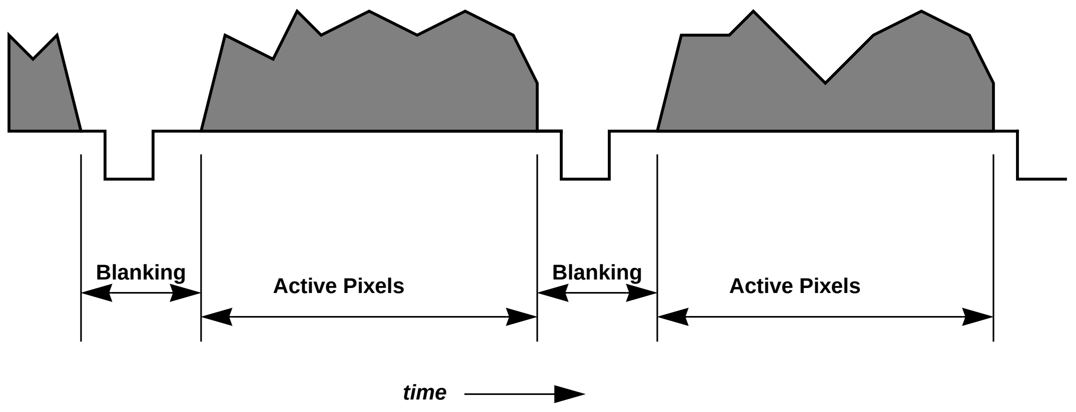

Figure 3-3 Line Showing Blanking Region

Figure 3-3 shows a typical line of a frame that contains active pixels. This figure shows active lines preceding and following the full line in the middle. The gray area—the picture content of the active pixel area—is variable because the picture in each line is different from the other.

You can see the blanking regions that separate the active pixels in each horizontal line. These are known as *horizontal blanking regions* because they constitute the black (or *blanked*) area between two horizontal lines. Typically, all lines in the frame containing active pixels have identical active and blanking geometries.

The synchronization pulse (the *sync* pulse) is the downward pulse in the blanking region. This pulse triggers the monitor to stop moving the beam in the rightward direction and resume drawing on the left side, one line lower. The horizontal line begins at the falling edge of one sync pulse and ends at the falling edge of the next sync pulse.

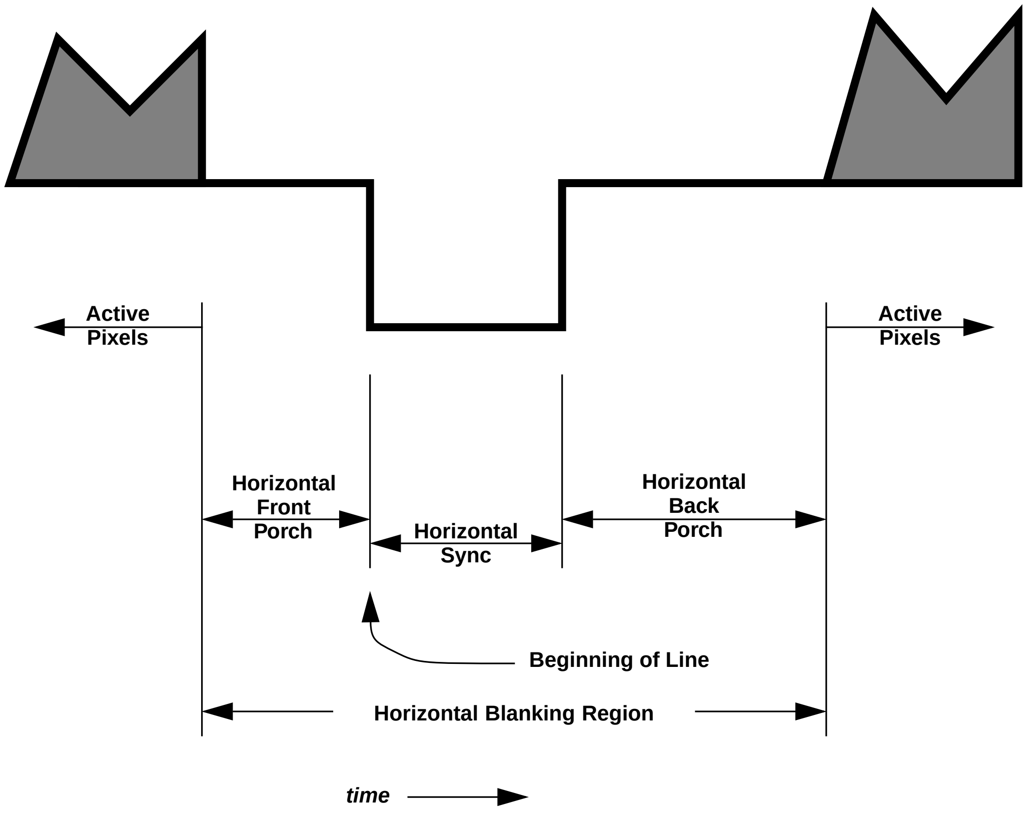

Figure 3-4 Detail of Horizontal Blanking Region

In Figure 3-4, you can see detail of the horizontal blanking region with the active pixels of the previous line terminating on the left side of blanking and those of the next line beginning on the right side of blanking. The components of the horizontal line are as follows:

- The *active pixels* contain the picture content — the visible pixels.
- The *horizontal front porch* is the period of the line between the active pixels and the beginning of the horizontal sync pulse.

- The *horizontal sync pulse* is a change in voltage of the video signal. It is this change in voltage that triggers the monitor to stop its rightward progress and begin drawing again on the left side of the screen. The line begins with the start of the horizontal sync pulse (and ends with the start of the next horizontal sync pulse).
- The *horizontal back porch* is the period of time between the end of the horizontal sync pulse and the active pixels.

The front and back porches provide some dead time where the monitor can be black before and after the active portion of the picture; this blackness is particularly important during the period of time the electron beam flies back the screen's left side to start drawing once again. In composite video formats such as NTSC, PAL, and SECAM, the horizontal back porch also contains the color burst, a color calibration reference.

If you are writing a video format because you have a special monitor, you need to know the durations of each different section of the line. The durations themselves are usually given in time units, although sometimes the durations are supplied in pixels (in which case you should use the duration of a pixel).

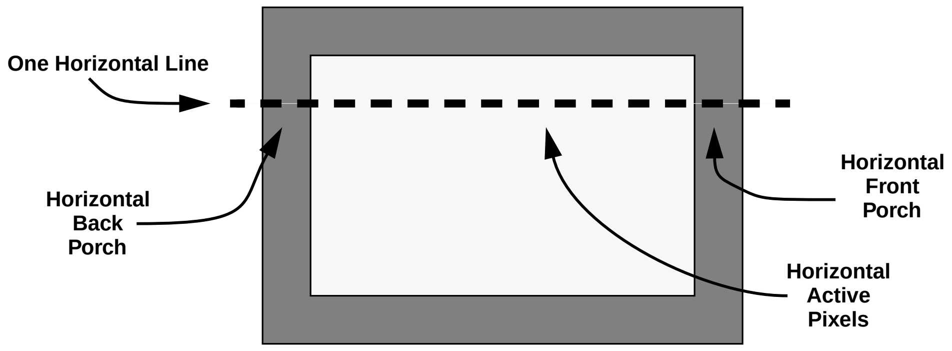

Figure 3-5 Detail of Horizontal Line in Screen Orientation

If you compare a horizontal line in Figure 3-5 to Figure 3-4, you can see a correspondence in the lines that contain active pixels. The horizontal front porch is the blanking region to the right of the active pixels in Figure 3-5; the horizontal back porch is to the left of the active pixels. The horizontal sync pulse cannot be shown in this kind of picture: it triggers the beam to fly back to the left side to screen to begin painting again.

All this talk of *horizontal* brings up a term used in video shorthand notation: the horizontal line is often referred to by the single roman letter *H*. The term is used so often that it now defines even the length of a horizontal line: *One H*.

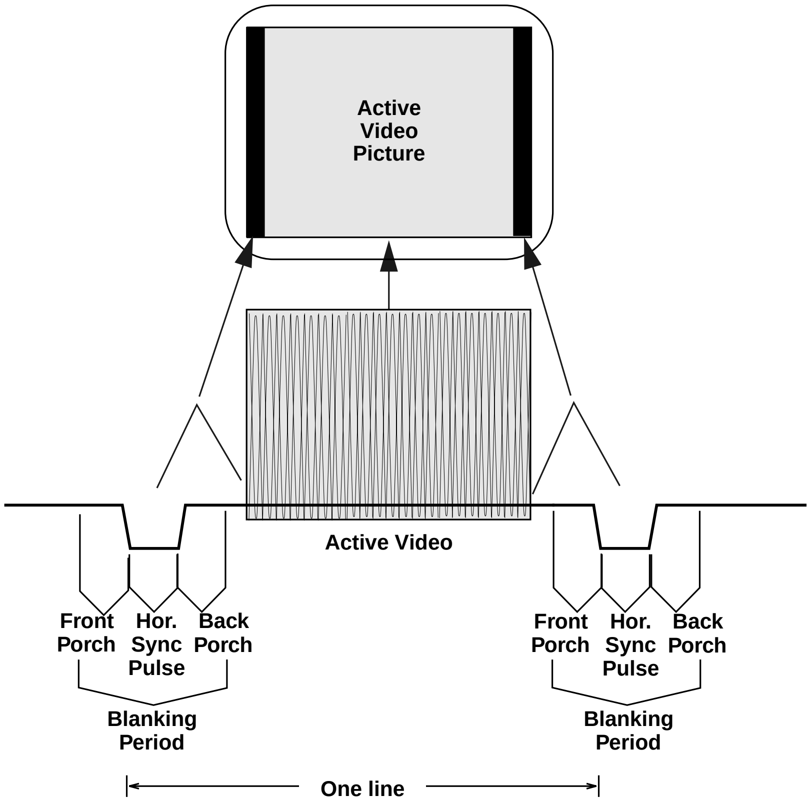

Figure 3-6 Summary of Horizontal Intervals

The diagram in Figure 3-6 wraps up our discussion of the horizontal interval, showing a single line of active video (shown in grey) and surrounded by two horizontal blanking regions (or *periods*).

The Vertical Interval

The vertical blanking region is similar in function to the horizontal blanking region: it has dead time that allows the monitor to display black picture content and a synchronization signal that directs the electron beam to fly back to its starting position. However, one begins to think vertically instead of horizontally.

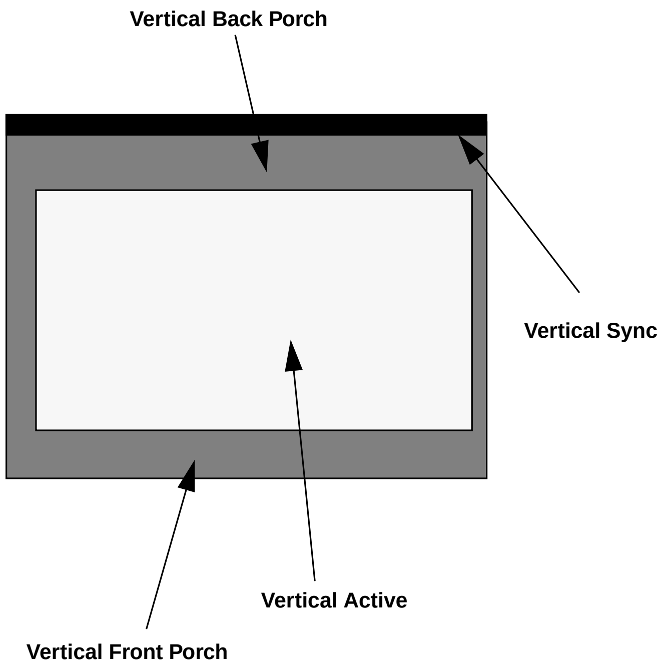

Figure 3-7 Detail of Vertical Regions in Screen Orientation

Figure 3-7 shows the vertical regions that are analogous to the horizontal regions found in Figure 3-5. You can see that each of the regions is longer than that of its horizontal counterpart, measured in lengths of multiple lines instead of portions of a line. The vertical sync cannot be shown well in a screen orientation illustration such as Figure 3-7 because sync triggers the beam to fly back to the top of the screen so the monitor can begin to show the next active area; it is represented here by a black band at the top of the frame.

Normally, one thinks of a video frame consisting of each of these regions. However, video formats consisting of more than one field have one vertical blanking region for each field; in this circumstance, the vertical front porch of one field is followed by the vertical sync of the next field.

To formalize the definition, the components of the vertical blanking region are as follows:

- The *active lines*, those containing the rendered frame buffer contents.
- The *vertical front porch*, the lines between the vertical sync and those containing active pixels.
- The *vertical sync*, the lines containing one or more vertical sync pulses. The frame begins at the start of vertical sync. In video formats with more than one field, each field begins with a vertical sync.
- The *vertical back porch*, the lines between those containing active and those containing vertical sync pulse(s).

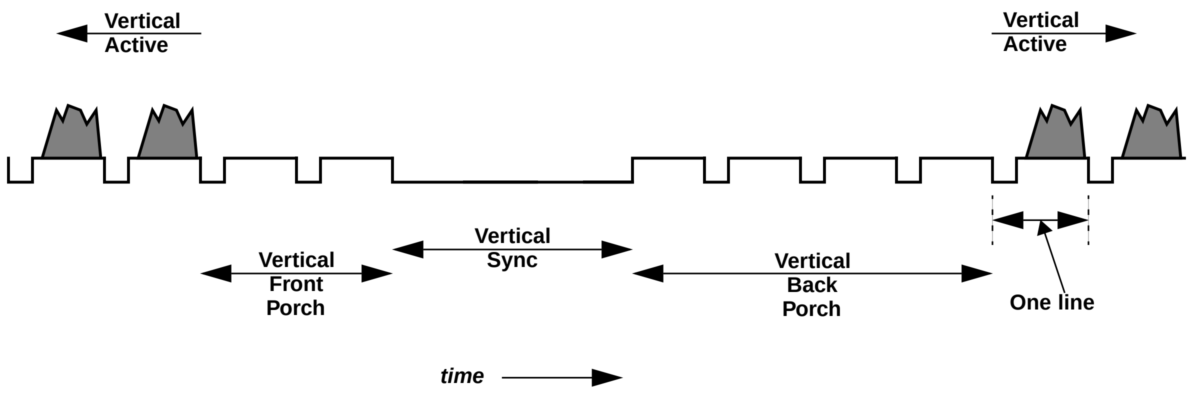

Figure 3-8 A Typical Vertical Blanking Region

In Figure 3-8, you can see a vertical blanking region for a typical video format. The scale is much larger than that shown in the diagrams of the "The Horizontal Line" section, with each different component measuring more than one line (the number of lines varies, depending on the format). The figure shows some of the lines of the previous field followed by the vertical front porch, the vertical sync, the vertical back porch, and the lines of the next field.

The vertical sync pulse triggers the monitor to stop its downward trek at the end of each line. When it receives the vertical sync pulse, the monitor starts drawing new lines at the top of the screen.

Each line in the frame is numbered, with the first line beginning with the beginning of the vertical sync pulse. A specific point in the frame is usually referenced by line number and an offset (in time units) into the line.

The vertical interval shown in Figure 3-8 is characteristic of a video format type known as *block sync*; you can see the vertical sync is a single long synchronization pulse. The block sync is common enough that a typical geometry of this form of pulse is included in the block sync template; see Chapter 2, "Using the Block Sync Template" for information on how the block sync template works.

Although the block sync type of video format is very common, other types of sync are also in general use. Another popular type of video format is called *commercial sync*, which contains smaller pulses in place of the large sync pulse.

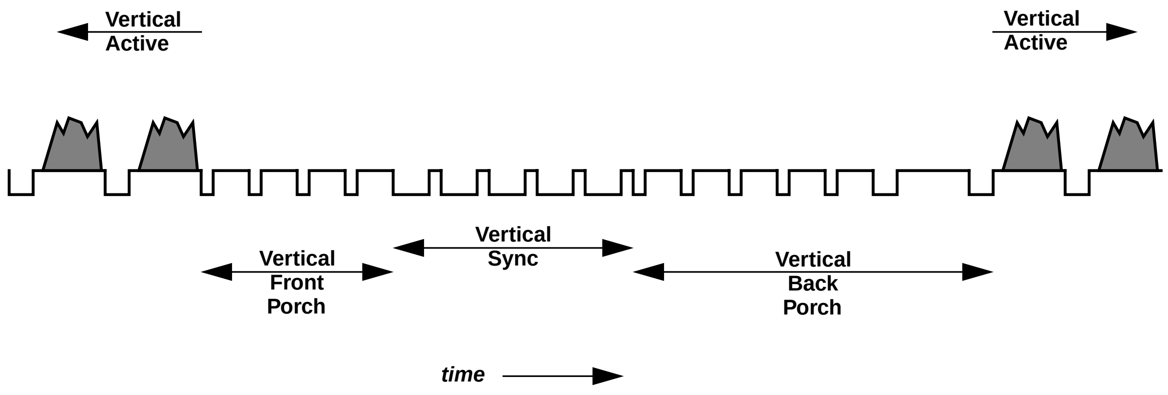

Figure 3-9 Vertical Blanking Region of Commercial Sync Format

The smaller pulses of vertical sync shown in Figure 3-9 act in the same way as the long sync pulse does in the block sync type of video format: vertical sync triggers the monitor to start drawing at the top of the screen. This commercial sync format also has short pulses through vertical front porch and some of vertical back porch. Characteristic of the commercial sync type of video format, these pulses are called *equalization pulses* and ease the monitor into and out of vertical sync. The half-line pulses in vertical sync are known as *serration pulses*.

Level of Sync

Most video formats have only two voltage levels of sync, low and high. Also, some monitors (notably, those for high-definition television, or *HDTV*) require an additional third level of sync called *tri-level*.

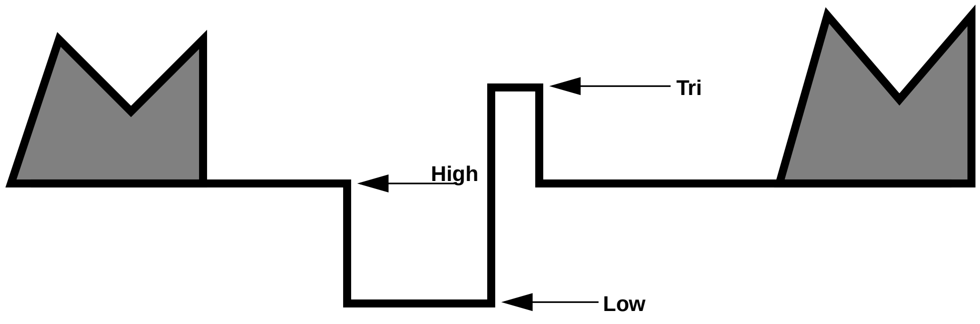

Figure 3-10 Different Levels of Sync

You can see the three different levels of sync in Figure 3-10:

- *Low*, also known as synchronization level. This is the lowest possible level of a sync signal.
- *High*, also known as blanking level. This is the level attained during blanking when sync signals do not drive the level otherwise.
- *Tri*, used for tri-level sync. This level of sync is at a predefined level, higher than the blanking level.

The Field of the Format

Thus far, we have discussed formats with only one *field*; that is, one contiguous set of video lines surrounded by the front and back porches. Formats with only one field are sometimes described as *progressive scan* formats. This single progression across and down the screen differs from the pattern used by multiple-field formats which may need to make many passes to draw the entire frame.

In the following sections, you can read about the following topics:

- "Interlaced Formats," where lines are interleaved
- "Stereo Formats," where the screen layout remains constant but content differs

Interlaced Formats

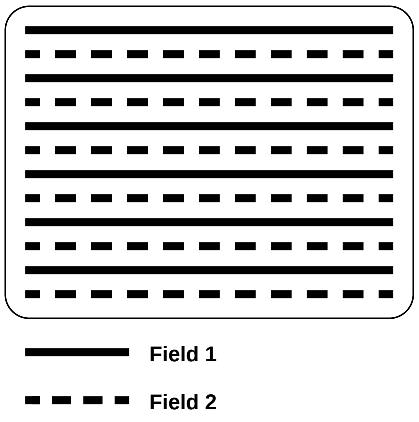

Figure 3-11 Interlaced Format Line Layout

The layout of the lines in Figure 3-11 shows the interleaving of lines of a typical two-field interlaced video format. In the first field of this example, all the odd-numbered lines are drawn. When the first field concludes, the drawing starts again at the top of the screen but only the even-numbered lines are drawn. This differs from the line layout of the progressive-scan frame shown in Figure 3-2 in which every line is drawn from top to bottom.

The interlacing layout may differ from one video format to the next. Although the example in Figure 3-11 shows the first field containing the odd-numbered lines, a different video timing format may have the even-numbered lines in the first field.

The format shown in Figure 3-11 is has two fields, typical for interlaced formats. However, there is no prohibition against formats with many more fields, if the monitor requires it. Some output generation hardware may impose additional restrictions on the number of fields you can create.

Stereo Formats

Stereoscopic vision video formats allow the viewer to see different images in each eye and are used with in conjunction with hardware to aid the illusion (such as eyewear that obscures one eye at a time). Silicon Graphics offers two different types of stereo formats:

- Per-Window Stereo, sometimes called *new-style* stereo. This stereo format allows some windows to show stereoscopic images while other images maintain their standard monoscopic display.
- Full-Screen Stereo, the *old-style* stereo format, is usually used only with applications whose windows occupy the entire screen. During its initialization, the application itself typically switches the video format to this format, restoring the previous format on exit. This type of stereo format is not recommended for new development.

Both of these stereo formats use the same method of addressing the monitor. Their difference comes from the organization of the frame buffer.

Per-Window Stereo

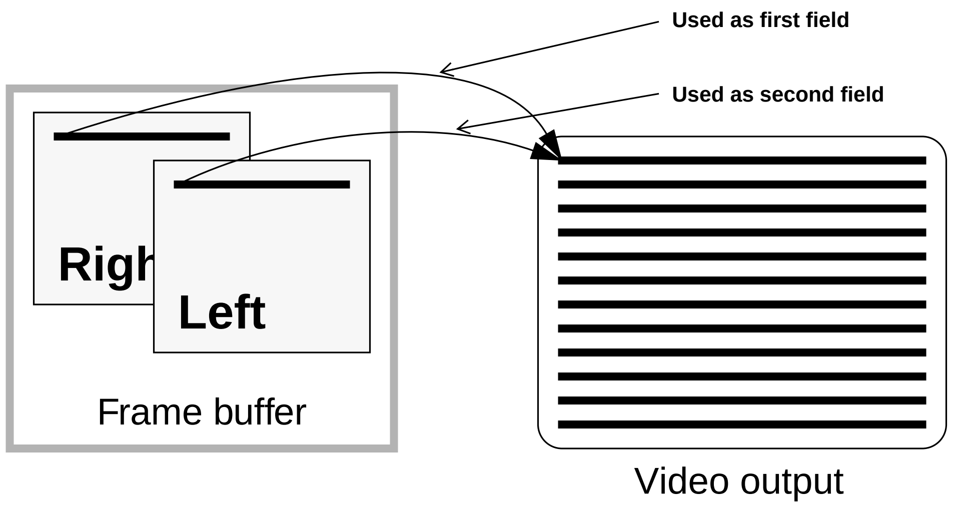

Figure 3-12 Per-Window Stereo Line Layout

Contrast the multiple-field interlaced format with the way in which the graphics/video subsystem creates a multiple-field video for stereo display, shown in Figure 3-12. In the stereo video format, the frame is drawn twice as often as it would be for comparable monocular vision.

The interlaced video format described in the previous section ("Interlaced Formats" on page 20) has fields that differ because the monitor displays them differently. The stereo format has fields that differ because the content of the frame buffer differs from each field, but the monitor displays each of the fields in the same way, on the same location on the monitor. The difference between the left and right fields in the frame buffer is that the drawing program draws the fields slightly differently

In per-window stereo, each pixel in the frame buffer contains a different section for left and right eyes (you can think of this as two pixels stored in the same address).

Figure 3-12 shows the left buffer drawn first, the right buffer second. A different monitor might require that the right buffer be drawn first.

Full-Screen Stereo

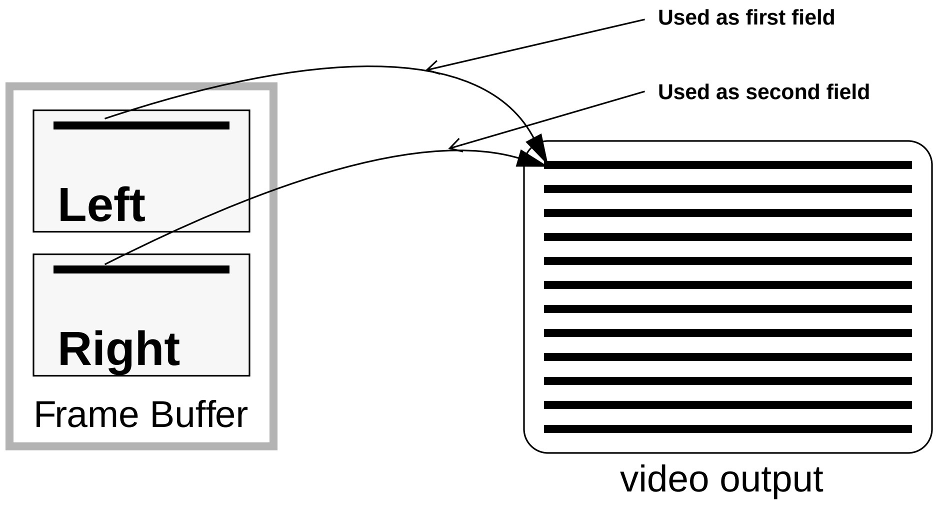

Figure 3-13 Full-Screen Stereo Line Layout

Compare Figure 3-13 to Figure 3-12 describing Per-Window Stereo. The difference between these two stereo modes is relatively minor: full-screen stereo fetches its left and right pixels from different locations, the pixels in the upper part of the frame buffer displaying one eye, pixels in the lower part give the other eye.

This full-screen stereo mode also requires additional support. See XSGISetStereoMode(3X11).

Definitions of Components

The definition of video format components can differ from one specification to another. For example, the PAL definition of the beginning of the frame is at the beginning of the vertical synchronization pulse; the definition for NTSC places the beginning of the frame at the first equalization pulse after the last line of active video (Report 624-4, Section 11A, XVIIth Plenary Assembly of the CCIR, Düsseldorf, 1990).

Confusing? Yes. But not problematic to the construction and analysis of the frame.

To solve the problem of varying definitions, the video format compiler declares a standard boundary for each component of the video frame. These boundaries apply regardless of the video format used.

Under some circumstances, you may need to convert from the standard used by a video format specification to the compiler's definition. In the case of the position of the beginning of frame in the NTSC and PAL video formats, the video format compiler uses the same definition as does PAL, with the start of the frame beginning at the vertical synchronization pulse. In this circumstance, the NTSC format adopts a slightly different definition.

For definitions of horizontal line components, see "The Horizontal Line" on page 12; vertical components are defined in "The Vertical Interval" on page 16.

The Pixel-to-Clock Ratio

This section describes an artifact of the way video is created. The pixel-to-clock ratio does not have much to do with video formats themselves.

Video is often grouped into units of multiple pixels for handling in hardware. By grouping pixels, the video hardware deals with a group of multiple pixels, not with individual pixels.

What is the consequence of this grouping? Although the pixels retain their individual identity from the frame buffer, the horizontal and vertical blanking intervals (see "The Horizontal Line" on page 12 and "The Vertical Interval" on page 16) do not have the same flexibility. You cannot program, for example, a sync transition to occur on an arbitrary pixel boundary. Instead, these transitions can occur only at pixel positions that are on the boundaries of these groups. The quantizing effect forces the compiler to alter positions of some transitions.

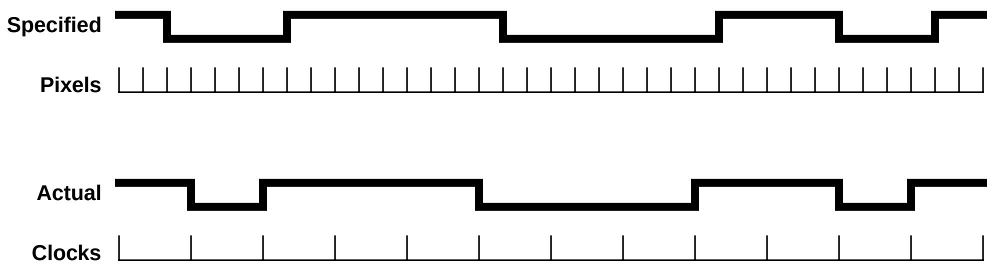

Figure 3-14 Quantization Example

For example, Figure 3-14 shows the quantization effect of a hardware output device that has a pixel-to-clock ratio of 3:1 (three pixels per clock). The series of transitions labeled *Specified* shows the set of transitions as they might be specified to the compiler. Because the hardware can resolve transitions only every three pixels, the compiler will round the time of each transition to be that of the nearest clock transition. The result is shown in the set of transitions labeled *Actual*.

Some output hardware is fixed at only one ratio, while other hardware may support different quantization ratios based on the final frequency, optional attributes, or other characteristics. The release notes accompanying the video format compiler for your hardware will help you determine the quantization value you should expect. You can see the quantization value applied to your video format by using the **-a ascii** command-line option of the compiler.

Chapter 4

Compiling Native Language Video Formats

Compiling video formats is similar to compiling any other source language. A good place to start is with the vfc man (reference) page; this shows the different options available to you.

Moreover, using the compiler to create video format object files from your own source files is similar to using templates. See "How to Use the Compiler With Templates" on page 3 for a description of that process.

This chapter has the following main sections:

- "Specification of Rules Files"
- "Format Analysis"
- "The Pre-Processor and Its Options"

Specification of Rules Files

You can write a video format to run on multiple architectures. However, it must be compiled independently for each architecture. To compile for each architecture, you must specify two distinguishing characteristics of the target hardware: the chip and board that will run the timing you generate with the compiler.

Use the **-c** option to specify the rules files used for each chip and board. For example, if the rules file for the chip is *samplechip.def* and the board is *sampleboard.def*, you use the following syntax:

```
vfc -c chip=samplechip.def,board=sampleboard.def ...
```

Refer to vfc(1) for details of which chip and board to use for each architecture. The man (reference) page there is updated more frequently than is this document.

Format Analysis

When it compiles a format, the video format compiler analyzes the frame to determine locations and durations of salient features, such as sync, back porch, and so on. You can use the **-a ascii** option to get a textual description.

Example 4-1 Use of -a ascii Output

```
1280x1024_60.vfo:
 Total lines per frame:   1065
 Total pixels per line:   1680
 Active lines per frame:  1024
 Active pixels per line:  1280
 Frames per second:       60
 Fields per frame:        1
 Swaps per frame:         1
 Pixel clock:             107.352 MHz, period = 9.31515 nsec
 Hardware pixel rounding:  every 2 pixels
 Line analysis:
  Length:                 1680 Pixels, 1 Lines, 15.6495 usec; (line 0)
  Frequency:              63.9 KHz, period = 15.6495 usec
 Horizontal Sync:         120 Pixels, 1.11782 usec; (line 38)
 Horizontal Back Porch:   240 Pixels, 2.23564 usec; (line 38)
 Horizontal Active:       1280 Pixels, 11.9234 usec; (line 38)
 Horizontal Front Porch:  40 Pixels, 372.606 nsec; (line 38)
 Field Information:
  Field Duration:           1.7892e+06 Pixels, 1065 Lines, 16.667 msec; (line 0)
  Vertical Sync:            5040 Pixels, 3 Lines, 46.9484 usec; (line 0)
  Vertical Sync Pulse:      5160 Pixels, 3.07143 Lines, 48.0662 usec; (line 0)
  Vertical Back Porch:      58800 Pixels, 35 Lines, 547.731 usec; (line 3)
  Vertical Active:          1.72e+06 Pixels, 1024 Lines, 16.025 msec; (line 38)
  Vertical Front Porch:     5040 Pixels, 3 Lines, 46.9484 usec; (line 1062)
```

In Example 4-1, you can see the output created by **-a ascii** to the standard high-resolution video format (the format source is in Example 5-1). The definitions of each named section is in the tutorial in Chapter 3, “Building Blocks of a Video Format.” In parentheses, following some specifications, you will find the line number on which the compiler made the determination of each item.

You can optionally specify a file name as an argument to this option, as in **-a ascii=/usr/tmp/foo**; the default is *stdout*.

The Pre-Processor and Its Options

As a convenience to users, *vfc* uses a pre-processor when compiling formats. The default is **`cc -E`** (see cc(1) for more information on the C-language preprocessor); however, not all users have the C development environment. To use a different pre-processor, use the **`-i`** option.

The `-p` option to *vfc* allows you to pass options directly to the pre-processor. Be certain to quote the string you pass, for example:

```
vfc -p "-DFPS=30" ...
```

When you quote the string, it keeps *vfc* from interpreting the pre-processor's option as one of its own. You can specify as many **`-p`** options as you need: *vfc* will accumulate all of them and apply them in order.

Chapter 5

Native Compiler Language

This chapter discusses the true basis for the video format compiler: the details of the native language and what the pieces do.

- "Building a Video Frame" on page 31, shows the components of a video frame.
- "Assignment Statements" on page 35, describes the expressions permitted in the video format source files.
- "The General Parameters Section" on page 38, reviews the basic variables that describe the overall frame.
- "The Active Line Section" on page 42, discusses the parameters that describe the active lines of the frame.
- "The Field Description" on page 43, describes how to define most of the detail of a frame in the transitions of sync in each field.

Building a Video Frame

In writing a video format in the compiler's native language, you describe the pattern of the synchronization signal pulses. You also describe the durations of each component of the horizontal line. Given this information, the compiler can produce a video format with which you can program the video generation hardware.

You build the video frame from one or more video fields, and each field is built by describing a series of components. The components are discussed in Chapter 3, "Building Blocks of a Video Format." Each field must be built independently because each field describes a contiguous stand of video lines surrounded by blanking.

The High-Resolution Format

Usually, it is not necessaryto write a video format completely from scratch. Instead, try copying or modifying an existing format source file. For block sync formats, the Silicon Graphics standard high-resolution format is a good start. See Example 5-1.

Example 5-1 The Silicon Graphics Standard High-Resolution Format

```
/*
** 1280x1024_60.vfs - SGI standard format
*/

General
{
    FieldsPerFrame = 1;
    FramesPerSecond = 60;
    TotalLinesPerFrame = 1065;
    TotalPixelsPerLine = 1680;
    ActiveLinesPerFrame = 1024;
    ActivePixelsPerLine = 1280;
    FormatName = "1280x1024_60";
}

Active Line {
    HorizontalFrontPorch = 0.372 usec;
    HorizontalSync = 1.12 usec;
    HorizontalBackPorch = 2.23 usec;
 }

Field
{
    Vertical Sync =
    {
        {
            /*
             * Sync goes low here (at beginning of the
             * line, time = 0.0) but does not recover to the
             * high state until the first line of Vertical
             * Back Porch.
             */
            Length = 1.0H;
            Low = 0.0 usec;
        }
        repeat 2 {
            /* Two lines with no transitions */
            Length = 1.0H;
        }
    }

    Vertical Back Porch =
    {
        {
            /*
             * Only one transition:  sync goes high at the
             * time 1.12 usec (HorizontalSync) into the
             * frame.
             */
            Length = 1.0H;
            High = HorizontalSync;
        }
        repeat 34 {
            /*
             * Normal horizontal sync.  Goes low at time=0.0,
             * at beginning of the line.
             */
            Length = 1.0H;
            Low = 0.0 usec;
            High = HorizontalSync;
        }
    }

    Active =
    {
        repeat 1024 {
            /* Normal horizontal sync */
            Length = 1.0H;
            Low = 0.0 usec;
            High = HorizontalSync;
        }
    }

    Vertical Front Porch =
    {
        repeat 3 {
            /* Normal horizontal sync */
            Length = 1.0H;
            Low = 0.0 usec;
            High = HorizontalSync;
        }
    }
}
```

The language in Example 5-1 is standard; the format is that of the Silicon Graphics standard high-resolution monitor. The layout of the information in a format must follow a special sequence.

Example 5-2 Standard Layout of Video Source File

```
General {
    /* Overall Description */
}

Active Line {
    /* The components of the horizontal blanking */
}

Field {
    /* Description of sync pattern for a field */
}

Field {
    /* Description of sync pattern for a field */
}
```

Example 5-2 shows the standard layout of the source file.

- The *General* section describes the overall geometry of the frame. For details on this section, see “Assignment Statements” on page 35.
- The *Active Line* section describes the horizontal blanking section of an active line. You can read about this section in “The Active Line Section” on page 42.
- The *Field* section describes the pattern of sync as it transitions between high and low. You can add as many field sections as needed to describe your format, one for each field. See “The Field Description” on page 43.

Execution Order

The compiler executes your video source program in the order as presented in Example 5-2:

1. The General section.
2. The Active Line section.
3. The Field sections, in the order found in the source file.

Assignment Statements

The compiler uses assignment statements to derive much of the simple parametric information in a video format.

You specify the values in the General section via a series of assignment statements using this form:

```
variable = value;
```

The assignment statement is one form of an expression (the formal definition of the "expression:" is in Appendix A, "Native Language Grammar"). The name of the variable you are using depends on the section you are using. Refer to "The General Parameters Section" on page 38, "The Active Line Section" on page 42, and "The Field Description" on page 43 for details on the variables required.

Time Expressions

Many variables in the compiler deal with absolute or relative positions in the video frame or with quantities of time. The compiler has special time expressions to deal with specifying these values.

Permissible Syntax

- Number of pixels. You must use the `pixels` suffix when specifying this. For example:

  ```
  35 pixels
  ```

- Number of seconds. You must use the `sec` suffix for seconds, the `msec` suffix for milliseconds (10e-3 seconds), or the `usec` suffix for microseconds (10e-6) seconds. For example:

  ```
  0.3 sec
  55.2 msec
  4.75 usec
  ```

- Number of lines. You must use the `lines` or the `H` suffix for to specify lines, as in:

  ```
  5 lines
  0.5 H
  ```

- Number of clock ticks. This expression, not normally used except within Silicon Graphics, specifies the duration of hardware clock ticks. The duration of the clock is often related in some way to the pixel frequency; however, it is dependent completely upon the hardware on which your format will execute. An example of the syntax is:

  ```
  38 clocks
  ```

  It makes no sense to specify a non-integer number of clocks; it will be quantized to the nearest clock.
- The sum of two time elements:

  ```
  (38 usec + 5 pixels)
  ```
- The difference of two time elements:

  ```
  (2 lines - 3 usec)
  ```
- The product of time multiplied by a scalar:

  ```
  (0.5H + 0.29 usec) * 5.5
  ```
- The quotient of a division of time by a scalar:

  ```
  (25 lines + 3 pixels) / 2.0
  ```

For a formal definition of how to specify time units, see “constant-time:” in Appendix A, “Native Language Grammar.”

Quantization

You will usually find it more satisfactory and more transportable to use expressions specifying durations in seconds rather than in pixels or lines. The video format compiler quantizes time expression in seconds to the nearest clock group multiple without complaining; however, when quantizing time units specified in pixels or lines, the compiler will report warnings if the exact position cannot be achieved.

For details on quantization of the pixel-to-clock ratio and quantization, see “The Pixel-to-Clock Ratio” on page 24.

User-Defined Variables

You can use your own variables to supplement the compiler's variables. User-defined variables sometimes make calculations easier; they also allow you to communicate information from one section to another.

You must always define a variable before you use it. The general form for variable definition is as follows:

`[` *storage-class* `]` *data-type variable-name* `[ =` *value* `] ;`

The items surrounded by square brackets ([ ]) are optional. For a formal definition, see "compound-statement:" in Appendix A, "Native Language Grammar." If you define your own variables, the definition must come before any executable statements in the compound statement block.

Data Types

You have your choice of one of the following data types when defining a variable:

- Integer—a 32-bit integer quantity. The data type is `int`.
- Double—a double similar to the double in the C language. The data type is `double`.
- Time—a specific time in the frame. You specify the data type as `time`. Time variables can be specified in the units as described in "Time Expressions" on page 35.
- String—a variable-length string. You specify the data type as `string`. You assign to string variables with double-quoted strings, as in `"High-Resolution Format"`.

Storage Classes

The video format compiler has two storage classes of variables, each with its own scope and associated lifetime:

- Automatic Variables—These have the scope of the most tightly-enclosing compound statement, a set of curly brackets (`{ }`); the lifetime of the variable is the time in which the block is executed. This is the default storage class when you do not specify one explicitly. For example, the time variable *thirdPoint* is declared below as an automatic variable:

  ```
  time thirdPoint;
  ```

- Exported Variables—these have scope across the entire source input file and lifetime that becomes valid when the variable is first parsed and continues until the end of the program. You must specify a storage class of `exported`, as in the declaration below of the integer variable *syncCount*:

  ```
  exported int syncCount;
  ```

  The lifetime of exported variables is important: even before the section of source code which defines the variable is executed, the variables are detected and instantiated.

You can treat user variables as you would any predefined system variable in the program.

Scope and Lifetime

If you do not have much experience with programming languages, you may not be familiar with the terms *scope* and *lifetime*. Simply put, scope is the region of the program where the variable is valid; lifetime is the duration when the variable contains valid data.

The General Parameters Section

The General section of the video format source file gives overall information about the format. The General section is executed first so that it provides information about the format to the rest of the compiler.

You provide the information in the General section in the general form shown in Example 5-3.

Example 5-3 Example of General Section Layout

```
General {
    /* Overall Description */
    assignment statement
    assignment statement
    assignment statement
    ...
}
```

Variables

The assignment statements specify the values of several system-defined variables. If you are looking for detail on assignment statements, see “Assignment Statements” on page 35; the variables are listed in Table 5-1.

Table 5-1 Values Specified in the General Section

Variable Name	Meaning	Optional
FieldsPerFrame	Integer. The number of fields in the video format. A field is a contiguous set of active lines surrounded by a vertical blanking interval. If a format has active video lines that cease and restart later, the format has more than one field.	No
FramesPerSecond	Double. The number of frames to be displayed per second, to the best resolution of the hardware. The video format defines one frame; thus, the entire video format repeats at the rate specified by this variable. This value is used to establish the pixel clock.	No
TotalLinesPerFrame	Integer. The number of lines in the video frame, including lines in vertical blanking. This value is used to establish the pixel clock.	No
TotalPixelsPerLine	Integer. The number of pixels in each line of video, including pixels in horizontal blanking. This value is used to establish the pixel clock.	No

Table 5-1 (continued) Values Specified in the General Section

Variable Name	Meaning	Optional
ActiveLinesPerFrame	Integer. The number of lines in the frame buffer used by the video format. Note that this may not be the same as the total number of active lines in the frame: for single-field formats, the number of active lines in the field is the same as that drawn from the frame buffer; for multiple-field formats that are interlaced, this is the total number of lines in all fields; for multiple-field formats which repetitively fetch from the same frame buffer space for each field (such as stereo or field sequential color formats), this is the number of lines in only one of the fields. In the case of formats that contain active lines that are only a portion of a line (such as that of the NTSC and PAL half-lines), you must round each half line up to the next whole-line size. Formally stated, set this variable to the whole number of lines from which pixels are extracted, regardless of whether a whole line or just a partial line of pixels is extracted. The mechanism you use to tell the compiler whether active lines are half lines is described in "The Field Description" on page 43.	No
ActivePixelsPerLine	Integer. The number of pixels in one line of the frame. If the format has fields that may begin or end with half lines, supply the length of the whole line.	No
FrameBufferHeight	Integer. The number of lines this format occupies in the frame buffer. This variable defaults to the value to which this variable should normally be set: *ActiveLinesPerFrame*. However, some video formats occupy irregular frame buffer footprints; in that circumstance, you should set this variable to aid frame buffer layout software.	Yes
FrameBufferWidth	Integer. The number of pixels this format occupies in the frame buffer. This variable defaults to the value to which this variable should normally be set: *ActivePixelsPerLine*. However, some video formats occupy irregular frame buffer footprints; in that circumstance, you should set this variable to aid frame buffer layout software.	Yes

Table 5-1 (continued) Values Specified in the General Section

Variable Name	Meaning	Optional
ScreenHeight	Integer. The number of lines this video format occupies on a video monitor. This variable defaults to the value in the variable *ActiveLinesPerFrame*. You should set this variable if the screen height differs from the default.	Yes
FormatName	String. A descriptive name for the video format.	Yes
FullScreenStereo	Integer (acting as boolean). This is a special-purpose variable used within Silicon Graphics engineering in limited circumstances; users need never use it. This variable specifies the format is to be used in conjunction with a special fetch sequence from the frame buffer to provide full-screen stereo capabilities.	Yes, defaults to false.

See Example 5-1 for an example of actual use of the General section.

The Pixel Rate

The *pixel rate* of the video format is expressed in pixels per second, the aggregate bandwidth of all pixels flowing from an output port of the computer system. The formula is as follows:

```
TotalLinesPerFrame * TotalPixelsPerLine * FramesPerSecond
```

The video output hardware of your computer system has a maximum pixel rate at which it can operate. Refer to documentation on your particular system to determine the maximum rate.

Within the compiler, the pixel rate is needed to perform conversions between time and pixels. Thus, the General section is executed before any other section so conversions can be computed.

The Active Line Section

You describe the composition of the horizontal blanking in the Active Line section. It is called *Active Line* because these parameters describe the behavior of the horizontal blanking region on lines that contain active pixels. If you need a review of the components of the horizontal blanking region, "The Horizontal Line" on page 12 describes its different sections. The active line section allows you to describe it.

You specify the Active Line section as shown in Example 5-4.

Example 5-4 The Active Line Template

```
Active Line {
    /* Active Line Component Description */
    assignment statement
    assignment statement
    assignment statement
    ...
}
```

The assignment statements specify the system-defined variables that describe the length of the components of horizontal blanking. The variables are listed in Table 5-2.

Table 5-2 Values Specified in the Active Line Section

Variable Name	Meaning	Optional
HorizontalFrontPorch	Time. The length of time of the horizontal front porch of an active line.	No
HorizontalBackPorch	Time. The duration of the horizontal back porch of an active line.	No
HorizontalSync	Time. The duration of the horizontal sync pulse.	No

Specify the durations of each of the components in units of seconds, milliseconds, or microseconds if possible. Placement on a specified pixel can lead to quantization messages; see "Quantization" on page 36.

See Example 5-1 for an example of actual use of the Active Line section.

The Field Description

When writing a video format, you must describe each field individually. As described in "The Field of the Format" on page 20, a field consists of contiguous lines of video surrounded by vertical blanking. Therefore, each time active lines cease and start again in a format, you must define a new field.

To define the field, specify each transition of sync in the frame. You need not specify the active section because it is implied by the values you specified earlier (as shown in "The Active Line Section" on page 42); however, you must individually describe the excursions between low, high, and tri-level of the sync signal (see "Level of Sync" on page 19 for definitions of these levels).

For example, Figure 3-3 on page 12 shows an excerpt of a few lines of a typical frame of video. If this picture were descriptive of the format you wish to write, you would need to describe sync going from its high position to low position, then the low-to-high transition; you would need to describe these two transitions for each of the blanking regions in the diagram—and for the rest of the frame.

A field contains these general classes of description:

- The field components—the major vertical sections of the field, as described in "Field Components" on page 44.
- The field attributes—additional information, not related to timing, that describes a field. These are described in "Field Attributes" on page 48.

Field Components

The components of the field, as described in “The Vertical Interval” on page 16, are as follows:

- Vertical sync.
- Vertical back porch.
- Vertical active.
- Vertical front porch.

You must define component in each field in the followering order.

Example 5-5 Field Composition

```
Field
{
    Vertical Sync =
    {
        sync transition set
        sync transition set
        sync transition set
        ...
    }

    Vertical Back Porch =
    {
        sync transition set
        ...
    }

    Active =
    {
        sync transition set
        ...
    }

    Vertical Front Porch =
    {
        sync transition set
        ...
    }
}
```

Example 5-5 shows an example of the layout you might use. The grammar is formally described in “field-definition:” in Appendix A, “Native Language Grammar.” For a real example, see Example 5-1.

You may place only sync transition sets within each of the field components. You may use as many sync transition sets as needed to describe the format. For information on this part of the language, see “Sync Transition Set” on page 45.

Be especially careful to place the proper active lines in the Active section: only and all lines placed in the active component will have active pixels on them. If the first or last line in the Active section is not a whole line (such as with the NTSC and PAL half lines), only part of the output line will have active pixels.

Sync Transition Set

The sync transition set defines a set of transitions of the video sync signal on a line or portion of a line. The length of the defined line portion is a required statement, the transitions are optional. That is, it is possible to define a line with no transitions of sync.

Example 5-6 Simple Sync Transition Sets, Whole Lines

```
{
    /* Normal horizontal sync, one repetition */
    Length = 1.0H;
    Low = 0.0 usec;
    High = 1.19 usec;
}

repeat 3 {
    /* Normal horizontal sync, three rep. */
    Length = 1.0H;
    Low = 0.0 usec;
    High = 1.19 usec;
}
```

Two simple sync transition sets appear in Example 5-6, each set delimited by curly brackets (`{ }`). All repetitions are concatenated to the previous set, so you must specify the sync transition sets in the order in which they are to appear in the frame.

Example 5-7 Simple Sync Transition Set, Half Lines

```
repeat 6 {
    Length = 0.5H;
    Low = 0 usec;
    High = 27.1 usec;
}
```

A single simple transition set is in Example 5-7.

Repeat

The first sync transition set in Example 5-6 is executed once, while the second sync transition set is repeated three times; otherwise they are identical. One could have shortened the text of the definition simply by omitting the first set and specifying the repeat value of the second set as `repeat 4` instead of the two separate definitions.

The sync transition set shown in Example 5-7 also has a repetition factor—it is executed six times.

If you do not specify an explicit repetition count, a sync transition set is executed once.

The repetition count can be an integer expression. See the formal grammar of "sync-transition-multiplier:" in Appendix A in for a formal treatment.

Length

The length of the transition set is specified with an assignment to the *length* variable with a duration, as in this example:

```
length = 0.5 lines;
```

The length of the two sync transition sets of Example 5-6 are both one line, so the total length of all the sets in that example is four lines (1 line + 3(1 line) = 4 lines).

The length of the set in Example 5-7 is one half-line; when the compiler performs the six repetitions, its total length is three lines.

Example 5-8 Sync Transition Set Length

```
repeat 100 {
    Length = 1.0 H;
    sync transitions...
}
```

The sample source code in Example 5-8 shows 100 repetitions of a one-line set. Regardless of what changes in the sync signal on the line, the source of this example creates 100 lines of video.

Thus, the length specified is not dependent on any of the sync-level transitions. In fact, it is possible to have a sync transition set with no transitions at all, just a length. For example, see the vertical sync pulse of Figure 3-8; the pulse is more than three lines long, so it has no transitions at all for two lines. The source file for this is shown in Example 5-1, where the Vertical Sync component has two transition sets: one in which the sync transition goes low, followed by two repetitions of the set where no transitions occur at all—only the length.

Sync Level

The sync-level statements specify the time at which the sync transition changes level (sync levels are defined in "Level of Sync" on page 19). You use this form:

level `=` *time-expression*`;`

The *level* can be `high`, `low`, or `tri`.

The time-expression is a time constant (see "Time Expressions" on page 35) or a time variable. When you use time variables, you can make your source program easier to read and less prone to error. In so doing, you document why sync is making a transition (by using a descriptive name) and make cut-and-paste errors less likely than if you explicitly use time values (because the text of variable names cannot be corrupted without the compiler reporting an error).

For an example of using variables, see Example 5-1. It uses the variable *HorizontalSync*, described in "The Active Line Section" on page 42. In that example, most of the sync transitions (all but those in the vertical sync) make a transition to low at the beginning of the line (0.0 usec). The sync transitions that define the point at which sync make the transition to the high state do not specify a time constant but instead refer to the *HorizontalSync* variable.

Field Attributes

Each field can have attributes assigned individually to it. These attributes describe features of the frame not related to timing, such as how pixels should be fetched from the frame buffer.

Each attribute is set before any of the components of the field are specified. The components of the field are Vertical Sync, Vertical Back Porch, Vertical Active, Vertical Front Porch (see "Field Components" on page 44). The attributes are set via an assignment statement of this form:

attribute `=` *value*`;`

You can assign the attributes directly, or not specify them and allow the compiler to use default values.

Table 5-3 Field Attributes

Attribute	Description	Default for each field
`skip`	Also known as *stride*, this attribute specifies how lines should be fetched from the frame buffer. The value specifies the number of lines that should be skipped in order to determine location of the next line for this field. To fetch the next consecutive line, skip is set to `1`; this is the case for progressive-scan single-field formats. To fetch the line after the next consecutive line, skip should be set to `2`.	`1`
`offset`	This attribute specifies the number of lines from the top of the frame at which this field should begin fetching. To begin fetching a field's pixels at the top line of the allocated frame buffer, set offset to `0`. To fetch beginning at the second line of the frame buffer, set offset to `1`.	`1`

Table 5-3 (continued) Field Attributes

Attribute	Description	Default for each field
`swap`	Use this attribute to control whether the graphics subsystem should enable frame swap during the corresponding field (i.e., should allow the hardware to swap from front to back buffers when the program calls the **glXSwapBuffers**(3G) function). To enable a swap for a field, set swap to `true`; to disable swap for a field, set swap to `false`. Why enable instead of disabling swap on different fields of a multi-field format? If you want inter-field motion, you will need to swap on each field. This motion is useful for broadcast-type and the RS-170 formats, which have pretty slow frame rates. However, for stereo formats, swapping between fields would yield motion between left and right eyes (i.e., different images presented to each eye); this could make for a nauseating display. Field sequential monitors have plenty of tearing between color fields on a good day. You probably do not want to make it worse by presenting different images on color fields; you would be deliberately causing misalignment.	`true`
`eye`	This attribute is used for stereo (binocular) monitors. It specifies for which eye the pixels in the field should be fetched. You can specify `{Left}` or `{Right}`; you may also specify a comma-separated list.	`{ Left, Right }`
`color`	This attribute is used for color field-sequential monitors. It specifies which colors should be fetched from the frame buffer for the field. You can specify `{Red}`, `{Green}`, or `{Blue}`; you can also specify a comma-separated list.	`{ Red, Green, Blue }`

Chapter 6

Examples of Native Compiler Language

The description of the language itself in Chapter 5, “Native Compiler Language,” is a starting point for learning about the language of video formats. The examples in this chapter will help solidify your understanding.

Read the next section, “Using Examples” on page 51, before proceeding to read the examples. Each of these examples is presented in source form and has brief remarks describing the important points:

- “Interlaced Format” on page 52 shows a two-field interlaced format.
- “Stereo Format” on page 55 shows a stereoscopic video format.
- “Color Field Sequential” on page 59 shows a video format for a color field sequential monitor.

You may also be able to find examples on the Silicon Surf web site and on the developer’s CDs. There is not a lot of interest in the general developer community (sniff!), but we get space from time to time.

Using Examples

You will rarely need to write a format completely from scratch. Most people who write formats find a format similar to what they are writing and modify it to suit their needs. If you can, you should do the same.

Many of the examples that accompany this book (provided as part of the vfc installation) and are in the directory */usr/gfx/ucode/common/vfc/vfs/*. Refer to those files when writing your own format.

Interlaced Format

This format is similar to the PAL-I timing format. Its two fields interlace spatially, as described in “Interlaced Formats” on page 20.

Example 6-1 Interlaced Format

```
/*
** 768x576_25i.vfs - RGB PAL standard
*/

General
{
    exported time SerrationDuration;
    exported time EqualizationDuration;

    FieldsPerFrame = 2;
    FramesPerSecond = 25;
    TotalLinesPerFrame = 625;
    TotalPixelsPerLine = 944;
    ActiveLinesPerFrame = 576;
    ActivePixelsPerLine = 768;
    FormatName = "PAL";

    SerrationDuration = 27.3 usec;
    EqualizationDuration = 2.35 usec;
}

Active Line
{
    HorizontalFrontPorch = 1.3 Usec;
    HorizontalSync = 4.7 Usec;
    HorizontalBackPorch = 5.96 usec;
}

Field
{
    swap = true;
    skip = 1;
    offset = 0;

    Vertical Sync = {
        repeat 5 {
            Length = 0.5H;
            Low = 0 usec;
            High = SerrationDuration;
        }
    }
```

```
        Vertical Back Porch = {
            repeat 5 {
                Length = 0.5H;
                Low = 0 usec;
                High = EqualizationDuration;
            }
            repeat 17 {
                Length = 1.0H;
                Low = 0 usec;
                High = HorizontalSync;
            }
            {
                Length = 0.5H - HorizontalFrontPorch;
                Low = 0 usec;
                High = HorizontalSync;
            }
        }

        Active = {
            {
                /* No sync edge transitions needed here. */
                Length = 0.5H + HorizontalFrontPorch;
            }
            repeat 287 {
                Length = 1.0H;
                Low = 0 usec;
                High = HorizontalSync;
            }
        }

        Vertical Front Porch = {
            repeat 5 {
                Length = 0.5H;
                Low = 0 usec;
                High = EqualizationDuration;
            }
        }
    }

    Field
    {
        swap = true;
        skip = 1;
        offset = 1;
        Vertical Sync = {
            repeat 5 {
                Length = 0.5H;
                Low = 0 usec;
                High = SerrationDuration;
            }
        }

        Vertical Back Porch = {
            repeat 5 {
                Length = 0.5H;
                Low = 0 usec;
                High = EqualizationDuration;
            }
            {
                Length = 0.5H;
            }
            repeat 17 {
                Length = 1.0H;
                Low = 0 usec;
                High = HorizontalSync;
            }
        }

        Active =
        {
            repeat 287 {
                Length = 1.0H;
                Low = 0 usec;
                High = HorizontalSync;
            }
            {
                Length = 0.5H;
                Low = 0 usec;
                High = HorizontalSync;
            }
        }

        Vertical Front Porch = {
            repeat 5 {
                Length = 0.5H;
                Low = 0 usec;
                High = EqualizationDuration;
            }
        }
    }
```

Note the following features of the format shown in Example 6-1:

- The skip and offset attributes describe the monitor's interlacing pattern.
  - The `skip = 1;` of the first field is the same as `skip = 1;` of the second. Both fields interlace similarly by drawing every other line, as described in "Interlaced Formats" on page 20. You can find information on the `skip` attribute in Table 5-3.
  - The starting line at which each field begins differs. The first field uses the attribute `offset = 0;` to indicate that drawing should begin on the first line of the monitor. The second field uses offset = 1; to indicate that the first line of the field is offset one line into the display of the monitor. Table 5-3 describes the `offset` attribute.

  The attributes are only part of the story and tell the graphics pipe how pixels should be fetched from the frame buffer. The synchronizing pulses of the vertical sync components of the field tell the monitor which frame is first.
- This format permits motion between the two fields. This motion is enabled with the `swap = true;` attribute of the first field, echoed by the `swap = true;` attribute of the second field. Were the second field to have its swap value set to false, no swaps would be permitted between fields. See the `swap` attribute in Table 5-3.

Stereo Format

This format describes a two-field stereo format, close to the size of VGA. Monitors showing field-based stereo display slightly different views in two successive fields. In one field, only one eye is permitted to view the image (usually by special glasses); the succeeding field is shown only to the other eye.

The format shown here is known as *new-style* stereo (also known as *per-window* stereo). It differs from *old-style* stereo (or *full-screen* stereo) in that the new style fetches pixels of both fields from the same location in the frame buffer; this is described in "Stereo Formats" on page 21. The old-style stereo formats used pixels in different portions of the frame buffer for each eye's field. The new-style stereo format is shown in Example 6-2.

Example 6-2 Stereo Format

```
/*
** 640x480_120s.vfs - stereo VGA at 60Hz/frame
*/
```

```
General
{
    ActiveLinesPerFrame = 480;
    ActivePixelsPerLine = 640;
    FramesPerSecond = 60;
    FieldsPerFrame = 2;
    TotalLinesPerFrame = 1050;
    TotalPixelsPerLine = 800;
    FormatName = "640x480_120s - 120Hz Stereo VGA";
}

Active Line
{
    HorizontalFrontPorch = 0.397 usec;
    HorizontalSync = 1.190 usec;
    HorizontalBackPorch = 1.587 usec;
}

Field
{
    eye = { Left };
    swap = true;

    Vertical Sync = {
        {
            Length = 1.0H;
            Low = 0 usec;
        }
        repeat 2 {
            Length = 1.0H;
        }
    }

    Vertical Back Porch = {
        {
            Length = 1.0H;
            High = HorizontalSync;
        }
        repeat 31 {
            Length = 1.0H;
            Low = 0 usec;
            High = HorizontalSync;
        }
    }
```

```
        Active = {
            repeat ActiveLinesPerFrame {
                Length = 1.0H;
                Low = 0 usec;
                High = HorizontalSync;
            }
        }

        Vertical Front Porch = {
            repeat 10 {
                Length = 1.0H;
                Low = 0 usec;
                High = HorizontalSync;
            }
        }
    }

    Field
    {
        eye = { Right };
        swap = false;

        Vertical Sync = {
            {
                Length = 1.0H;
                Low = 0 usec;
            }
            repeat 5 {
                Length = 1.0H;
            }
        }

        Vertical Back Porch =
        {
            {
                Length = 1.0H;
                High = HorizontalSync;
            }
            repeat 28 {
                Length = 1.0H;
                Low = 0 usec;
                High = HorizontalSync;
            }
        }
    Active =
    {
        repeat ActiveLinesPerFrame {
            Length = 1.0H;
            Low = 0 usec;
            High = HorizontalSync;
        }
    }

    Vertical Front Porch =
    {
        repeat 10 {
            Length = 1.0H;
            Low = 0 usec;
            High = HorizontalSync;
        }
    }
}
```

Note these features of the format shown in Example 6-2:

- The format has two fields, both of which have similar structure. The significant difference in timing of the two fields is the length of vertical sync: to allow monitor hardware to discriminate between the two fields, the second field's vertical sync pulse is somewhat longer than that of the first (note the difference in the **repeat** count).

  How different do the sync signals of the two fields need to be? This is just an example; unfortunately, monitor manufacturers have not adopted a common standard for recognition of sync signals of the two fields. Check with the documentation that accompanies your monitor for details; because this information is often sketchy, you may need to contact the monitor manufacturer directly. If the monitor is distributed by Silicon Graphics with your system, the company will provide you with whatever information is available.
- The statements `eye = {Left};` in the first field and `eye = {Right};` in the second field are the significant features in the example. They use the `eye` attribute described in Table 5-3.
- Stereo displays should not have motion between the two fields; this would have an unsettling effect on the viewer, potentially destroying the stereo effect. To inhibit swap, the second field uses the statement `swap = false;`. Table 5-3 describes the `swap` attribute.

It is also useful to reduce the swap interval to reduce demand on the rendering hardware of the graphics pipe. If swap were set to true for both fields, the pipe would swap at 120 Hz instead of 60 Hz, halving the time between potential swaps.

Color Field Sequential

Field sequential monitors have special hardware. Instead of displaying all three colors (red, green, and blue) simultaneously, these monitors display only one color at a time. The monitor displays the entire picture once showing only the red pixels, another field only with green, another blue. Each color field is shown at three times the normal rate so thateach frame passes at the normal speed. When the fields pass rapidly, the eye merges the sequential fields successfully. See Example 6-3.

Example 6-3 Color Field Sequential Format

```
/*
** 640x480_180q.vfs - field sequential
*/

#define SER (1.0H-HorizontalSync)

General
{
    FieldsPerFrame = 3;
    FramesPerSecond = 60;
    TotalLinesPerFrame = 1560;
    TotalPixelsPerLine = 880;
    ActiveLinesPerFrame = 480;
    ActivePixelsPerLine = 640;

    FormatName = "Field Sequential 640x480_180q";
}

Active Line {
    HorizontalFrontPorch = 40 pixels;
    HorizontalSync = 80 pixels;
    HorizontalBackPorch = 120 pixels;
}

/*
** red (synchronizing) field
*/
```

```
Field
{
    Color = { red };
    swap = true;

    Vertical Sync = {
        repeat 6 {
            Length = 1.0H;
            Low = 0 usec;
            High = SER;
        }
    }

    Vertical Back Porch = {
        repeat 33 {
            Length = 1.0H;
            Low = 0 usec;
            High = HorizontalSync;
        }
    }

    Active = {
        repeat 480 {
            Length = 1.0H;
            Low = 0 usec;
            High = HorizontalSync;
        }
    }

    Vertical Front Porch = {
        {
            Length = 1.0H;
            Low = 0 usec;
            High = HorizontalSync;
        }
    }
}

/*
** green field
*/
```

```
Field
{
    Color = { green };
    swap = false;

    Vertical Sync = {
        repeat 3 {
            Length = 1.0H;
            Low = 0 usec;
            High = SER;
        }
    }

    Vertical Back Porch = {
        repeat 36 {
            Length = 1.0H;
            Low = 0 usec;
            High = HorizontalSync;
        }
    }

    Active = {
        repeat 480 {
            Length = 1.0H;
            Low = 0 usec;
            High = HorizontalSync;
        }
    }

    Vertical Front Porch = {
        {
            Length = 1.0H;
            Low = 0 usec;
            High = HorizontalSync;
        }
    }
}

/*
** blue field
*/
```

```
Field
{
    Color = { blue };
    swap = false;

    Vertical Sync = {
        repeat 3 {
            Length = 1.0H;
            Low = 0 usec;
            High = SER;
        }
    }

    Vertical Back Porch = {
        repeat 36 {
            Length = 1.0H;
            Low = 0 usec;
            High = HorizontalSync;
        }
   }

    Active = {
        repeat 480 {
            Length = 1.0H;
            Low = 0 usec;
            High = HorizontalSync;
        }
    }

    Vertical Front Porch = {
        {
            Length = 1.0H;
            Low = 0 usec;
            High = HorizontalSync;
        }
    }
}
```

Note these salient features of the format shown in Example 6-3:

- The three fields are similar in timing except for the length of vertical sync (note the difference in the **repeat** count). In the first field, vertical sync is longer. This allows hardware to discriminate between color fields and the beginning of the frame.
- The attribute of importance is that of color. This format differs because the different fields specify `Color = {red};`, `Color = {green};`, and `Color = {blue};`. You can find the description of this attribute in Table 5-3.
- Swaps are enabled only for the first field (`swap = true;`), disabled for the second (`swap = false;`) and third (`swap = false;`). This inhibits motion between the two fields. Table 5-3 describes this attribute.

No Transitions

You may find it necessary to produce a segment of time in your video format where the synchronization signal makes no transitions at all. In Example 6-4, you can see a portion of code extracted from the standard high-resolution format originally introduced in Example 5-1.

Example 6-4 No Transitions

```
Vertical Sync =
    {
        {
            /*
             * Sync goes low here (at beginning of the
             * line, time = 0.0) but does not recover to the
             * high state until the first line of Vertical
             * Back Porch.
             */
            Length = 1.0H;
            Low = 0.0 usec;
        }
        repeat 2 {
            /* Two lines with no transitions */
            Length = 1.0H;
        }
    }
```

In this example:

- The first sync transition set executes only once (no explicit `repeat` statement) and contains only a transition to the low state.
- The second sync transition set executes twice. This set has only a length associated with it. Because no transitions are specified, the time passes without transitions. At the end of this sync transition set, the synchronization signal is still set to the low state.

Chapter 7

Board Hardware Definition

If you are on a video output hardware design team and are defining a new board with all its rules, this chapter. If you are not on such a team, you can skip reading this chapter.

Two general sections make up the board definition:

- The board description in which you specify the board's parameters (see the next section, "General Notes About Writing Definition Files" on page 65). This section is also where you define the names of the signals and describe their relationship to the chip
- The rules definition (see "Special Signal Definition" on page 69). This is where the transitions of all signals are defined.

Given a properly-defined chip definition file, the video format compiler synthesizes the program needed to drive the chip to produce the signals you specify.

General Notes About Writing Definition Files

Variables

Don't miss the touching description of "User-Defined Variables" on page 37. These variables can be defined in the same way and scope as can variables in the C language.

Tip: Pay attention to *exported* variables, as described in "Storage Classes" on page 38. All exported variables are written to the output file and can be used by the software that loads video formats. Exported variables are handy for storing the results of calculations made when producing the video format. These variables are typically calculated in the rules section, but they are also often derived in the post-processing statements, described in "Pre-Processing and Post-Processing Statements" on page 91. Exported variables have a lifetime that begins when the variable is first parsed (not when the code section is executed), so they may be defined in the rules section and set in the format source file.

Statements

To help you deal with complicated situations, you may use “Control Statements” as you would in the C language. Also, you can use “Diagnostic Statements” to warn the user of errors during processing.

Control Statements

You may use the following statements, as you would in the C language:

`if` *(expression) statement*

`if` *(expression) statement* `else` *statement*

`for` *(expression; expression; expression) statement*

Diagnostic Statements

These statements allow you to pass information along to the user. The compiler prints any string you specify as it would any other internally generated messages.

`info` *string-expression*

`warning` *string-expression*

`error` *string-expression*

The `info` and `warning` statements allows compilation to proceed; the `error` statement causes the compiler to terminate before producing an output file and generates an error status code on exit.

Tip: You can build string expressions by using the `+` operator. Any numeric value expressions are converted to string expressions automatically, and expressions of quoted strings to be concatenated are perfectly legal.

Board Description

This section explains how to describe the board in the board definition file.

- The first part of this section of the board description file shows the different parameters you can use to describe the board; see "Parameter Definition" on page 67.
- The second part of this section of the board description file defines signal names; see "Board Signal Assignment" on page 68.
- The next part of the board description file associates signals with special attributes; see "Special Signal Definition" on page 69.

Before writing the rules, you should carefully describe your board. A thorough description makes the compiler work harder for you.

Parameter Definition

These variables describe the way in which video is generated on your board. Some are used for programming, others for bounds checking. The variables are in Table 7-1.

Table 7-1 Board Hardware Definition Variables

Variable Name	Description	Required
VideoClockRatioNumerator	Integer. Part of the description of the pixel-to-video clock ratio. This value describes the number of pixels in the ratio.	Yes
VideoClockRatioDenominator	Integer. Part of the description of the pixel-to-video clock ratio. This value describes the number of video clocks in the ratio.	Yes
SystemVideoClockRateMaximum	Double. The highest rate at which the video clock operates. This compiler reports an error if the user's format exceeds this rate.	Yes
SystemVideoClockRateMinimum	Double. The lowest rate at which the video clock operates. This compiler reports an error if the user's format is less than this rate.	Yes

Table 7-1 (continued) Board Hardware Definition Variables

Variable Name	Description	Required
DACRateMaximum	Double. The highest rate at which the output DAC operates. The compiler reports an error if the user's format exceeds this rate.	No
DACRateMinimum	Double. The highest rate at which the output DAC operates. The compiler reports an error if the user's format is less than this rate.	No
DisplayArchitecture	String. The display architecture on which generated *.vfo* files will successfully operate. For example, `MG_VC3` for MardiGras VC3 ASIC (IMPACT or Octane), `O2_GBE` for O2 GBE ASIC, or `IR` for InfiniteReality.	Yes
OptionHardware	String. The option hardware on which generated *.vfo* files will successfully operate. For example, `Presenter`, `DDO2`, or `GVO`.	No
VfoVersion	String. The version of the rules file. Can be used during loading to constrain which versions of builds may be used.	Yes

Board Signal Assignment

The chip definition (not described in this book) supplies the chip's names for each signal it can generate. Typically, these are the names of the pin-out of the chip (or may just be names the chip uses internally). Because a single kind of format generation chip can be used on more than one board, these names often do not have particular meaning (or can sometimes have meaning other than their true use).

The signal name definition statement associates a chip's signal name with a name meaningful on the board. It is that board signal name that you can use within the rules when you assign transitions.

You must have a board signal assignment statement for each signal you intend to use. Although you need not use every signal on a chip, you must make the assignment for those you do intend to use (even if the name of the chip and the board are identical).

The syntax of the statement is as follows:

`signal` *board-signal-name* `uses chip hardware signal` *chip-signal-name;*

- The *board-signal-name* is any string (usually in double quotes) you choose. The name should be meaningful to application on the board; you will use this name in the rules section (see "Rules Definition" on page 70).
- The *chip-signal-name* is the name assigned by the writer of the chip definition file. You must specify it exactly as presented there.

Special Signal Definition

The compiler requires some additional information about how your output signals are used when running. These signals drive the frame analysis process which is in turn used to provide information for the **-a ascii** option (for more information, see vfc(1)). Frame analysis is also used to drive the rules section of the compiler (see "Rules Definition" on page 70).

Each special signal definition specifies the signal to fill each role. The general form is

signal-type `is` *board-signal-name* `active` *direction* `;`

The *signal-type* is one of those detailed in the following paragraphs. The *board-signal-name* is the same name you specified in the board signal assignment (see "Board Signal Assignment" on page 69). The direction specifies the polarity of the signal when active (either `high` or `low`).

The following are board signal assignments:

- Composite Sync Signal

  `composite sync signal is` *board-signal-name* `active` *direction* `;`

  The composite sync signal is very important for frame analysis. The signal you designate is used for determining the positions of all frame portions.
- Active signal

  `active signal is` *board-signal-name* `active` *direction* `;`

  The active signal is used for frame analysis to determine on which lines active pixels lie (those containing picture content — see "Architecture of a Video Frame" on page 10 for details if you need them). The active lines are used to determine the horizontal blanking interval values as well as to determine the number of fields in the format (by counting the clusters of active lines).

- Trilevel sync signal

  `trilevel sync signal is` *board-signal* `active` *direction* `;`

  The trilevel sync specifies which, if any, of the output signals is used for tri-level sync. If you define this signal, the compile defines the user tri-level signal for you.

- Pixel requesting signal

  `pixel requesting signal is` *board-signal* `active` *direction* `;`

  If your board has a signal that corresponds to the transfer of video pixels, you should specify it in this statement. The signal need not correspond to the time at which the format would have the pixels transfer, but the duration in clocks must be correct as measured by *VideoClockRatioNumerator* and *VideoClockRatioDenominator* (see Table 7-1).

  If you set this signal, the compiler can verify that no rounding errors have changed the correct duration of the pixel requesting signal.

Rules Definition

This section of the definition file is where you can define the signal transitions necessary to operate a video board. The master timing reference is the user's video format: all signals reference the user's format for their temporal positions.

- The compiler stores all transitions in an internal form called the Edge Database. You may find the discussion of "The Edge Database" on page 71 helpful in writing rules.
- Your principal weapon for altering signals is "The set signal Statement for Transitions" on page 71.
- Read "The within Statement for Location" on page 73 to determine where signal transitions should be set.
- The compiler offers some pre-defined signals you can copy or simply check. Those are described in "The User Signals for Reference" on page 83.
- The compiler defines some variables for your use as well. Check "System-Defined Variables" on page 85 for instructions.

The Edge Database

All signal transitions in the format are placed in the edge database. This includes the transitions the user specifies in the format (see “The User Signals for Reference” on page 83) and the transitions on signals that are defined in the rules (see “The set signal Statement for Transitions” on page 71). The within statement (see “The within Statement for Location” on page 73) uses the edge database for reference with some signals.

The edge database is resident in the compiler until the compiler terminates. Because the edge database is always ready for your use, you can reference it within pre-processing and post-processing statements (see “Pre-Processing and Post-Processing Statements” on page 91).

The edge database stores transitions in time units at the maximum internal resolution of the edge database. Transitions are not quantized to the clock period until they are extracted from the database. Delay of quantization attempts to minimize successive quantization error when dealing with absolute time units.

Initial State

The *initial state* of the signal is the state of the signal at the beginning of the frame. You have two choices: `high` or `low`. Syntax for the statement is as follows:

`signal` *board-signal-name* `initial state` *direction* ;

Some chip hardware has signals for which the initial state is programmable; the compiler can set the initial state properly if you use this signal. Other hardware may have fixed initial state requirements; the compiler can check to be certain your initial state setting is compatible with hardware.

The set signal Statement for Transitions

Use the `set signal` statement to describe signal transitions on any signal you generate with the video format compiler. This enters the transition in the edge database (for more information on this part of the video format compiler, see “The Edge Database” on page 71).

Syntax for this statement is as follows:

`set signal` *board-signal-name transition-direction* `at` *time-expression* [*error-handling*];

The following are components of this statement:

- signal-name

  The signal-name you use is any of the signals you have defined in the board signal assignment (see “Board Signal Assignment” on page 69).

- transition-direction

  The transition direction can be either `high` or `low`, similar to the manner in which sync signals are defined in “Level of Sync” on page 19.

  You may use a variable instead of these predefined keywords; define an integer variable with the value you choose and use the variable for the direction. If you are copying the polarity of the innermost range of the scope, you can use `CorrespondingPolarity`.

- time-expression

  You can use any of the time expression specified in “Time Expressions” on page 35. Because you will probably be using `within` statements most of the time, make ample use of the `BeginTime` and `EndTime` variables the compiler defines for you (see “How You Can Use Range Information” on page 75).

  If the time you specify is at a point before the beginning of the frame or after the end of the frame, the compiler issues a warning message, but wraps the time to the proper point in the current frame.

- error-handling

  Error handling is an optional clause of the set signal statement. You need set it only under circumstances that require it.

  Only one error handling clause is defined at present:

  – `ignoring wrap error`—This clause instructs the compiler not to issue a warning if the *time-expression* is at a point before the beginning of the frame or after the end of the frame.

The within Statement for Location

The `within` statement gives you reference points to the architecture of a frame. You specify a time range in which you are interested, and the compiler supplies the variables, giving you the time position in the frame of that range. Given those variables, you can use `set signal` to set transitions.

The compiler has two forms of the *within* statement:

- `within` (*range-expression first-expression*, *last-expression* [; *warn-expression*]) *board-rules-statement*
- `within each` (*range-expression* [; *warn-expression*]) *board-rules-statement*

Both statements use a *range-expression* to specify which portion of the frame should be iteratively selected. The range expressions, explained in detail in the following sections, permit you to specify logical portions of the frame (for example, vertical front porch, Horizontal Sync, and so on); transitions of previously-defined signals; and absolute periods of time.

You should understand these concepts—vertical front porch, Horizontal Sync, and so on—before you begin. If you need a brief overview or if you want to know the compiler's definition of many of these terms, see Chapter 3, "Building Blocks of a Video Format."

The `within` statement has clauses that permit you to specify which of the range expressions you wish to choose by specifying the first and last instances of that range, inclusive. Instances are one-based: the first instance is *1*, the second instance is *2*, the last instance is "number of lines." For example, the following would iteratively select lines two, three, and four:

```
within (line 2,4) ...
```

The `within each` statement is a special case of the `within` statement. The `within each` iteratively chooses every instance of the specified range. For example, the following statement iteratively selects every horizontal front porch defined in the format:

```
within each (Horizontal Front Porch) ...
```

Nesting Statements

Statements may be nested, with each statement successively narrowing the scope. For example, the statement within Example 7-1 has narrowed its scope to the second line of vertical back porch of the first field.

Example 7-1 Nested Board Rules Statements

```
within (field 1,1)
    within each (Vertical Back Porch)
        within (line 2,2)
             ...
```

In Example 7-1, the following are successively selected:

1. In `within (field 1,1)`, the `within` statement selects the first field by specifying the range beginning with field 1 and ending with field 1. Remember, instances are one-based.
2. In `within each (Vertical Back Porch)`, this statement selects the time span embodied with the vertical back porch of field 1 (the field being narrowed in the previous statement).

   Note: Although "within each" would normally iterate through all instances of this item within the parenting scope, only one vertical back porch exists for any field (so the additional restriction is not necessary).
3. In `within (line 2,2)`, the statement limits the scope to line 2 (both beginning and ending with 2) of the previous scope (which had been narrowed to vertical back porch of field 1).

By successively narrowing the scope, the above figure has selected a time span that starts with the beginning of line 2 of vertical back porch of field 1 and ends with the end of line 2 of vertical back porch of field 1.

A nested scope can never exceed the bounds of a parenting scope. If a nested range expression specifies a range greater than the range of the parenting scope, the scope is simply limited to the parenting scope.

It is possible to narrow a range such that no time remains in the innermost scope as in Example 7-2.

Example 7-2 Narrowing Scope So No Time Remains

```
within each (Vertical Back Porch)
    within each (Vertical Front Porch)
        ...
```

Because vertical front porch and vertical back porch define mutually exclusive time spans within the frame, the innermost range specifies no time span at all and is not a valid scope.

How You Can Use Range Information

When in the bounds of a board rules selection, three variables become active. Their values depend on the format as specified by the user and by previous signal definitions.

- `BeginTime`—returns the starting time of the innermost range. This variable is of type time.
- `EndTime`—returns the end time of the innermost range. This variable is of type time.
- `CorrespondingPolarity`—returns the polarity of the innermost range. This variable is of type integer.
- `FramePortionIndex`—returns the index of the within item iteration (that is., *0* for the first execution of the within block, *1* for the second execution, and so on). This variable is of type integer.

Each range expression described in Table 7-2 sets values uniquely, according to its rules and bounding scope.

Table 7-2 Range Expressions

Expression Syntax	Description	Start Time	End Time	Polarity
`frame`	The entire frame. This is provided as a convenience and is not strictly necessary because it is the default range. (See "Default Range" on page 83.)	Zero seconds	Duration of the frame.	Positive
`field`	The field of the frame. The user defines the number of frames in the field via the variable FieldsPerFrame. (See "The General Parameters Section" on page 38).	The beginning of vertical sync for this field	The end of vertical front porch for this field	Positive
`line`	The whole line boundary. Line durations are defined by the user's format via this formula: $\frac{1}{FPS \times TLPF}$[a] FPS is set by the user via FramesPerSecond and TLPF is set by the user via TotalLinesPerFrame. (These variables are described in "The General Parameters Section" on page 38.)	The beginning of the whole line (as calculated in Description)	The end of the whole line (as calculated in Description)	Positive

Table 7-2 (continued) Range Expressions

Expression Syntax	Description	Start Time	End Time	Polarity
`pixel`	The whole pixel boundary. Pixel durations are defined by the user's format via this formula: $\frac{1}{\mathrm{FPS} \times \mathrm{TLPF} \times \mathrm{TPPL}}$ FPS is set by the user via FramesPerSecond, TLPF is set by the user via TotalLinesPerFrame, and TPPL is set by the user via TotalPixelsPerLine. (These variables are described in "The General Parameters Section" on page 38.) The start point of pixel is calculated from the beginning of the frame, not the beginning of the period; therefore, it is possible to have a nested pixel time range quantized to a point not coincident with its parent time range. The user directly defines the total number of pixels in the frame via this equation (using the previously defined variables): $\mathrm{TLPF} \times \mathrm{TPPL}$	The beginning of the whole pixel (as calculated in the formulain the description column).	The beginning of the whole pixel (as calculated in the formula in the description column).	Positive

Table 7-2 (continued) Range Expressions

Expression Syntax	Description	Start Time	End Time	Polarity
`Vertical Front Porch`	The first of three portions of the vertical blanking interval. One vertical blanking interval exists for each field in the frame. The vertical front porch for the first field may be defined following the active portion of the last field, yet still be associated with the first field. For formats that employ military sync (see definition in the `Vertical Sync` entry of this table on page 79), vertical front porch has a duration equal to zero. There is one vertical front porch in each field.	These choices are possible: 1) If the format, such as NTSC and PAL, has an intra-line pulse (that is, a pulse that does not fall at the beginning of a whole line from the beginning of the frame), time begins at the start of that pulse. 2) If the format has no intra-line pulse, the time at the beginning of the first line following the last active line of the frame.	The time at the leading edge of the vertical sync pulse. If this format contains more than one vertical sync pulse, the end of the vertical front porch is the leading edge of the first pulse	Positive

Table 7-2 (continued) Range Expressions

Expression Syntax	Description	Start Time	End Time	Polarity
`Vertical Sync`	The second of three portions of the vertical blanking interval. This is often considered the true beginning of the frame. A vertical sync pulse is defined to be a pulse that is longer than a horizontal blanking pulse (found on active lines). Three types of vertical sync patterns are possible: 1) Commercial sync (typical for broadcast) that contains some number of equalizing pulses preceding sync pulses (in vertical front porch); vertical sync, with serration pulses; equalizing pulses following sync pulses (in vertical back porch). 2) Block sync (such as the Silicon Graphics standard high-resolution format) with a single long pulse that may span several lines. See Chapter 2 for a discussion of block sync. 3) Military sync, which has no vertical synchronization pulses and simply has a period of time without active lines. Each field contains exactly one vertical sync.	These choices are possible: 1) For commercial and block sync, the time of the leading edge of the first vertical sync pulse. 2) For military sync, the time of the first full line without active picture.	These choices are possible: 1) For commercial sync, the time of the beginning of the first whole line containing no vertical sync pulses (that is, long pulses). 2) For block sync, the beginning of the first line where the long sync pulse ends. 3) For military sync, the beginning of the first line containing active picture.	Negative

Table 7-2 (continued) Range Expressions

Expression Syntax	**Description**	**Start Time**	**End Time**	**Polarity**
`Vertical Back Porch`	The third of three portions of the vertical blanking interval. For formats that employ military sync (see definition in the `Vertical Sync` entry of this table on page 79), vertical back porch has duration equal zero.	The end of vertical sync (see definition in the `Vertical Sync` entry of this table on page 79).	These choices are possible: 1) If the format contains intra-line pulses (see vertical front porch), the time of the first active video. 2) If the format does not contain intra-line pulses, the beginning of the first line containing active picture.	Positive
`Field Active`	The active portion of this field. It is the time in the field exclusive of the vertical blanking interval. Each field contains exactly one Field Active portion.	The end of vertical back porch (see definition in the `Vertical Back Porch` entry of this table on page 80).	The beginning of vertical front porch (see definition in the `Vertical Back Porch` entry of this table on page 80).	Positive
`Horizontal Front Porch`	The first of three portions of the horizontal blanking interval containing the dead time preceding horizontal sync. Each line contains exactly one Horizontal Front Porch.	The end of active video in the previous line.	The beginning of the horizontal sync pulse.	Positive
`Horizontal Sync`	The second of three portions of the horizontal blanking interval defined as the duration of time while the horizontal sync pulse is low. The time marks the point of the 50% crossing of the pulse. Each line contains exactly one Horizontal Sync period.	The falling edge of the sync pulse (50% crossing).	The rising edge of the sync pulse (50% crossing).	Negative

Table 7-2 (continued) Range Expressions

Expression Syntax	Description	Start Time	End Time	Polarity
`Horizontal Back Porch`	The last of three portions of the horizontal blanking interval, containing the dead time following horizontal sync. In composite formats, this is used for color burst. Each line contains exactly one Horizontal Back Porch period.	The end of the horizontal sync pulse.	The beginning of active video.	Positive
`Line Active`	The active pixels in the line.	The time of the first active element.	The time of the last active element.	Positive
edge-transition-direction `edge of` *board-signal-name*	The time of the specified edge. This directive allows you to specify a transition that has a direction of 1) `rising` (low to high); 2) `falling` (high to low); or `any` (matches both rising and falling). The transition refers to a change in the sense of a board signal. The transition must be defined prior to the execution of the range expression in which it is referenced; transitions are defined in: 1) the "set signal" statement (see "The set signal Statement for Transitions" on page 71); or 2) in one of the special user signals, defined when the user employs the compiled language (see "The User Signals for Reference" on page 83). The polarity corresponds to the sense of the transition. If the transition is a falling edge, the polarity is `low`; if the transition is rising, the polarity is `high`.	The time of the transition.	Same as start time.	Can be either. See the discussion in the Description column

Table 7-2 (continued) Range Expressions

Expression Syntax	Description	Start Time	End Time	Polarity
pulse-transition-direction `pulse of` *board-signal-name*	The time of the specified pulse. This directive allows you to specify a transition that has a direction of 1) `positive` (signal in `high` state); 2) `negative` (signal in `low` state). For a discussion of how transitions are defined, see definition in the "edge-transition-direction `edge of` board-signal-name" entry of this table on page 81. The polarity corresponds to the sense of the pulse. If the pulse is positive-going, the polarity is positive; if the pulse is negative-going, the polarity is negative.	The time of the transition.	Same as start time	Can be either. See the discussion in the Description column
time-expression	The specified duration. This allows you to specify an absolute duration as measured from the beginning of the frame. If you specify an iterative time value that does not comprise exact whole-number units of the frame, the last time expression will be truncated to the time at the end of the frame. Put more simply, you will never get an end time past the end of the frame, even if the whole unit of time would properly set it past the end of the frame.	Consider *i* to be the number of iterations of the time expressions of duration *d*. The start time is $(i-1)\times d$	$i\times d$	Can be either. See the discussion in the Description column

Default Range

The default range is the name given to the range before any range selections are applied and is set to the entire frame. This is equivalent to setting the following range:

```
within each (frame) ...
```

Although not harmful, it is not necessary to use this construct.

Ranges That Do Not Fall on Whole-Unit Boundaries

Nested ranges always find the beginning and end of the whole unit for which they are defined. Examine this example, where we assume vertical front porch contains 10.5 lines and starts in the middle of a defined line of video. Watch what happens when the following statements are used:

```
within each (Vertical Front Porch)
    within each (line)
        ...
```

In the above example, the statement `within each (Vertical Front Porch)` would be repeated exactly 10 times, corresponding to the number of whole units (in this case, lines) contained within vertical front porch. Because vertical front porch begins on a half-line boundary, the first whole unit (line) begins half a line past the beginning of vertical front porch.

In other words, nested ranges always round forward in time to the next whole unit.

The User Signals for Reference

The edge database contains the set of all transitions of signals from which the specific chip hardware is loaded. In addition to the signals defined in the board and chip definitions, the compiler defines several standard signals internally. These signals are defined either automatically when the user employs the high-level language, or manually if the user cites them using the line-based language.

These signals can be referenced from within the hardware rules and should be used in preference to the signals that they alias; these signals do not automatically generate any other signal. For example (and most notably), you must produce the actual sync output from the board. The actual sync signal must be copied from the “User Sync” signal; this can be done trivially as in Example 7-4.

Example 7-3 Copying User Sync Signal to a Board Signal

```
/*
 * Copy the user sync signal to the board signal CSYNC
 */
within each (edge of user sync) {
    set signal “CSYNC” CorrespondingPolarity at BeginTime;
}
```

These signals represent a timing ideal that may differ from the actual representation on the hardware. For example, the user active signal is the time at which the active signal makes transitions during a line of video; however, the hardware may need to advance or delay the actual active signal output from the chip. By specifying the ideal time at which the transition should be made in the user signal, the rules may derive the chip time from the ideal.

User Sync Signal

The user sync signal is high when the signal is at blanking level, low when at sync level. Within the rules or the line-based language, you may refer to `user sync` (as in Example 7-4).

User Blank Signal

The user blank signal is high when the signal is enabled (that is, the signal is high when blanking is enabled, low when blanking is disabled and pixels are displayed). Within the rules or the line-based language, you may refer to `user blank`.

The user blank signal aligns itself on whole lines, presuming a normal frame layout. This works correctly with the half lines of commercial sync and with the whole lines of block sync and military sync. If the source video format does not have its active area aligned with whole lines, you should not use the user blank signal; instead, reference the user sync signal directly (see “User Sync Signal” on page 84).

User Active Signal

The user active signal is high when the signal is enabled (that is, high when pixels are displayed). The distinction between user blank and user active is on those lines when active takes only one-half the line (for example, first lines of one field of NTSC and PAL): on those half lines, the duration of user active is the full line, while the duration of user blank is half-line. With this difference between the signals, those hardware platforms that must always fetch an entire line when requesting pixels may use both signals to create the distinction; those that can request partial lines may do so by referencing only the blank signal. Within the rules or the line-based language, you may refer to `user active`.

System-Defined Variables

The compiler defines several variables for your use. These variables are described in several places:

- "How You Can Use Range Information" on page 75.
- Those variables in Table 5-2 on page 42.
- "Field Attribute Variables" on page 85.
- "Format Information Variables" on page 86.

Field Attribute Variables

These variables get their values from the attributes the user specifies for each field. The attributes and their definitions are in Table 5-3.

Each variable is an array with each element corresponding to a field. The array is zero-based, so the first field is in array element `[0]`, the second in element `[1]`, and so on. How many fields in the format? You can use the variable *FieldsPerFrame* (see Table 5-2) to find out.

Table 7-3 Field Attribute Variables

Name[a]	Data Type	Description
FieldLineCount	int	The number of lines in the field.
FieldLineOffset	int	The offset specified by the user.
FieldLineSkip	int	The skip specified by the user.
FieldSwap	boolean	Is `true` if swap enabled this field, `false` if not.
FieldColor	mask	Is 1 for red, 2 for green, 4 for blue (multiple values can be specified with inclusive-OR).
FieldEye	mask	Is 1 for left, 2 for right (multiple values can be specified with inclusive-OR).

a. These names are shown without the array index. All have FieldsPerFrame elements.

Format Information Variables

These are useful only after frame analysis takes place, so you can use them in either rules definitions or post-processing operations.

Table 7-4 Format Information Variable

Name	Data Type	Description
`SwapsPerFrame`	int	The number of swaps per frame.

Built-In Functions

A limited number of functions are available to aid with writing rules.

Table 7-5 Functions

Function Name	Returned Data Type	Description
TransitionsDefinedOnSignal (*board-signal-name*)	Boolean	Returns `true` if the specified signal has any transitions already defined on it, `false` if no transitions were yet defined. Transitions may already be defined on the signal if the signals were user-defined signals (see "The User Signals for Reference" on page 83); or were defined with line-based language (see Chapter 8, "Line-Based Format Definition Language"). For a discussion of the use of this function, see "Anticipating Line-Based Definitions in the Rules" on page 98.
sec (*time-expression*)	float	Returns a floating-point value equal to the number of seconds in *time-expression*.

Examples of Writing Rules

Copying User Sync

It is often necessary to create a signal with the same shape as the sync signal provided by the user. Because the compiler provides the user sync signal as a predefined set of transitions in the database, you can easily copy the signal. You can read about the user sync signal in "User Sync Signal" on page 84.

Example 7-4 Copying the User Sync Signal

```
within each (edge of user sync) {
    set signal "CSYNC" CorrespondingPolarity at BeginTime;
}
```

To copy user sync to a signal named *CSYNC*, this example uses the `each edge` range expression; for information, see the entry "edge-transition-direction `edge of` board-signal-name" in Table 7-2. For each edge of the user sync signal in the database, the range expression causes `within` to execute the `set signal` statement. The `CorrespondingPolarity` value copies the polarity of the user sync signal, while `BeginTime` sets the transition to the same time as the user sync signal (see "How You Can Use Range Information" on page 75 for information on these variables).

Copying Another Signal With Changes

It is unusual to use another true signal with no changes at all (why would a hardware designer require two identical signals?). More often, some changes are necessary to make a signal operate properly.

Example 7-5 Copying Another Signal With Changes

```
within each (rising edge of "IMPREG_LD") {
    set signal "XMAPREG_LD" high at BeginTime;
}
within each (falling edge of "IMPREG_LD") {
    set signal "XMAPREG_LD" low at BeginTime - 0.1 usec;
}
within each (positive pulse of "IMPREG_LD") {
    time pulseDuration;

    pulseDuration = (EndTime - BeginTime);

    if (pulseDuration < 1.0 usec)
        error "Insufficient IMPREG_LD pulse == " + pulseDuration;
}
```

The `each rising edge` range expression is similar to the copy operation shown in "Copying User Sync" on page 87, yet this expression discriminates by copying only half the edges (only the rising edges). For information, see the entry "edge-transition-direction `edge of` board-signal-name" in Table 7-2. The time and polarity of *XMAPREG_LD* transitions match those of *IMPREG_LD*.

However, note the `each falling edge` range expression's `set signal` statement. The low transition of *XMAPREG_LD* is advanced by 0.1 usec; the new transition precedes the time of *IMPREG_LD* by that amount of time.

The last range expression, `each positive pulse`, performs error checking and has no `set signal` statement associated with it; see the entry "pulse-transition-direction `pulse of` board-signal-name" in Table 7-2. This example shows checking the duration of a pulse (delimited by `BeginTime` and `EndTime`). If the pulse is too short, the compiler will issue an error message generated by the `error` statement; see "Diagnostic Statements" on page 66.

Calculating Scalar Variables

The example in "Copying Another Signal With Changes" on page 88 shows calculation of a variable for detecting an error state. It is possible to calculate exported variables for later use as well (see "Variables" on page 65).

Example 7-6 Calculating Scalar Variables

```
{
    exported double HBackPorchClampStart;
    exported double HBackPorchClampLength;

    within (field active 1,1) {
        within (line 2,2) {
            time LineStartTime;

            LineStartTime = BeginTime;

            within each (Horizontal Back Porch) {
                time tBackPorchStart;
                time tBackPorchLength;

                tBackPorchStart = BeginTime - LineStartTime + 2 clocks;

                HBackPorchClampLength = sec(HorizontalBackPorch)/2;
                HBackPorchClampStart =
                    sec(tBackPorchStart) + (HBackPorchClampLength/2);
            }
        }
    }
}
```

The code in Example 7-6 shows the definition and calculation of two variables: *HBackPorchClampStart* and *HBackPorchClampLength*. You can see the declaration of the variables is the first item within a compound statement delimited by curly brackets (`{ }`). Syntax requires you to declare items only at the beginning of a block.

Example 7-6 shows use of nested `within` statements. The innermost loop is defined as the horizontal back porch area of second line of the first field's active region of this format.

The exported length variable is calculated by determining the duration (in seconds) of the back porch using the **sec** function. The exported start variable also uses the *tBackPorchStart* variable, the calculated time position of the relative start location of horizontal back porch within a line.

Control Statements

You can use `if`, `for` and `while` statements to alter control flow to execute different sections of code. You can see their use in Example 7-7.

Example 7-7 Using Control Statements in Rules

```
{
    int nEdgeCnt = 0;

    /* check for serrations */
    within (Vertical Sync 1, 1) {
        within each (edge of user sync) {
            nEdgeCnt++;
        }
    }

    if (nEdgeCnt > 2) {
        /* we have serrations, so don't create transitions */
        signal "PLL_PHASE" initial state = high;
    } else {
        /* mask out vertical from the PLL's phase detector */
        signal "PLL_PHASE" initial state = low;
        within each (Vertical Sync) {
            set signal "PLL_PHASE" low  at BeginTime - 2.5H
                ignoring wrap error;
            set signal "PLL_PHASE" high at EndTime + 2.5H
                ignoring wrap error;
        }
    }
}
```

The code in Example 7-7 performs different operations based on whether the user's sync signal contained serration pulses (see "The Vertical Interval" on page 16).

To determine whether the format has serrations, this code fragment counts the number of transitions (using the `each edge` range expression) in the first vertical sync region. Note that the variable *nEdgeCnt* is defined at the beginning of a compound statement because syntax requires variables be declared there. If this variable were not needed, the extra curly brackets (`{ }`) at the beginning and end of the example would probably not be needed.

If the format does have serrations, no signal transitions are defined—only the initial state is set. However, if the sync signal does have serration pulses, the two `set signal` statements are executed every vertical sync. The difference of 2.5 lines (`2.5H`) from the beginning and end of the vertical sync region puts the transitions beyond the beginning and end of the frame; therefore, the ignoring wrap error statement is needed to keep the compiler from issuing a warning for this circumstance. See "The set signal Statement for Transitions" on page 71.

Pre-Processing and Post-Processing Statements

These statements allow you to execute private functions needed to complete your format. You can define pre- and post-processing statements in any file. These statements are executed before and after the rules processing is performed, respectively.

The syntax of the statement is

`preprocess {` *statement*`... }`

`postprocess {` *statement* `... }`

You may perform any processing you need within these statements. Moreover, you can have as many pre- or post-processing statements as needed; they are executed in the order the compiler encounters them.

Communication Between Format Source and Rules Files

In some circumstances, you may find it necessary to communicate some information between the video format source file and the rules file. For example, you may wish to direct the rules file to perform some special processing based on instructions in the source file.

As with most programming, you can find many ways to accomplish the same task using vfc. However, exported variables (see “Storage Classes” on page 38) are designed to make this work easy. The lifetime of these variables is unusual: regardless of where exported variables are defined in any of the source files, their lifetime commences before any code in any source file is executed. Thus, it is possible to define the variable in the rules file and assign the variable in a video format source file (even though the code in rules files is executed after the source).

As an example, imagine a display board which produces separate horizontal and vertical sync signals in addition to the normal sync embedded in one of the (usually green) data signals. In some circumstances, it may be useful to misappropriate the horizontal signal to produce a composite output. For example, this trick is especially useful when trying to genlock another SGI system to a format with an ambiguous sync signature.

In this circumstance, the author of the rules file needs to allow the author of the video format source to transmit that information. The means to do this most easily is with an exported variable. In our example, the variable is *SerratedCSyncOnHSync*.

Example 7-8 Video Format Source Setting SerratedCSyncOnHSync

```
General
{
    FieldsPerFrame = 2;
    FramesPerSecond = 25;
    TotalLinesPerFrame = 1065 * 2;
    TotalPixelsPerLine = 1680;
    ActiveLinesPerFrame = 1024;
    ActivePixelsPerLine = 1280;
    FormatName = “1280x1024_25r2 (PAL frame-reset)”;
    SerratedCSyncOnHSync = true;
}
...
```

In Example 7-8, the video format source is declared much the same as any other. The addition to the normal variables set in this section is the assignment of *SerratedCSyncOnHSync* to true.

Example 7-9 Rules File Source Defining and Using SerratedCSyncOnHSync

```
define board_hardware
{
    /* set this to generate a serrated CSync signal */
    /* on HSync for block sync formats */
    exported int SerratedCSyncOnHSync = 0;
```

```
...
    if (SerratedCSyncOnHSync) {
        info “Creating serrated composite for field-matched genlock”;
        PFDSourceSignalName = “HSYNC”;

        within each (field) {
            Time b;
            Time e;
            Time m;
            Time EndOfField;

            EndOfField = EndTime;

            within (Vertical Sync 1, 1) {
                b = BeginTime;
                e = EndTime;
            }

            within each (line) {
                if (BeginTime < EndOfField) {
                    m = (BeginTime + EndTime) / 2;
                    /* are we a line during vertical sync? */
                    if ((m >= b) && (m < e)) {
                        set signal “HSYNC” low at BeginTime;
                        set signal “HSYNC” high at EndTime - lenHSync;
                    }
                    else /* standard line */
                    {
                        set signal “HSYNC” low at BeginTime;
                        set signal “HSYNC” high at BeginTime+lenHSync;
                    }
                }
            }
        }
    }
    else /* standard hsync pulse train */
    {
        PFDSourceSignalName = “CSYNC”;

        within each (line) {
            set signal “HSYNC” low at BeginTime;
            set signal “HSYNC” high at BeginTime + lenHSync;
        }
    }
```

In Example 7-9, you can see the declaration of the variable *SerratedCSyncOnHSync* as the first line in the rules file section. This declaration is the only thing you need to create the exported variable for the lifetime of any source code. Later in the rules section, the variable is used to determine behavior: if *SerratedCSyncOnHSync* is set, the rules file simulates a composite sync signature for the format by supplying a different-looking pattern during vertical sync than for lines outside of vertical sync.

Chapter 8

Line-Based Format Definition Language

From time to time, you may come across a video format for which rules are difficult to write. Either the format is too complicated or the rules just do not quite apply. Rather than contort rules files unnecessarily, there is an escape mechanism.

Writing a video format in line-based language allows you to express some or all of the generated signals for each line.

Note: Normally, writing line-based definitions is reserved only for Silicon Graphics engineering personnel who can anticipate problems in the relationship to other signals this might cause in rules generation. If you do not understand those relationships, stop now!

The Line-Based Language

To use the line-based language, you must put the directives in a special section, declared as follows (similarly to that of the Field declaration):

```
Line Based {
    statement...
}
```

The line-based language has but two statement types, one to set initial state, the other to specify a transition:

`Signal` *signal-name* `initial state =` *direction*`;`

`Transition Line Range` *signal-name* `=` *start-line* `to` *end-line direction* `at` *time-expression*`;`

The line-based language is very simple. For each transition, you specify a line number and time on that line. The components are as follows:

- *signal-name* is the signal being addressed. This is often the synchronization signal produced by the hardware. You may also use one of the user signals as the signal name (see “The User Signals for Reference” on page 83).
- *direction* is high or low, as described in “The set signal Statement for Transitions” on page 71.
- Both *start-line* and *end-line* are integer line numbers on which the transition is to occur. If you wish a transition to occur on only one line, specify both values to be the same line number.
- For *time-expression*, you can use any of the time expressions specified in “Time Expressions” on page 35.

The line-based initial state has the same function as the same statement in rules generation, described in “Initial State” on page 71. The transition line range statement is similar to that described in “The set signal Statement for Transitions” on page 71.

Examples

The following example sets the signal *HSYNC* signal. The first statement sets the signal to the low state on each line (note the range "`1 to TotalLinesPerFrame`"); the second statement sets the signal to high at a different position on each line. The variable *GlobalDelay* is private to this format and is set elsewhere in the source file.

Example 8-1 Setting a Signal on Every Line

```
Line Based {
    signal "HSYNC" initial state = low;

    Transition Lines Range "HSYNC" = 1 to TotalLinesPerFrame
        low at (GlobalDelay - HorizontalFrontPorch);

    Transition Lines Range "HSYNC" = 1 to TotalLinesPerFrame
        high at (HorizontalSync + HorizontalBackPorch +
                 GlobalDelay - 1.0H);

}
```

The following example, also using the *GlobalDelay* variable, sets the *VSYNC* signal to high on line 23 and low on line 311 (setting the low and high line numbers of the range to the same value executes the statement on only that single line).

Example 8-2 Use of Single Line Ranges

```
Line Based {
    signal "VSYNC" initial state = low;

    Transition Lines Range "VSYNC" = 23 to 23
        high at (GlobalDelay - HorizontalFrontPorch );

    Transition Lines Range "VSYNC" = 311 to 311
        low at (GlobalDelay - HorizontalFrontPorch );
}
```

Anticipating Line-Based Definitions in the Rules

Did you read “The Edge Database” on page 71? Most of the activity in writing a format and rules is the act of adding transitions to the edge database. Without taking some care, you can add too many transitions if you write in line-based language.

How can you add too many transitions? If you specify a transition in line-based language and then specify similar transitions in rules (see “The set signal Statement for Transitions”), you will find both transitions in the edge database.

The solution? Use the function **TransitionsDefinedOnSignal**, described in Table 7-5. This function allows you to detect whether the edge database already contains any transitions on a specified signal.

Example 8-3 Use of Function TransitionsDefineOnSignal

```
/*
 * COMP_SYNC
 */
if (!TransitionsDefinedOnSignal("COMP_SYNC")) {

    /*
     * Should follow user sync exactly
     */
    signal "COMP_SYNC" initial state = high;
    within each (edge of user sync) {
        set signal "COMP_SYNC" CorrespondingPolarity at BeginTime;
    }
}
```

Example 8-3 shows proper use of the **TransitionsDefinedOnSignal** function. You can see that no additional transitions are added using `set signal` if line-based language has already been used to set the transitions.

Appendix A

Native Language Grammar

This format uses the notation that has become more or less standard: non-terminal tokens followed by a colon. Indented items following the token are valid solutions. An ellipsis (...) following any item means it can be repeated. Optional items have the suffix $_{opt}$ appended to them. One departure from the standard: except for the first token, the non-terminals are listed alphabetically.

The grammar starts with the non-terminal token "program".

program:

    program-line...

active-line-section:

    compound-statement

compound-statement:

    {compound-statement-declaration... statement...}

compound-statement-declaration:

    declaration-storage-class$_{opt}$ *declaration-data-type declaration-init , ... ;*

constant:

    constant-double

    constant-int

    constant-string

    constant-time

constant-double:

    double-value

constant-int:

`0x` *int-value*

`true`

`false`

`high`

`low`

constant-string:

" quoted-string"

constant-time:

constant-double `pixels`

constant-double `sec`

constant-double `msec`

constant-double `usec`

constant-double `lines`

constant-double `H`

constant-double `clocks`

declaration-data-type:

`double`

`int`

`string`

`time`

declaration-init:

variable-name variable-initialization$_{opt}$

declaration-storage-class:

`exported`

expression:

constant

variable

(expression)

! expression

~ expression

expression++

*expression * expression*

expression / expression

expression % expression

expression + expression

expression - expression

expression << expression

expression >> expression

expression < expression

expression <= expression

expression > expression

expression >= expression

expression == expression

expression != expression

expression `||` *expression*

expression && expression

expression `|` *expression*

expression & expression

expression = expression

eye-name:

`right`

`left`

field-definition:

`field` *compound-statement*

frame-portion-definition:

frame-portion-identifier = { sync-transition-set... }

frame-portion-identifier:

`vertical back porch`

`vertical front porch`

`vertical sync`

`active`

general-section:

  `general` *compound-statement*

primary-color:

  `red`

  `green`

  `blue`

program-line:

  general-section

  active-line-section

  field-definition

statement:

  expression

  `color` = *{ primary-color , ... }*

  `eye` = *{ eye-name , ... }*

  frame-portion-definition

  `if` *(expression) statement*

  `if` *(expression)* `else` *statement*

sync-transition-instruction:

  `length` = *time-expression ;*

  sync-transition-direction = *time-expression ;*

sync-transition-multiplier:

  `repeat` *expression*

sync-transition-set:

  sync-transition-multiplier$_{opt}$ *{ sync-transition-instruction... }*

time-expression:

  expression

variable:

  identifier

variable-initialization:

  = *expression*

Index

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-3402-003.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389