Chapter 5

User Input

This chapter tells you how to program your application to respond to input from a user. User input can occur thorough a variety of devices, including the keyboard, mouse and peripheral devices.

- Section 5.1, “Event Handling,” describes event handling mechanisms and discusses input models.
- Section 5.2, “Input Devices,” describes the types of input devices available and lists the values they report.
- Section 5.3, “Video Control,” tells you how to query and control video parameters.

5.1 Event Handling

In the previous chapters, the emphasis has been on drawing graphics. You probably want to do more than just watch graphics on the screen; the next step is to add user input capability. A *user interface* provides a mechanism for detecting user input, acting on the input received, and determining when user input has ceased.

An *input event* is generated when you click a mouse button or press a key on the keyboard. Input events and the system responses associated with them are the dialogue used to communicate with the computer through the user interface. *Event handling* is the process that manages this communication: including rules that govern what constitutes an input event, how event priorities are established, and how events are communicated back and forth between the user and the computer.

User input events on the IRIS are handled by the *X server* (see the X Window System documentation). You can set up user input using X facilities, using the GL, or using a combination of X and GL known as *mixed-mode*, or *mixed-model*, programming.

Working with X gives you the advantage of using X-based *toolkits* and *widgets* to create your user interfaces. The X Window System provides facilities for event management through the Xt routines for timed events and X event routines for queued events. The X input model provides more information than the GL for each event and allows for multiple server, screen and window communication. If you want to use Motif™, Athena widgets, Xt, or multiple server I/O, you *must* use the X input model. See the X Window System documentation for more information about structuring X input environments.

For mixed-model programming, see the `glresources()` man page, and the `GLX` group of man pages, all of which begin with the upper-case letters GLX.

A GL program can detect input events in two ways: *queueing* or *polling*. Queueing saves input events and places them in an event *queue*, a section of memory where the events are stored in the order received; the first event in the queue is the first event to get processed. It is just like waiting in line to buy movie tickets—the first person in line gets the first tickets.

Polling continually *samples* (checks) the device(s) for input. Polling is not very efficient because the system spends a lot of time listening for input that could be used for doing other tasks. It is possible for the system to miss an input event that happens when it isn't listening.

Queueing is the recommended way to manage user input. Queueing allows you to process rapid up-and-down button transitions and capture all momentary device state changes.

Another difference between queuing and polling is the effect that the window system has on them. When you queue buttons and valuators, an event is generated only for the process controlling the window that currently has *input focus*.

Input is enabled for the window that has the cursor in it; that window currently has the input focus. In a multiple window system input occurs in the window that has input focus; it does not occur in any other windows that are open concurrently.

5.1.1 Queueing

You decide which devices to queue, and establish rules about what constitutes a state change, or *event*, for those devices.

Any change in the state of a device for which queueing is enabled generates an entry in the event queue. Each queue entry consists of the device number and the current value of the device. If the input device is a button, the value is either 1 (pressed) or 0 (released). If the device is a *valuator*, such as the *x* position of the mouse, its value is an integer that indicates the position of the device.

To use queueing:

1. Enable queueing for the selected device with `qdevice()`.
2. Check the queue for an event, using `qread()`, `qtest()` or `blkqread()`.
3. Perform the appropriate action for that event.
4. Return to the event loop
5. Disable event queueing when the event loop is exited.

For maximum efficiency, you should not put complicated redraw operations inside the event loop and you should provide some mechanism for emptying the queue when it gets full.

The input queue can contain up to 101 events at a time. To check for overflow, you can queue the device `QFULL`. This inserts a `QFULL` event in the graphics input queue of a GL program at the point where queue overflow occurred. This event is returned by `qread()` at the point in the input queue at which data was lost. Use `qreset()` to empty the queue when it gets full.

By default, only the pseudo-devices `INPUTCHANGE` and `REDRAW` are queued.

An `INPUTCHANGE` event appears when you direct the input focus to a new window or to the background. When an `INPUTCHANGE` event occurs, the identifier of the window that has input focus is placed on the queue; a 0 is placed on the queue when input focus is removed.

A `REDRAW` event appears in the queue when you move or reshape a window, when part of a window is uncovered or when the display mode changes.

GL subroutines for processing user input are listed next, with a brief description of what the command does, followed by its ANSI C specification.

qdevice

`qdevice()` enables queueing for the specified device (*dev*) (for example, a keyboard, button, or valuator). The argument of `qdevice()` is a device number. Each time the device changes state, an entry is made in the event queue.

```
void qdevice(Device dev);
```

unqdevice

`unqdevice()` disables the queueing of events from the specified device. If the device has recorded events in the queue that have not been read, those events remain in the queue. (You can use `qreset()` to flush the event queue.)

```
void unqdevice(Device dev);
```

isqueued

`isqueued()` finds out whether or not a specific device is queued. `isqueued()` returns a Boolean value. TRUE indicates that the device is enabled for queuing; FALSE indicates that the device is not queued.

```
boolean isqueued(short dev);
```

qenter

`qenter()` creates event queue entries. It places entries directly into the program's own event queue. `qenter()` takes two 16-bit integers, *qtype* and *val*, and enters them into the event queue.

```
void qenter(short qtype, short val);
```

You can use any one of the next three subroutines—`qtest()`, `qread()`, `blkqread()`—to check the event queue for an entry.

qtest

`qtest()` returns the device number of the first entry in the event queue; if the queue is empty, it returns zero. `qtest()` always returns immediately to the caller and makes no changes to the queue.

```
long qtest();
```

qread

`qread()`, like `qtest()`, returns the device number of the first entry in the event queue. However, if the queue is empty, it waits until an event is added to the queue. `qread()` returns the device number, writes the data part of the entry into the short pointed to by data, and removes the entry from the queue.

```
long qread(short *data);
```

blkqread

`blkqread()` returns multiple queue entries. Its first argument, data, is an array of short integers, and its second argument, *n*, is the size of the array data. `blkqread()` returns the number of shorts returned in the array data, which is filled alternately with device numbers and device values. Note that the number of entries read is twice the number of queue entries, hence it can be at most *n* / 2.

You can also use `blkqread()` when only the last entry in the event queue is of interest (for example, when a user-defined cursor is being dragged across the screen and only its final position is of interest).

```
long blkqread(short *data, n);
```

qreset

`qreset()` removes all entries from the queue and discards them.

```
void qreset(void);
```

qgetfd

`qgetfd()` allows a GL program to use the IRIX system call *select* to determine when there are events waiting to be read in the graphics input queue. A call to

`qgetfd()` returns a file descriptor that may be used as part of the *readfds* parameter of the *select* system call. When *select* indicates that the file descriptor associated with the graphics input queue is ready for reading, a call to `qread()` or `blkqread()` does not cause the program to block.

```
long qgetfd(void);
```

tie

You can `tie()` a queued button to one or two valuators so that whenever the button changes state, the system records the button change and the current valuator position in the event queue. `tie()` takes three arguments: a button *b* and two valuators *v1* and *v2*. You `tie()` one valuator to a button by making *v2* equal to 0. Whenever the button changes state, three entries are made in the queue that record the current state of the button and the current position of each valuator. You can untie a button from valuators by making both *v1* and *v2* equal to 0.

```
void tie(Device b, Device v1, Device v2);
```

attachcursor

`attachcursor()` attaches the cursor to the movement of two valuators. Both of its arguments, *vx* and *vy*, are valuator device numbers that correspond to the device that controls the horizontal and vertical location of the cursor, respectively. By default, *vx* is MOUSEX and *vy* is MOUSEY. The valuators at *vx* and *vy* determine the cursor position in screen coordinates. Every time the values at *vx* or *vy* change, the system redraws the cursor at the new coordinates.

```
void attachcursor(Device vx, Device vy);
```

To control cursor position from within a program, attach the cursor to `GHOSTX` and `GHOSTY`. The program can then use `setvaluator()` on `GHOSTX` and `GHOSTY` to move the cursor. `attachcursor()`, like `blink()`, is not reset to the default when the program exits.

curson and cursoff

`curson()` and `cursoff()` determine the visibility of the cursor in the current window. Use them to control the state of the cursor while drawing in the selected window.

`curson()` makes the cursor visible in the current window; `cursoff()` makes it invisible.

```
void curson();
void cursoff();
```

By default, the cursor is on when a window is created.Use `getcursor()` to find out whether the cursor is visible. See Chapter 11, "Frame Buffers and Drawing Modes," for more information on `getcursor()`.

noise

Some valuators are noisy; that is, they report small fluctuations, indicating movement when no event has occurred. `noise()` allows you to set a lower limit on what constitutes an event. The value of a noisy valuator *v* must change by at least *delta* before the motion is recognized. `noise()` determines how queued valuators make entries in the event queue. For example, `noise(v,5)` means that valuator *v* must move at least five units before a new queue entry is made.

```
void noise(Device v, short delta);
```

5.1.2 Polling

Polling immediately returns the value of a device that is a button or valuator, regardless of which window has focus. For example, the statement `getbutton(LEFTMOUSE)` returns 1 if the left button of the mouse is down and 0 if it is up. Programs that use polling should watch for changes to input focus and adjust their behavior accordingly.

getvaluator

`getvaluator()` polls the status of a valuator. The argument to `getvaluator()` is a valuator device number (*val)* that reflects the current state of the device.

```
long getvaluator(Device val);
```

getbutton

`getbutton()` polls the status of a button, whether the button is queued or not. The argument to `getbutton()` is the number of the device you want to poll (*num*). `getbutton()` returns TRUE if the button is down, FALSE if it is up.

```
Boolean getbutton(Device num);
```

getdev

`getdev()` polls up to 128 valuators and buttons concurrently. Specify the number of devices you want to poll (*n*) and an array of device numbers (*devs*). (See Tables 5-1, 5-2, and 5-3 for listings of device numbers.) The *vals* array returns the state of each device in its corresponding array location.

```
void getdev(long n, Device devs[], short vals[]);
```

The following sample program, *input.c*, uses queueing to control a simple drawing program, that lets you sketch in the window with the left mouse button.

```
#include <gl/gl.h>
#include <gl/device.h>

#define X    0
#define Y    1

main()
{
   short val, mval[2], lastval[2];
   long org[2], size[2];
   Device dev, mdev[2];
   Boolean run;
   int leftmouse_down = 0;
   lastval[X] = -1;
   prefsize(400, 400);
   winopen("input");
   color(BLACK);
   clear();
   getorigin(&org[X], &org[Y]);
   getsize(&size[X], &size[Y]);
   mdev[X] = MOUSEX;
   mdev[Y] = MOUSEY;
   getdev(2, mdev, lastval);     /* initialize lastval[] */
   lastval[X] -= org[X];
   lastval[Y] -= org[Y];
```

```
    qdevice(LEFTMOUSE);
    qdevice(ESCKEY);
    qdevice(MOUSEX);
    qdevice(MOUSEY);
    color(WHITE);          /* prepare to draw white lines */

  while (1) {
    switch (dev = qread(&val)) {
        case LEFTMOUSE:
            leftmouse_down = val;
            break;
        case MOUSEX:
            val[X] = val - org[X];
            break;
        case MOUSEY:
            mval[Y] = val - org[Y];
            if (leftmouse_down) {
                bgnline();
                    v2s(lastval);
                    v2s(mval);
                endline();
            }
            lastval[X] = mval[X];
            lastval[Y] = mval[Y];
            break;
        case ESCKEY:
            exit(0);
    }
  }
}
```

5.2 Input Devices

Input devices accept user input and translate that input into data that a program can use. The GL supports three classes of input devices:

Buttons — Return a Boolean value: FALSE when they are not pressed (open) and TRUE when they are pressed (closed).

Valuators — Return an integer value that represents their current status. For example, a mouse is a pair of valuators: one reports horizontal position and the other reports vertical position.

Pseudo-devices — Return information about other system events. For example, the keyboard returns ASCII characters. Most of these pseudo-devices register events. The keyboard device reports character values when keys (or combinations of keys) are pressed. If you press the **a** key, an ASCII **a** is reported; if you press the **<Shift>** key, nothing is reported, but if you hold down the **<Shift>** key and then press the **a** key, an ASCII **A** is reported.

Devices are named by a unique integer within the domain 1 to 32767, inclusive. Table 5-1 shows how the device domain is organized.

Type	Range	Device Class
	0x001 — 0x0FF	Buttons
	0x100 — 0x1FF	Valuators
Reserved	0x200 — 0x2FF	Pseudo-devices
Devices	0x300 — 0xEFF	Reserved
	0xF00 — 0xFFF	Additional Buttons
	0x1000 — 0x2FFF	Buttons
User-definable	0x3000 — 0x3FFF	Valuators
Devices	0x4000 — 0x7FFF	Pseudo-devices

Table 5-1 Class Ranges in the Device Domain

Facilities exist to let you define your own input devices. Additional information is available in the GL man pages.

5.2.1 Buttons

User input buttons include the mouse buttons, keyboard keys, buttons on a dial and button box, digitizer tablet buttons, and a menu button. Table 5-2 lists the names of the buttons and their descriptions.

Devices	Description
MOUSE1	Right mouse button
MOUSE2	Middle mouse button
MOUSE3	Left mouse button
RIGHTMOUSE	Right mouse button
MIDDLEMOUSE	Middle mouse button
LEFTMOUSE	Left mouse button
SW0...SW31	32 buttons on dial and button box
AKEY...PADENTER	All the keys on the keyboard
BPAD0	Pen stylus or button for digitizer tablet
BPAD1	Button for digitizer tablet
BPAD2	Button for digitizer tablet
BPAD3	Button for digitizer tablet
MENUBUTTON	Menu button

Table 5-2 Input Buttons

5.2.2 Valuators

Valuators are single-value input devices. Valuators report a 16-bit integer value, such as the horizontal and vertical position of the mouse, or the current setting of a dial. Table 5-3 shows the valuator names and descriptions.

Devices	Description
MOUSEX	x valuator on mouse
MOUSEY	y valuator on mouse
DIAL0...DIAL7	Position of dials on dial and button box
BPADX	x valuator on digitizer tablet
BPADY	y valuator on digitizer tablet
CURSORX	x valuator attached to cursor (usually MOUSEX)
CURSORY	y valuator attached to cursor (usually MOUSEY)
GHOSTX	x ghost valuator
GHOSTY	y ghost valuator
TIMER0...TIMER3	Timer devices

Table 5-3 Input Valuators

The following devices are valuators that return specific information about the system.

Timer Devices

The GL timer devices are used to get input events at regular intervals. The timers use an internal clock that approximates the screen refresh interval. The clock rate is approximately 67 Hz. The event “time” returned by the timers is the frame count recorded during the elapsed event. You should not use GL timers to measure chronological time, nor should you use them to synchronize your graphics programs. To record events less frequently, use `noise()`.

For example, if you call `noise (TIMER0, 30)`, only every 30th event is recorded, generating one event approximately every half second.

Cursor Devices

The cursor devices are pseudo-devices equivalent to the valuators currently attached to the cursor. (See the `attachcursor()` man page for more information.)

Ghost Devices

Ghost devices, `GHOSTX` and `GHOSTY`, do not correspond to physical devices, although they can be used to change a device under program control. For example, to drive the cursor from software, use `attachcursor(GHOSTX,GHOSTY)` to make the cursor position depend on the ghost devices. Then use `setvaluator()` on `GHOSTX` and `GHOSTY` to move the cursor.

5.2.3 Keyboard Devices

The keyboard device returns ASCII values that correspond to the keys typed on the keyboard. The device interprets keyboard movements in the standard manner; for instance, it reports an event only on a downstroke, taking into account the **<Ctrl>** and **<Shift>** keys.

Note: There is a hardware mechanism in the keyboard that reads multiple key-down events if the key is held down and begins to auto-repeat.

Be careful to understand the difference between the device and the values it returns when you queue the keyboard.

If your program contains the instruction: `qdevice(KEYBD)`, the statement `dev = qread(&val)` returns the following:

```
dev = KEYBD
val = the ASCII integer index of the character pressed.
```

To test for individual keystrokes, you can use instructions of the format:

```
qdevice(AKEY);
```

This returns the device AKEY when the **A** key is pressed and the value 1 when the key is pressed; 0 when it is released.

5.2.4 Window Manager Tokens

The tokens listed in Table 5-4 can be queued as pseudodevices to monitor GL window manager events, which are generated when windows are activated or moved. In all cases, the system returns the window identification number (id) of the window experiencing the event.

Token	Description
DEPTHCHANGE	Indicates an open window has been pushed or popped. This token is not supported.
DRAWOVERLAY	Indicates damage to the overlay planes. Queue this token if you use overlay planes.
INPUTCHANGE	Indicates a change in the input focus. If the value is 0, input focus has been removed from the process. If the value is non-0, it indicates the window id of the window that has just gained input focus.
REDRAW	The window manager inserts a REDRAW token each time the window needs to be redrawn. The REDRAW token is queued automatically.
REDRAWICONIC	Queues automatically when iconsize() is called. The window manager sends this token when a window needs to be redrawn as an icon by the program itself.
WINSHUT	When queued, the window manager sends this token when Close is selected from a program's Window (frame) menu, or when the close fixture is selected from the title bar of a program's window. If WINSHUT is not queued, the Close item on the program's Window menu appears grayed out and has no effect if selected.
WINQUIT	When queued, the window manager sends this token rather than killing a process when Quit is selected from a program's Window (frame) menu.
WINFREEZE WINTHAW	If queued, the window manager sends these tokens when windows are stowed to icons and later unstowed, rather than blocking the processes of the stowed windows. These devices should be queued if the program plans to draw its own icon (see iconsize()) or is a multiwindow application.

Table 5-4 Window Manager Event Tokens

5.2.5 Spaceball™ Devices

Table 5-5 lists the devices returned by `qread()` when the optional Spaceball input device sends an event onto the queue.

Devices	Description
SBPERIOD	Number of periods of 0.25 ms since sending the last non-0 set of Spaceball data
SBTX	Right / left push
SBTY	Up / down push
SBTZ	Away / towards push
SBRX	Twist about right / left axis
SBRY	Twist about up / down axis
SBRZ	Twist about away / towards axis
SBBUT1	Button 1
SBBUT2	Button 2
SBBUT3	Button 3
SBBUT4	Button 4
SBBUT5	Button 5
SBBUT6	Button 6
SBBUT7	Button 7
SBBUT8	Button 8
SBPICK	Pick button

Table 5-5 Spaceball Input Buttons

For more information about the optional Spaceball input device, see the documentation that is packaged with the Spaceball option.

5.2.6 Controlling Devices

The GL provides subroutines that initialize device values and control the characteristics and behavior of the system's peripheral input/output devices. For example, some of these routines turn the keyboard click on, `clkon()`, and off, `clkoff()`, or set the keyboard bell. You set these controls to your preference or needs.

setvaluator

`setvaluator()` assigns an initial value (*init*) to a valuator. The arguments *min* and *max* are the minimum and maximum values the device can assume.

```
void setvaluator(Device val, short init, short min, short max);
```

clkon

If `clkon()` is called, the keyboard makes an audible click whenever a key is pressed.

```
void clkon(void);
```

clkoff

`clkoff()` turns off the keyboard click.

```
void clkoff(void);
```

lampon

`lampon()` and `lampoff()` control the four lamps on old-style keyboards labeled L1, L2, L3, and L4 on the keyboard. Each 1 in the four lower-order bits of the *lamps* argument to `lampon()` turns on the corresponding keyboard lamp.

```
void lampon(Byte lamps);
```

lampoff

Each 1 in the four lower-order bits of the *lamps* argument to `lampoff()` turns off the corresponding keyboard lamp.

```
void lampoff(Byte lamps);
```

ringbell

`ringbell()` rings the keyboard bell.

```
void ringbell(void);
```

setbell

`setbell()` sets the duration of the keyboard bell: 0 is off, 1 is a short beep, and 2 is a long beep.

```
void setbell(Byte mode);
```

dbtext

`dbtext()` writes text to the LED display in a dial and button box. The string *str* must be eight or fewer uppercase characters.

```
void dbtext(char str[8]);
```

setdblights

`setdblights()` controls the 32 lighted switches on a dial and switch box. For example, to turn on switches 3 and 7, the third and seventh bits to the right of mask must be set to 1; that is, (1<<3)|(1<<7).

```
void setdblights(unsigned long mask);
```

5.3 Video Control

You can query the status and control the behavior of certain video parameters. You can also determine information about the video monitor used on your system and specify its operating parameters with the following subroutines.

Note: RealityEngine systems use a graphical user interface to set the video format. See the RealityEngine Owner's Guide for information about this interface and the video formats available.

blankscreen

`blankscreen()` turns the screen display on and off. b=TRUE stops display; b=FALSE restarts display.

```
void blankscreen(Boolean bool);
```

blanktime

`blanktime()` sets the screen blanking time-out. By default, the screen blanks (turns black) after the system receives no input for about 15 minutes. This protects the color display. `blanktime()` changes the amount of time the system waits before blanking the screen. It can also disable the screen blanking feature.

```
void blanktime(long nframes);
```

nframes specifies the screen blanking time-out in frame times based on the standard 60 Hz monitor. For software compatibility, the factor of 60 is used, regardless of the monitor type. To calculate the value of *nframes*, multiply the desired blanking latency period (in seconds) by 60. For example, when *nframes* is 1800, the blanking latency period is 5 minutes. There are 60 frames per second; *nframes* is 60 times the number of seconds that the system waits before blanking the screen. When *nframes* is 0, screen blanking is disabled.

setmonitor

`setmonitor()` sets the monitor to one of the video formats listed in Table 5-6.

```
void setmonitor( short type);
```

Not all formats are supported by all systems. If a format is not supported, it is ignored. Some formats may require the use of optional products.

Note: IRIS Indigo systems do not support `setmonitor()`. Elan supports NTSC, PAL, HZ60, HZ72, STR_RECT, and RS-343 monitors.

getmonitor

`getmonitor()` queries the type of the current display monitor, as listed in Table 5-6.

Table 5-6 lists monitor type tokens and the video formats they represent.

Type	Video Format
STR_RECT	120Hz stereo format, if supported by hardware
HZ90_STEREO	90Hz stereo format, if supported by hardware
HZ76	76Hz, noninterlaced
HZ72	72Hz, noninterlaced
HZ60	60Hz noninterlaced
HZ30	30Hz interlaced
HZ30_SG	30Hz noninterlaced with sync on green
A343	RS-343 component RGB (1280×960), if supported by hardware
HDTV	HDTV format, if supported by hardware
PAL	PAL- 625 line component RGB (768×575) or SECAM
NTSC	NTSC - RS1070A 525 line component RGB (768×575)
VGA	VGA component RGB (640×497)
PR60	Pixel replication of one-quarter resolution 60Hz format

Table 5-6 Monitor Types

getothermonitor

`getothermonitor()` returns the other monitor types supported by the hardware. Most systems are not limited to one optional video mode. The display hardware can support all the video modes. `getothermonitor()` normally returns `MON_ALL` showing that all monitor types are supported. `getothermonitor()` returns `MON_GEN_ALL` if the optional genlock board is installed in the system.

setvideo

`setvideo()` sets the specified video hardware register, *reg*, to the indicated *value*. `setvideo()` and `getvideo()` support several video boards. The DE_R1 is physically present on IRIS-4D / B / G / GT / GTX systems, and is emulated on other systems.

getvideo

`getvideo()` returns the value of the specified video hardware register. The returned value of `getvideo()` is the one read from register *reg*, or -1, which indicates that *reg* is not a valid register, or that you queried a video register on a system without that particular board installed

Table 5-7 lists the video register values for `setvideo()` and `getvideo()`.

Video Option Board	Register
Display Engine Board	DE_R1
CG2 / CG3 Composite Video and Genlock Board	CG_CONTROL
	CG_CPHASE
	CG_HPHASE
	CG_MODE
VP1 Live Video Digitizer Board	VP_ALPHA
	VP_BRITE
	VP_CMD
	VP_CONT
	VP_DIGVAL
	VP_FBXORG
	VP_FBYORG
	VP_FGMODE
	VP_GBXORG
	VP_GBYORG
	VP_HBLANK
	VP_HEIGHT
	VP_HUE
	VP_MAPADD

Table 5-7 Video Register Values

Video Option Board	Register
	VP_MAPBLUE
	VP_MAPGREEN
	VP_MAPRED
	VP_MAPSRC
	VP_MAPSTROBE
	VP_PIXCNT
	VP_SAT
	VP_STATUS0
	VP_STATUS1
	VP_VBLANK
	VP_WIDTH

Table 5-7 (continued) Video Register Values

videocmd

`videocmd()` initializes the Live Video Digitizer option. If you don't have a Live Video Digitizer, you might want to skip this section.You can initialize the Live Video Digitizer in either RGB or composite video mode, for both NTSC and PAL video sources. The *cmd* parameter initiates the specified command. Table 5-8 lists the values for *cmd*, which are defined in the file *gl/vp1.h*.

Token	Description
VP_INITNTSC_COMP	Initialize the optional Live Video Digitizer for a composite NTSC video source
VP_INITNTSC_RGB	Initialize the Live Video Digitizer for an RGB NTSC video source
VP_INITPAL_COMP	Initialize the Live Video Digitizer for a composite PAL video source

Table 5-8 Live Video Digitizer Commands

Token	Description
VP_INITPAL_RGB	Initialize the Live Video Digitizer for an RGB PAL video source

Table 5-8 Live Video Digitizer Commands