Video Capture Developer's Guide and API Reference

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: http://www.qnx.com/

Electronic edition published: Thursday, February 20, 2014

Table of Contents

About this Guide and Reference

The video capture framework provides applications the ability to capture frames from a video input source. These frames can be passed to a graphics component such as Screen for display.

To find out about:	See:
How to use the video capture API	*Video Capture Developer's Guide* (p. 9)
The video capture header files and libraries	*Header files and libraries* (p. 10)
The tasks required to capture video	*Implementing video capture* (p. 12)
A sample program you can use as a reference	*Sample video capture program* (p. 15)
Arrays used for video capture and the properties that can be applied to them	*Properties applied to arrays* (p. 17)
Vidoe capture contexts	*Contexts* (p. 19)
Dynamically allocated and statically allocated video capture buffers	*Buffers* (p. 20)
Configuring your video capture implementation for different platforms	*Platform-specific considerations* (p. 23)
The video capture API	*Video Capture API Reference* (p. 25)

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications.

The following table summarizes our conventions:

Reference	Example
Code examples	`if( stream == NULL )`
Command options	`-lR`
Commands	`make`
Environment variables	***PATH***
File and pathnames	`/dev/null`
Function names	*exit()*
Keyboard chords	**Ctrl –Alt –Delete**
Keyboard input	`Username`
Keyboard keys	`Enter`
Program output	`login:`
Variable names	*stdin*
Parameters	*parm1*
User-interface components	**Navigator**
Window title	**Options**

We use an arrow in directions for accessing menu items, like this:

You'll find the Other... menu item under **Perspective → Show View** .

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we use a forward slash (/) as a delimiter in all pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

Technical support

Technical assistance is available for all supported products.

To obtain technical support for any QNX product, visit the Support area on our website (*www.qnx.com*). You'll find a wide range of support options, including community forums.

Chapter 1
Video Capture Developer's Guide

The video capture framework provides applications the ability to capture frames from a video input source. These frames can be passed to a graphics component such as Screen for display.

This guide describes the video capture framework and the tasks involved in implementing video capture in your project. For detailed information about the video capture API, see the "*Video Capture Library API Reference* (p. 25)".

Video Capture on its own does not display the frames.

Header files and libraries

The video capture framework uses common and board-specific header files and libraries.

Header files

To use video capture your application needs to include the following header files:

- the common header file `vcapture/capture.h`
- the `vcapture/capture-*-ext.h` header file(s) for:
  - the SOC (system-on-a-chip) on your board (e.g., `vcapture/capture-j5-ext.h` for a Jacinto 5 board)
  - the decoder on your board (e.g., `vcapture/capture-adv-ext.h` for an ADV* decoder

Libraries

Video capture is shipped with an empty implementation of the video capture library `libcapture.so`. To link with this library use the `-lcapture` command.

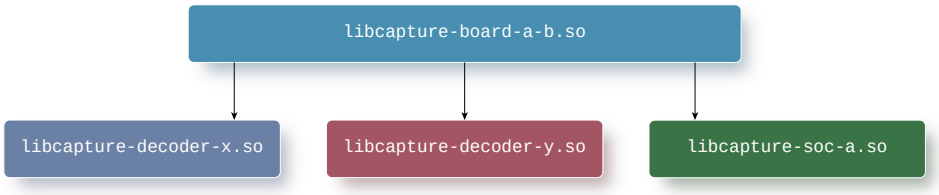

Figure 1: Overview of video capture libraries

At startup, you need to create an appropriate symbolic link (often a proc link) that points to the actual implementation of this library for your board. For example:

```
ln –sP /usr/lib/libcapture-board-j5-evm.so /usr/lib/libcapture.so
```

The board DLL will pull in the board-specific decoder(s), the SOC, and all required shared libraries.

Video capture uses board-specific versions of the following libraries:

`libcapture-board-*-*.so`

Mandatory library with knowledge of the board's SOC and decoder chip, and the number of capture devices and sources available on the board. This library redirects the video capture functions to `libcapture-soc-*.so` and `libcapture-decoder-*.so`.

`libcapture-decoder-*.so`

Optional library that initializes and applies properties to decoders (e.g. `libcapture-decoder-tvp5158.so` for the TI TVP5158 decoder). This library implements only the context and property related *capture_*_context()* and *capture_*_property_*()* functions. The source code for the functions in this library is in `decoder.c`.

On some boards, the video capture framework doesn't include a `libcapture-decoder-*.so` library, leaving control of the decoder to the decoder utility. In this case, a special board DLL (e.g. `libcapture-board-*-no-decoder.so`), which only pulls in a `libcapture-soc-*.so` library is used.

`libcapture-soc-*.so`

Mandatory library for the board's SOC.

Every board-specific library has the definition of the entire API set, as defined in `capture.h`. See *Video Capture API Library Reference* (p. 25) for more information.

Implementing video capture

Video capture involves several tasks, including reserving memory, connecting to a capture device, and releasing memory once the capture is finished.

Overview

To capture video, you need to:

- reserve memory for the video frame and metadata buffers
- connect to the capture device and set up the capture context
- validate the capture device's properties
- set the capture parameters
- start the video capture
- stop the capture
- destroy the buffers to release the memory they use

Note the following about video capture:

- The processes that use capture must be privileged processes.
- The video capture service does not handle displaying the captured video. For this, you need to use another resource, such a Screen.

Video capture code that you can use for reference is available in "Sample video capture program".

Preparation

Before it starts video capture, your program needs to:

- Set up your video capture input, using the combination of `CAPTURE_PROPERTY_DEVICE` and `CAPTURE_PROPERTY_SRC_INDEX` appropriate for your board.
- Connect to a screen and create a window. See the *Screen Graphics Subsystem Developer's Guide* for more information.
- Use your Screen API to create buffers and get pointers to these buffers, which you can pass to the video capture functions.

If your hardware doesn't support dynamic memory allocation and it requires you to use memory in a preallocated location, you will need to use *capture_create_buffers()* to create your buffers. For more information about this special case, see "Video capture buffers" in this guide.

Video capture tasks

To capture video, you need to perform the following tasks:

Set up context

1. Video capture requires a video capture context to which you can connect your input device. To create a video capture context, call *capture_create_context()*. This function returns a pointer to the capture context in the `capture_context_t` data structure, which you will then pass to your other functions during the video capture session.

Validate capture device properties

1. Call *capture_get_property_p()* to get information about the capture device.
2. Call *capture_is_property()* to check if the driver supports a property. Call this function once for each property you need to check.

Set the capture parameters

1. Call *capture_set_property_i()* for each capture property you need to set, using the CAPTURE_PROPERTY_* constants to set the device, brightness, destination buffers, etc.
2. Call *capture_set_property_p()* to hand the pointer to your properties array to the video capture library.

Capture the video frames

1. Make a final call to *capture_set_property_i()* with the second argument set to CAPTURE_ENABLE and the third argument set to `1` (one) to instruct the driver to start video capture when *capture_update()* is called.
2. Call *capture_update()* to start the video capture.
3. Create a loop to call *capture_get_frame()* to get the video frames from the hardware.
4. You can use the Screen function *screen_post_window()* to post the video frame for display in your screen window.
5. After you have posted a frame, mark the buffer that was used for the frame as available for reuse by calling *capture_release_frame()*.

Stop and clean up

1. When you want to stop video capture, call *capture_set_property_i()* with the second argument set to CAPTURE_ENABLE and the third argument set to `0` (zero) to disable video capture.

2. Call *capture_update()* to stop video capture.
3. If you will not restart video capture again immediately (the session is stopped rather than paused), your application must call *capture_destroy_context()* to destroy all contexts before it releases the capture buffers and exits.

- The *capture_destroy_context()* function is *not* signal handler safe! For recommendations on how to use *capture_destroy_context()* see the documentation for this function.
- You can't count on the OS being able to adequately clean up after your application exits, if the application does not destroy all the contexts it created for video capture. Failure to destroy a context before exiting can lead to memory corruption and unpredicatable system behavior.

Sample video capture program

This sample video capture code can be used for reference when building an application that uses video capture.

The following code sample shows how the video capture API can be used in an application. Note that the sample code doesn't include error checking, which may be quite useful in a production application.

```
main() {
    void    *pointers[n_pointers] = { 0 };
    // connect to screen
    // create a window
    // create screen buffers
    // obtain pointers to the buffers

    // Connect to a capture device
    capture_context_t context = capture_create_context( flags );
    if( !context ) {
        // TODO: Handle errors...
    }

    const char *info = NULL;
    capture_get_property_p( context, CAPTURE_PROPERTY_DEVICE_INFO, &info );
    fprintf( stderr, "device-info = '%s'\n", info );

    // Validate device's properties
    if( !capture_is_property( context, CAPTURE_PROPERTY_BRIGHTNESS )
          || !capture_is_property( context, CAPTURE_PROPERTY_CONTRAST )
          || !capture_is_property( context, ... )
     ) {
        capture_destroy_context( context );
        fprintf( stderr, "Unable to use buffer. Driver doesn't support some required properties.\n" );
        return EXIT_FAILURE;
    }

    // setup capture parameters
    capture_set_property_i( context, CAPTURE_PROPERTY_DEVICE, 1 );
    capture_set_property_i( context, CAPTURE_PROPERTY_BRIGHTNESS, 10 );
    capture_set_property_i( context, CAPTURE_PROPERTY_DST_NBUFFERS, n_pointers );
    capture_set_property_pv( context, CAPTURE_PROPERTY_DST_BUFFERS, n_pointers, pointers );

    // tell the driver to start capturing (when capture_update() is called).
    capture_set_property_i( context, CAPTURE_ENABLE, 1 );

    // commit changes to the H/W -- and start capturing...
    capture_update( context, 0 );

    while( capturing ) {
        int n_dropped;

        // get next captured frame...
        int idx = capture_get_frame( context, CAPTURE_TIMEOUT_INFINITE, flags );

        // the returned idx-ed pointer is 'locked' upon return from the capture_get_frame()
        // this buffer will remain locked until the capture_get_frame() is called again.

        // update screen
        screen_post_window( win, buf[idx], n_dirty_rects, dirty_rects, flags );

        // Mark the buffer identifed by the idx as available for capturing.
        capture_release_frame( context, idx );
    }

    // stop capturing...
    capture_set_property_i( context, CAPTURE_ENABLE, 0 );
    capture_update( context, 0 );

    ...
}
```

The sample above posts then releases each frame buffer. In a production application, you should use at least two frame buffers, so that you do not have to release a frame before the next one is posted. This will avoid delays and jitter.

Properties applied to arrays

Some of the properties defined in the video capture API are applied to arrays. Properties applied to arrays require special attention.

About arrays

Your application must set pointers to the arrays for which it needs to set properties, or for which it needs the capture library to get data from the video capture device. These pointers are stored in the capture library's *context*, which the application created by calling *capture_create_context()*. For instance, your client application can use the access modifiers defined by the CAPTURE_PROPERTY_FRAME_* properties to access and set or get the contents of the arrays in the current context.

Array resources such as CAPTURE_PROPERTY_FRAME_FLAGS and CAPTURE_PROPERTY_FRAME_SEQNO are not allocated by default. They need to be set, then passed to the video capture driver. Your application must:

1. Allocate the CAPTURE_PROPERTY_FRAME_* arrays with sufficient memory to hold the information they need. The number of elements in each array must be at least equal to the number of buffers you are using (set in CAPTURE_PROPERTY_FRAME_NBUFFERS).
2. Call *capture_set_property_p()* to pass a pointer to the array to the capture library.

The capture library stores this pointer to the array and updates the array whenever appropriate. Your applciation should read the data from the array to get updates whenever appropriate.

To instruct the capture library to stop collecting and providing data for a property, set the value of the property to NULL.

If the array for a property isn't set, the capture library won't request information about that property from the hardware. Since requests to hardware are expensive operations, this behavior reduces overhead.

Example

The code snippet below is an example of how to use arrays:

```
nbuffers = 3;
// allocate a seqno buffer.
uint32_t seqno[nbuffers];

// tell the capture library to use this array and update it when
// frames are captured.
capture_set_property_p( ctx, CAPTURE_PROPERTY_FRAME_SEQNO, &seqno );
...
// get a captured frame
int idx = capture_get_frame( ctx, ... );
```

```
// the frame data and the buffer of the 'idx' frame is locked.
if( -1 != idx ) {
// it is safe to access the contents of the seqno[idx]
printf( "captured a frame, seqno = %u\n", seqno[idx] );
}
...
capture_release_frame( ctx, idx );
...
// no longer safe to access seqno[idx], since the data may
// no longer be valid.
```

Contexts

Video capture uses *contexts* for storing and communicating information, such as frame properties.

About contexts

Contexts are used by the video capture framework to store and communicate device and processing properties. They are created by calling *capture_create_context()*, which returns a pointer to the context.

A video capture device can have one or more sources (or inputs). You can create multiple contexts, but you can have only one operational context for each device-source combination. For example, you can have an operational context for Device 1, Source 1, and another operational context for Device 1, Source 2, but you can't have a second operational context for either of these.

If you create more than one context for a device-source combination, the first context that enables capturing will be the context used by *capture_get_frame()*. Attempts to use the other contexts for that device-source combination will fail when *capture_update()* is called to apply instructions to the device.

Destroying contexts at exit

Your application must call *capture_destroy_context()* to destroy all contexts before it releases the capture buffers and exits. You can't count on the OS being able to adequately clean up after your application exits if the application doesn't destroy all the contexts it created for video capture.

Failure to destroy a context before exiting can lead to memory corruption and unpredictable system behavior.

Buffers

The driver or the client application can allocate memory for video frame data buffers. Only the client application can allocate metadata buffers.

Buffer allocation

Video capture uses *frame buffers* to store the video capture frame data and *metadata buffers* to store video capture metadata. Video frame buffers can be either *driver-allocated* or *application-allocated*. Metadata buffers can only be *application-allocated*. Both frame data and metadata buffers can also be *not allocated*.

Driver-allocated buffers

Driver-allocated memory is managed by the driver: the driver allocates and frees this memory. The application can use buffers using this memory only when *capture_get_property_p()* defines them as valid.

To allocate driver-controlled memory for video frame buffers or metadata buffers, call *capture_create_buffers()*. Calling this function:

1. creates a video capture context
2. connects to video device
3. allocates buffer memory for this video capture context

Allocation of memory for driver-allocated buffer memory differs, based on several conditions:

Hardware doesn't support dynamic buffer allocation

If the hardware doesn't support dynamic buffer allocation, then the driver must always re-use previously allocated buffers (for example, memory that was set aside for these buffers at startup).

Buffers may be at a predetermined (hard-coded) RAM address. The driver's allocate function merely returns a pointer to this memory that *capture_get_property_p()* mapped into the application's address space.

Hardware supports dynamic buffer allocation

If the hardware supports dynamic buffer allocation, the driver can either allocate new buffers or reuse buffers it has used previously, provided that these buffers were driver allocated and are suitable.

Application-allocated buffers and buffers that are not allocated can't be reused as driver-allocated buffers. (See “Not allocated” below.)

To hand driver-allocated memory over so it can be controlled by the calling application, call *capture_set_property_p()*.

Application-allocated buffers

Application-allocated memory is managed by the application: the application allocates and frees this memory. The driver can use the memory when the video capture API marks it as valid.

Application-allocated memory includes any buffer that isn't allocated by the video capture driver. For example, the application using video capture might acquire a buffer from another component, such as Screen. Because the driver didn't allocate the buffer, it is considered an application-allocated driver.

If your application allocates memory for your video capture buffers, then the driver won't allocate any memory (unless the hardware requires such buffers to exist even when they are not used by the application).

If the hardware doesn't support application-allocated memory, then your application should use *capture_create_buffers()* to create buffers.

Not allocated

There are several case where a buffer can be *not allocated*:

- The client application expressly sets up the video capture context so that the video capture library does *not* capture the video frames, but can still get frame metadata. The application can call *capture_get_frame()* to get a frame index, then look up the metadata for the frame. The video frame buffers *must not* be looked up or used. See “Unintentional freeing of driver-allocated buffers” below.
- The special case that can occur when the driver or application has allocated a buffer, but the driver later turns off an expensive (resource-intensive) hardware feature that was supposed to use this buffer. When the feature is turned off, the buffer becomes *not allocated*.

Unintentional freeing of driver-allocated buffers

Setting a buffer pointer to `NULL` sets the buffer to *not allocated*, which:

- causes the driver to cease using application-allocated buffer memory
- may cause the driver to free previously driver-allocated memory

Do not get a pointer to a buffer with *capture_get_property_p()*, then set the same pointer with *capture_set_property_p()*.

If the buffer is *driver-allocated*, this sequence of calls will cause the driver to free the buffer referenced by the pointer, then assume that the application owns the now nonexistent buffer, with unpredictable results.

If the buffer in question was initially *application-allocated*, then no ill effects occur.

Platform-specific considerations

Video capture may require adjustments to accommodate how different hardware platforms handle tasks such as buffer allocation.

Different hardware platforms handle tasks differently. To accommodate these differences, you may need to make adjustments to how your system handles video capture.

For example, we found that on some boards an unstable capture link could cause apparently random system crashes:

- The capture buffers are allocated by the WFD driver, which usually allocates buffers with the size specified by the application.
- Some synchronization data could be lost due to the unstable link, which causes the hardware to write data beyond the buffer boundary, with suprising results.

The only restrictions our application could impose on the hardware were maximum frame height and maximum frame width, selected from a limited set of values. To solve the problem caused by the unstable capture link, we changed the WFD driver to allocate the capture buffer of a size equal to the maximum frame height times the maximum frame width (`CAPTURE_PROPERTY_DST_HEIGHT * CAPTURE_PROPERTY_DST_WIDTH`).

For more information about buffer properties, see “Destination buffer properties”.

Chapter 2
Video Capture Library API Reference (capture.h)

The video capture API includes all the functions, data structures, and constants needed for video capture. It does not handle video display, which should be handled by another component, such as Screen.

The video capture API is thread safe, unless stated otherwise for a specific function.

Properties

The video capture API includes constants, data types, enumerated values, and macros specifying video capture properties.

The following constants can be passed as function arguments to specify video capture behavior:

CAPTURE_PROPERTY macro

CAPTURE_PROPERTY stores bit maps of information, either retrieved from a video capture driver and device, or specified by the user application and passed to the video capture library, the video capture driver, and the device. These bit maps are placed in the four bytes of a `uint32_t` value:

CAPTURE_PROPERTY(a, b, c, d)

```
( (a) << 24 | (b) << 16 | (c) << 8 | (d) )
```

Shifts the bits for CAPTURE_PROPERTY_* values.

Video capture behavior

CAPTURE_FLAG_LATEST_FRAME

`0x0001`

Get the latest frame and discard all the other queued frames.

CAPTURE_TIMEOUT_INFINITE

`-1ULL`

Never timeout; wait for frame indefinitely.

CAPTURE_TIMEOUT_NO_WAIT

`0`

Return immediately, even if there is no frame.

Interfaces, threads, offsets

The following values specify the video capture interface type, thread priority, and YUV offsets:

CAPTURE_PROPERTY_PLANAR_OFFSETS

```
CAPTURE_PROPERTY( 'Q', 'P', 'L', 'O' )
```

Read/Write `[3] int32_t`

The offset from the base address for each of the Y, U, and V components of planar YUV formats.

CAPTURE_PROPERTY_THREAD_PRIORITY

`CAPTURE_PROPERTY( 'Q', 'T', 'P', 'R' )`

Read/Write `int`

The scheduling priority of the capture thread. The default value is the priority for the application + 20.

CAPTURE_PROPERTY_INTERFACE_TYPE

`CAPTURE_PROPERTY('Q','P','I','F')`

Read/Write `uint32_t`

The interface type. See "Interface types" below.

Interface types

The following enumerated values are used to specify the interface type:

```
enum capture_iface_type {
 CAPTURE_IF_PARALLEL = 0,
 CAPTURE_IF_MIPI_CSI2,
};
```

CAPTURE_IF_PARALLEL

`0`

The interface is parallel.

CAPTURE_IF_MIPI_CSI2

The interface is a MIPI CSI2 interface.

See also "Data bus, and clock and data lane properties"

Debugging

CAPTURE_PROPERTY_VERBOSITY

`CAPTURE_PROPERTY( 'Q', 'V', 'B', 'R' )`

Read/Write `uint32_t`

Set the log verbosity level. Default is `0`.; increase this value to increase log verbosity for debugging.

Driver and device properties

The video capture API includes constants and macros to be used when specifying and retrieving driver and device properties.

The following define driver and device properties:

CAPTURE_PROPERTY_DEVICE_INFO

CAPTURE_PROPERTY('Q', 'I', 'N', 'F')

Read `const char *`

Returns string information about the video capture driver and device. All drivers support this property.

CAPTURE_ENABLE

CAPTURE_PROPERTY('Q', 'E', 'N', 'A')

Read/Write `uint32_t`

Capture start (`1`) and stop(`0`).

CAPTURE_PROPERTY_NDEVICES

CAPTURE_PROPERTY('Q', 'N', 'D', 'V')

Read `uint32_t`

The number of supported capture units.

CAPTURE_PROPERTY_DEVICE

CAPTURE_PROPERTY('Q', 'D', 'E', 'V')

Read/Write `uint32_t`

The active capture device in this context.

CAPTURE_PROPERTY_NSOURCES

CAPTURE_PROPERTY('Q', 'N', 'S', 'R')

Read`uint32_t`

Number of available source inputs; available after the device is set.

CAPTURE_PROPERTY_SRC_INDEX

CAPTURE_PROPERTY('Q', 'S', 'I', 'D')

Read/Writer `uint32_t`

The device video capture unit.

CAPTURE_PROPERTY_CONTRAST

`CAPTURE_PROPERTY( 'Q', 'C', 'O', 'N' )`

Read/Write `int32_t`

Contrast (`-128` to `127`).

CAPTURE_PROPERTY_BRIGHTNESS

`CAPTURE_PROPERTY( 'Q', 'B', 'R', 'I' )`

Read/Write `int32_t`

Brightness (`-128` to `127`).

CAPTURE_PROPERTY_SATURATION

`CAPTURE_PROPERTY( 'Q', 'S', 'A', 'T' )`

Read/Write `int32_t`

Color saturation (`-128` to `127`).

CAPTURE_PROPERTY_HUE

`CAPTURE_PROPERTY( 'Q', 'H', 'U', 'E' )`

Read/Write `int32_t`

Color hue (`-128` to `127`).

CAPTURE_PROPERTY_DEINTERLACE_FLAGS

`CAPTURE_PROPERTY( 'Q', 'D', 'E', 'I' )`

Read/Write `uint32_t`

Deinterlacing flag (bit-field).

CAPTURE_PROPERTY_DEINTERLACE_MODE

`CAPTURE_PROPERTY( 'Q', 'D', 'E', 'M' )`

Read/Write`uint32_t`

Deinterlacing mode; see “Deinterlacing”.

Data bus, and clock and data lane properties

The video capture API includes values for the data bus width, and for the clock and data lane properties.

Frame property arrays

The video capture API uses the CSI2 clock and data lane properties described below. Lane positions are all specified as follows:

Value	Number or position
`0` to `4`	Position number.
`-1`	Don't set the number or the position. Use whatever default number or position is already set in the hardware.

CAPTURE_PROPERTY_DATA_BUS_WIDTH

CAPTURE_PROPERTY('Q', 'D', 'B', 'W')

Read/Write `int` The data bus width, for parallel interfaces. Valid values are 8, 10, 16, etc.

CAPTURE_PROPERTY_CSI2_NUM_DATA_LANES

CAPTURE_PROPERTY('Q','C','N','D')

Read/Write `int` Number of CSI2 data lanes.

CAPTURE_PROPERTY_CSI2_CLK_LANE_POS

APTURE_PROPERTY('Q','C','C','P')

Read/Write `int` Position of CSI2 clock lane.

CAPTURE_PROPERTY_CSI2_DATA0_LANE_POS

CAPTURE_PROPERTY('Q','C','D','0')

Read/Write `int` Position of CSI2 data lane 0.

CAPTURE_PROPERTY_CSI2_DATA1_LANE_POS

CAPTURE_PROPERTY('Q','C','D','1')

Read/Write `int` Position of CSI2 data lane 1.

CAPTURE_PROPERTY_CSI2_DATA2_LANE_POS

CAPTURE_PROPERTY('Q','C','D','2')

Read/Write `int` Position of CSI2 data lane 2.

CAPTURE_PROPERTY_CSI2_DATA3_LANE_POS

CAPTURE_PROPERTY('Q','C','D','3')

Read/Write `int` Position of CSI2 data lane 3.

I²C decoder path and slave address

The video capture API includes constants for specifying the path and slave address for an I²C decoder.

The following specify the I²C decoder path and slave address:

CAPTURE_PROPERTY_DECODER_I2C_PATH

CAPTURE_PROPERTY('Q', 'D', 'I', 'P')

Read/Write `const char *`

Device path of the I²C decoder (e.g. `/dev/i2c0`).

CAPTURE_PROPERTY_DECODER_I2C_ADDR

CAPTURE_PROPERTY('Q', 'D', 'I', 'A')

Read/Write `uint32_t`

Slave address of the I²C decoder.

Video standards

The video capture API includes constants and macros to be used when specifying video standards.

Standards definitions

The video capture API uses the following constants for standards:

CAPTURE_PROPERTY_NORM

CAPTURE_PROPERTY('Q', 'N', 'O', 'R')

Read/Write `const char *`

Set the video standard. See “Standards macros” below.

CAPTURE_NORM_AUTO

"AUTO"

Read/Write Use auto-detection to get the video standard.

CAPTURE_PROPERTY_CURRENT_NORM

CAPTURE_PROPERTY('Q', 'Q', 'N', 'M')

Read `const char *`

Return the current detected video standard. See "Standards macros" below.

CAPTURE_NORM_NONE

"NONE"

There is no input.

CAPTURE_NORM_UNKNOWN

"UNKNOWN"

Detected standard is not known.

Standards macros

The following macros set standards values used by CAPTURE_PROPERTY_NORM and CAPTURE_PROPERTY_CURRENT_NORM:

CAPTURE_NORM_NTSC_M_J

"NTSC_M_J"

CAPTURE_NORM_NTSC_4_43

"NTSC_4_43"

CAPTURE_NORM_PAL_M

"PAL_M"

CAPTURE_NORM_PAL_B_G_H_I_D

"PAL_B_G_H_I_D"

CAPTURE_NORM_PAL_COMBINATION_N

"PAL_COMBINATION_N"

CAPTURE_NORM_PAL_60

"PAL_60"

CAPTURE_NORM_SECAM

"SECAM"

Polarity

The video capture API includes definitions for specifying polarity.

All polarity properties are specified as follows:

Value	Polarity
`0`	Set polarity to *not inverted*.
`1`	Set polarity to *inverted*.
`-1`	Don't set the polarity. Use whatever polarity is already set.

CAPTURE_PROPERTY_INVERT_FID_POL

CAPTURE_PROPERTY('Q', 'L', 'F', 'I')

Read/Write `int`

Specifies whether the field ID signal polarity is inverted.

CAPTURE_PROPERTY_INVERT_VSYNC_POL

CAPTURE_PROPERTY('Q', 'L', 'H', 'S')

Read/Write `int`

Specifies whether the vertical synchronization polarity is inverted.

CAPTURE_PROPERTY_INVERT_HSYNC_POL

CAPTURE_PROPERTY('Q', 'L', 'V', 'S')

Read/Write `int`

Specifies whether the horizontal synchronization polarity is inverted.

CAPTURE_PROPERTY_INVERT_CLOCK_POL

CAPTURE_PROPERTY('Q', 'L', 'P', 'C')

Read/Write `int`

Specifies whether the clock polarity is inverted.

CAPTURE_PROPERTY_INVERT_DATAEN_POL

CAPTURE_PROPERTY('Q', 'L', 'D', 'E')

Read/Write `int`

Specifies whether the “data_en” pin/signal polarity is inverted.

CAPTURE_PROPERTY_INVERT_DATA_POL

`CAPTURE_PROPERTY( 'Q', 'L', 'D', 'A' )`

Read/Write `int`

Specifies whether the data input polarity is inverted.

Deinterlacing properties and enumerated values

The video capture API includes enumerated values that specify deinterlacing behavior.

About deinterlacing

Deinterlacing can use a variety of techniques:

Adaptive

Use a motion-adaptive filter. This type of deinterlacing is usually done by the hardware.

Bob

Take the lines of each field and double them. This technique retains the original temporal resolution.

Bob 2

Discard one field out of each frame to improve the video quality. The temporal resolution is halved, however.

Weave

Combine two consecutive fields together. The temporal resolution is halved: the resulting frame rate is half of the original field rate.

Weave 2

Similar to *weave*, but the resulting frame rate is the same as the original field rate.

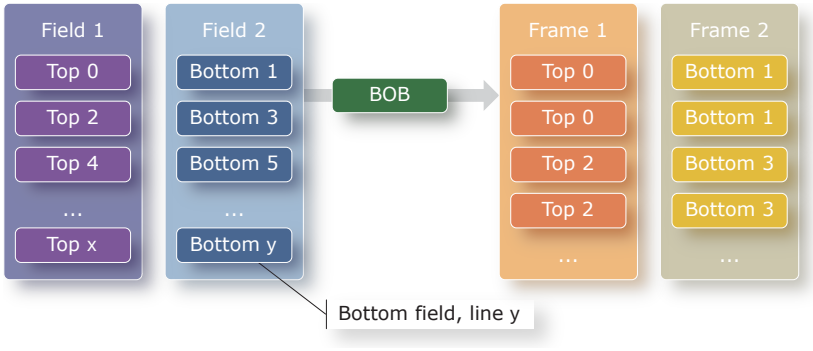

Figure 2: Deinterlacing in BOB mode

Enumerated values

```
enum capture_deinterlace_mode {
 CAPTURE_DEINTERLACE_NONE_MODE = 0,
 CAPTURE_DEINTERLACE_WEAVE_MODE,
 CAPTURE_DEINTERLACE_BOB_MODE,
 CAPTURE_DEINTERLACE_BOB2_MODE,
 CAPTURE_DEINTERLACE_WEAVE2_MODE,
 CAPTURE_DEINTERLACE_MOTION_ADAPTIVE_MODE,
};
```

The following enumerated types are used for setting the deinterlacing mode:

CAPTURE_DEINTERLACE_NONE_MODE

0

Don't deinterlace.

CAPTURE_DEINTERLACE_WEAVE_MODE

Use weave deinterlacing mode.

CAPTURE_DEINTERLACE_BOB_MODE

Use bob deinterlacing mode.

CAPTURE_DEINTERLACE_BOB2_MODE

Use alternate bob deinterlacing mode.

CAPTURE_DEINTERLACE_WEAVE2_MODE

Use alternate weave deinterlacing mode.

CAPTURE_DEINTERLACE_MOTION_ADAPTIVE_MODE

Use motion adaptive deinterlacing mode.

Buffer properties

The CAPTURE_PROPERTY_*_BUFFER_ types store buffer properties:

CAPTURE_PROPERTY_MIN_NBUFFERS

CAPTURE_PROPERTY('Q', 'M', 'N', 'B')

Read `uint32_t`

Minimum number of buffers required for a specific deinterlacing mode.

For properties used for deinterlacing frames from external sources (e.g., USB stick) see “External source properties”.

Source buffer properties

The video capture API includes definitions for the source buffer properties.

Source buffer properties include:

CAPTURE_PROPERTY_SRC_FORMAT

CAPTURE_PROPERTY('Q', 'S', 'F', 'O')

Read/Writer `uint32_t`

Source buffer format.

CAPTURE_PROPERTY_SRC_STRIDE

CAPTURE_PROPERTY('Q', 'S', 'F', 'S')

Read/Writer `uint32_t`

Source buffer stride, in bytes.

CAPTURE_PROPERTY_SRC_WIDTH

CAPTURE_PROPERTY('Q', 'S', 'W', 'I')

Read/Writer `uint32_t`

The width of the source, in pixels.

CAPTURE_PROPERTY_SRC_HEIGHT

CAPTURE_PROPERTY('Q', 'S', 'H', 'E')

Read/Writer `uint32_t`

The width of the source, in pixels.

CAPTURE_PROPERTY_CROP_WIDTH

CAPTURE_PROPERTY('Q', 'C', 'W', 'I')

Read/Writer `uint32_t`

The source viewport width, in pixels.

CAPTURE_PROPERTY_CROP_HEIGHT

CAPTURE_PROPERTY('Q', 'C', 'H', 'E')

Read/Writer `uint32_t`

The source viewport height, in pixels.

CAPTURE_PROPERTY_CROP_X

CAPTURE_PROPERTY('Q', 'C', 'X', 'P')

Read/Writer `uint32_t`

Source viewport x offset.

CAPTURE_PROPERTY_CROP_Y

CAPTURE_PROPERTY('Q', 'C', 'Y', 'P')

Read/Writer `uint32_t`

Source viewport y offset.

Destination buffer properties

The video capture API includes definitions for the source buffer properties.

The video capture API uses the following definitions for destination buffer properties:

CAPTURE_PROPERTY_DST_FORMAT

CAPTURE_PROPERTY('Q', 'D', 'F', 'F')

Read/Writer `uint32_t`

Destination buffer format.

CAPTURE_PROPERTY_DST_WIDTH

CAPTURE_PROPERTY('Q', 'D', 'F', 'W')

Read/Writer `uint32_t`

Destination frame width, in pixels.

CAPTURE_PROPERTY_DST_HEIGHT

CAPTURE_PROPERTY('Q', 'D', 'F', 'H')

Read/Writer `uint32_t`

Destination frame height, in pixels.

CAPTURE_PROPERTY_DST_STRIDE

CAPTURE_PROPERTY('Q', 'D', 'F', 'S')

Read/Writer `uint32_t`

Destination frame stride, in bytes.

CAPTURE_PROPERTY_DST_NBYTES

CAPTURE_PROPERTY('Q', 'D', 'F', 'B')

Read/Writer `uint32_t`

Size of destination buffer, in bytes.

For information about CAPTURE_PROPERTY, see "Properties".

Each allocated destination buffer (CAPTURE_PROPERTY_DST_NBYTES) must be at least the product of the destination stride and the destination frame height (CAPTURE_PROPERTY_DST_STRIDE * CAPTURE_PROPERTY_DST_HEIGHT).

Frame properties

The video capture API includes arrays for storing and communicating video frame properties.

Frame property arrays

The video capture API uses the arrays below for video frame properties. All these properties (except CAPTURE_PROPERTY_FRAME_BUFFERS, which is a pointer to the capture buffers) are indexed by the buffer index:

CAPTURE_PROPERTY_FRAME_BUFFERS

CAPTURE_PROPERTY('Q', 'F', 'B', 'A')

Read/Write `[] void*`

Pointers to the video capture buffers.

CAPTURE_PROPERTY_FRAME_TIMESTAMP

CAPTURE_PROPERTY('Q', 'F', 'B', 'T')

Read `[] uint64_t`

An array of CLOCK_MONOTONIC timestamps.

CAPTURE_PROPERTY_FRAME_TIMECODE

CAPTURE_PROPERTY('Q', 'F', 'B', 'C')

Read `[] struct smpte_timestamp`

An array of SMTPE timestamps.

CAPTURE_PROPERTY_FRAME_SEQNO

CAPTURE_PROPERTY('Q', 'F', 'B', 'S')

Read `[] uint32_t`

An array of frame sequence numbers.

CAPTURE_PROPERTY_FRAME_FLAGS

`CAPTURE_PROPERTY( 'Q', 'F', 'B', 'F' )`

Read `[] uint32_t`

An array of frame flags. See "Frame flags" below.

CAPTURE_PROPERTY_FRAME_NBYTES

`CAPTURE_PROPERTY( 'Q', 'F', 'B', 'B' )`

Read `[] uint32_t`

An array of frame sizes, in bytes.

CAPTURE_PROPERTY_FRAME_NBUFFERS

`CAPTURE_PROPERTY( 'Q', 'F', 'B', 'N' )`

Read/Writer `uint32_t`

The number of destination buffers that have been specified in CAPTURE_PROPERTY_FRAME_BUFFERS.

For information about how to use arrays, see "Properties applied to arrays".

Frame flags

The following flags specify properties for the CAPTURE_PROPERTY_FRAME_FLAGS array:

CAPTURE_FRAME_FLAG_ERROR

`0x0001`

There is an error in the frame.

CAPTURE_FRAME_FLAG_INTERLACED

`0x0002`

The frame is interlaced. For more information about interlacing, see "Deinterlacing enumerated values".

CAPTURE_FRAME_FLAG_FIELD_BOTTOM

`0x0004`

TBD

External source properties

The video capture API includes constants for managing the retrieval and deinterlacing of frames brought in from an external source.

The following constants are used to set properties when getting and deinterlacing frames from videos brought in from an external source such as a USB memory stick:

CAPTURE_FLAG_EXTERNAL_SOURCE

`0x0002`

Bit to set if the context to create is for an external source.

The following properties are used for deinterlacing frames brought in from an external source. They are relevant only when the CAPTURE_FLAG_EXTERNAL_SOURCE flag is set.

CAPTURE_FLAG_FREE_BUFFER

`0x0002`

Request a free buffer in which to put a frame from an external source.

CAPTURE_BUFFER_USAGE_RDONLY

`0x001`

Mark the buffer as read-only.

CAPTURE_PROPERTY_BUFFER_USAGE

`CAPTURE_PROPERTY( 'Q', 'B', 'U', 'S' )`

Read/Write `[] uint32_t`

An array of buffer usage flags. Element `i` indicates if the capture driver has only read (CAPTURE_BUFFER_USAGE_RDONLY) or read/write (CAPTURE_BUFFER_USAGE_RDWR) permission for buffer >`i`. The default is read-write permission.

CAPTURE_PROPERTY_BUFFER_INDEX

`CAPTURE_PROPERTY( 'Q', 'B', 'I', 'X' )`

Write `uint32_t`

The index of the buffer to be injected by *capture_put_buffer()*.

CAPTURE_PROPERTY_BUFFER_NFIELDS

`CAPTURE_PROPERTY( 'Q', 'B', 'N', 'F' )`

Write `uint32_t`

The number of fields contained in the buffer injected by *capture_put_buffer()*.

CAPTURE_PROPERTY_BUFFER_PLANAR_OFFSETS

`CAPTURE_PROPERTY( 'Q', 'B', 'P', 'O' )`

Write `[][3] int32_t`

A per-buffer array. The array has one row per field. Each row indicates the offset from the base address for each of the Y, U, and V components of planar YUV formats.

CAPTURE_PROPERTY_BUFFER_FLAGS

`CAPTURE_PROPERTY( 'Q', 'B', 'F', 'L' )`

Write `[] uint32_t`

A per-buffer array of buffer flag. The flag is a bit-field.

CAPTURE_PROPERTY_BUFFER_SEQNO

`CAPTURE_PROPERTY( 'Q', 'B', 'S', 'N' )`

Write `[] uint32_t`

A per-buffer array of sequence numbers. Each element indicates the sequence number of the field contained in the buffer.

Helper macros

The video capture API includes macros to help with common calculations.

Helper macros include:

Convert *from* interval

CAPTURE_INTERVAL_FROM_MS(x)

```
((x) * 1000000ULL)
```

CAPTURE_INTERVAL_FROM_US(x)

```
((x) * 1000ULL)
```

CAPTURE_INTERVAL_FROM_NS(x)

```
(x)
```

Convert *to* interval

CAPTURE_INTERVAL_TO_MS(x)

```
((x) / 1000000ULL)
```

CAPTURE_INTERVAL_TO_US(x)

```
((x) / 1000ULL)
```

CAPTURE_INTERVAL_TO_NS(x)

```
(x)
```

Field-frame conversion

CAPTURE_INTERVAL_NTSC_FIELD

```
16668333
```

CAPTURE_INTERVAL_NTSC_FRAME

```
(CAPTURE_INTERVAL_NTSC_FIELD * 2)
```

capture_context_t

Pointer to a video capture context.

Synopsis:

```
#include <vcapture/capture.h>
typedef struct _capture_context    *capture_context_t;
```

Library:

`libcapture`

Description:

The `capture_context_t` data structure is a pointer (or handle) to a video capture context. It is populated by a successful call to *capture_context_create()*.

For more information about video capture contexts, see "*Contexts* (p. 19)".

capture_create_buffers()

Allocate memory for video capture.

Synopsis:

```
#include <vcapture/capture.h>

int capture_create_buffers(capture_context_t context,
    uint32_t property)
```

Arguments:

context

Pointer to the video capture context.

property

The video capture frame properties set by *capture_set_property_i()*.

Library:

`libcapture`

Description:

The *capture_create_buffers()* function reserves memory for video capture:

- If your hardware supports dynamic memory allocation, you can simply use your Screen API to create buffers for video capture.
- If your hardware doesn't support dynamic memory allocation and requires you to use memory in a preallocated location, you will need to use *capture_create_buffers()* to create your buffers. For more information about buffers, see "Buffers".

Calls to *capture_create_buffers* are synchronous. The function frees old buffers, creates the new buffers, and returns immediately.

To free an existing buffer without creating a new one:

1. Call *capture_set_property_p()* to set the buffer property to `NULL`.
2. Call *capture_create_buffers()*.

- Call this function only when video capture is *not* in progress.
- If the hardware doesn't support application-allocated memory, calling this function will fail if it is called with a buffer pointer property set to anything other than `NULL`.

- Systems with hardware that supports only driver-allocated memory can simply reject any attempt to take a buffer out of driver-allocated state. On these systems, setting a buffer property to `NULL` returns success, but has no effect on memory allocation.

Do not get a pointer to a buffer with *capture_get_property_p()*, then set the same pointer with *capture_set_property_p()*.

If the buffer is *driver-allocated*, this sequence of calls will cause the driver to free the buffer referenced by the pointer, then assume that the application owns the now nonexistent buffer, with unpredictable results.

If the buffer in question was initially *application-allocated*, then no ill effects occur.

Example:

The code snippet below may be a useful reference on how to use *capture_create_buffers()*.

```
capture_set_property_i( context, CAPTURE_PROPERTY_DST_NBYTES, 320 * 240 * 4 );
capture_set_property_i( context, CAPTURE_PROPERTY_FRAME_NBUFFERS, 5 );

// create 5 frame buffers with 320x240x4 bytes each.
capture_create_buffers( context, CAPTURE_PROPERTY_FRAME_BUFFERS );

// get the buffers..
void   **frame_buffers;

capture_get_property_p( context, CAPTURE_PROPERTY_FRAME_BUFFERS, (void**)&frame_buffers );
```

Returns:

0

Success.

-1

An error occurred (`errno` is set).

Errors:

EINVAL

Invalid argument, or the argument is not a buffer property.

ENOMEM

Unable to allocate requested memory.

ENOSYS

The driver doesn't support buffer allocation.

capture_create_context()

Establish a connection to the capture device and create a video capture context.

Synopsis:

```
#include <vcapture/capture.h>
capture_context_t capture_create_context(uint32_t flags)
```

Arguments:

flags

Bitmask parameter; set to either `0` if the context to be created is for a local device, or to CAPTURE_FLAG_EXTERNAL_SOURCE if the context to be created is for an external source.

Library:

`libcapture`

Description:

The function *capture_create_context()*:

- connects to a video capture device
- creates a new context for video capture
- returns a pointer to the context in `capture_context_t`

You must create a context before you can set video capture properties and start video capture. The context contains both mandatory information, such as the device ID and input source ID (e.g. device 1, input source 2), and optional settings, such as brightness.

You can create more than one context, but you can have only one context in use for each device-source combination. If you have created multiple contexts for the same device-source combination, *capture_get_frame()* will use the first context you have enabled with *capture_set_property_i()*. Calls to *capture_update()* for the other contexts will fail.

Before your application exits, it must call *capture_destroy_context()* to destroy every context it created. Failure to destroy a context before exiting can lead to memory corruption and unpredictable system behavior.

For more information about video capture contexts, see "*Contexts* (p. 19)".

Returns:

A pointer to a new context in `capture_context_t`.

Success.

`NULL`

An error occurred (`errno` is set).

Errors:

`EBUSY`

Unable to create a context because one is already in use.

`EINVAL`

Invalid flags.

capture_destroy_context()

Disconnect from the video capture device, and destroy the context.

Synopsis:

```
#include <vcapture/capture.h>
void capture_destroy_context( capture_context_t context );
```

Arguments:

context

The pointer to the video capture context to destroy.

Library:

`libcapture`

Description:

The function *capture_destroy_context()*:

- disconnects from a video capture device
- destroys the specified context

When this function returns, you can safely release the video capture buffers you have been using with this context.

This function is *not* signal handler safe! We recommend that your application create a separate thread for signal handling. The signal thread can then destroy the capture context by instructing another thread to call *capture_destroy_context()*.

For more information about video capture contexts, see "*Contexts* (p. 19)".

Returns:

n/a

Errors:

n/a

capture_get_frame()

Get a frame from the video capture device.

Synopsis:

```
#include <vcapture/capture.h>

int capture_get_frame( capture_context_t context,
    uint64_t timeout, uint_32_t flags )
```

Arguments:

context

Pointer to the video capture context.

timeout

Wait before timing out. Set to any of:

- The number of nanoseconds the function should wait for a frame before timing out. The function may return in less time than the period specified by this argument.
- `CAPTURE_TIMEOUT_INFINITE` to wait indefinitey for a frame (never time out).
- `CAPTURE_TIMEOUT_NO_WAIT` to return immediately, even if there is no frame.

flags

Flag specifying how to handled queued frames. Set to one of:

- `0` (zero) to retrieve all queued frames in sequence.
- `CAPTURE_FLAG_LATEST_FRAME` to retrieve the latest frame, discarding all the other queued frames.

Library:

`libcapture`

Description:

The function *capture_get_frame()* retrieves frames from the device. If more than one frame is in the queue, depending on the behavior specified by the *flags* argument,

this function will either retrieve all queued frames in sequence or retrieve only the latest frame, dropping the others. The function maintains the dropped frame counter.

The buffer used to get a frame is locked for exclusive use by the client app until *capture_release_frame()* releases it back to the capture driver.

To avoid overwriting a frame before it has been displayed, your application should use *at least* three capture buffers to:

1. Call *capture_get_frame()* to get the index of the captured video frame.
2. Hand the frame buffer over for display by a call to a Screen function such as *screen_post_window()*.
3. Call *capture_get_frame()* to get another frame.
4. When the frame in Step 1 has been displayed, call *capture_release_frame()* to release the buffer for reuse by the capture driver.

Returns:

0

Success: the index of the captured buffer.

-1

An error occurred, or the function has timed out (`errno` is set).

Errors:

ECANCELED

The capture was disabled. This error occurs when one thread calls *capture_get_frame()* while capturing is enabled, then another thread disables capturing before *capture_get_frame()* has finished.

EINVAL

Invalid argument.

EIO

Hardware input or output error.

ENOMEM

Capturing isn't possible because the client application has locked all buffers; it has not released any buffers for use by the capture library.

ETIMEDOUT

The request for a frame has timed out; no frame was captured.

capture_get_free_buffer()

Get a free video capture buffer when bringing in video from an external source, and call the video capture function.

Synopsis:

```
#include <vcapture/capture.h>

int capture_get_free_buffer( capture_context_t
            context, uint64_t
            timeout, (uint_32_t
            flags | CAPTURE_FLAG_FREE_BUFFER )
```

Arguments:

context

Pointer to the video capture context.

timeout

Wait before timing out. Set to any of:

- The number of nanoseconds the function should wait for a buffer before timing out. The function may return in less time than the period specified by this argument.
- `CAPTURE_TIMEOUT_INFINITE` to wait indefinitely for a buffer (never time out).
- `CAPTURE_TIMEOUT_NO_WAIT` to return immediately, even if there is no buffer.

flags

For internal use at this time. Must be set to `0` (zero).

Library:

`libcapture`

Description:

The function *capture_get_free_buffer()* returns the index to a free buffer. After calling this function, the client application should call *capture_put_buffer()* to place the capture buffer in the video capture stream, then use another thread to call *capture_get_frame()* to return a processed frame (for example, the frame is scaled or deinterlaced).

Returns:

0

Success: the index of the captured buffer returned by *capture_put_buffer()*.

-1

An error occurred, or the function has timed out (`errno` is set).

Errors:

EINVAL

Invalid argument.

capture_get_property_i()

Get video capture driver and device properties.

Synopsis:

```
#include <vcapture/capture.h>

int capture_get_property_i( capture_context_t context,
    uint32_t prop, int32_t *value )
```

Arguments:

context

Pointer to the video capture context.

prop

The property to get.

value

A pointer to the location where the retrieved property is located.

Library:

`libcapture`

Description:

The function *capture_get_property_i()* retrieves video capture driver and device properties, which may have been set by *capture_set_property_p()*.

Returns:

0

Success

-1

An error occurred (`errno` is set).

Errors:

ENOENT

This property can't be retrieved by the method used.

ENOTSUP

The driver or the device doesn't support the specified property.

capture_get_property_p()

Get video capture driver and device properties.

Synopsis:

```
#include <vcapture/capture.h>

int capture_get_property_p( capture_context_t context,
    uint32_t prop, int32_t **value )
```

Arguments:

context

Pointer to the video capture context.

prop

The property to get.

value

A reference to the location where the retrieved property will be placed.

Library:

`libcapture`

Description:

The function *capture_get_property_p()* retrieves video capture driver and device properties, and then places them in an array, where they can be read.

Returns:

`0`

Success: the number of filled elements in the array.

`-1`

An error occurred (`errno` is set).

Errors:

ENOENT

This property can't be retrieved by the method used.

ENOTSUP

The driver or the device doesn't support the specified property.

capture_is_property()

Check if the connected video capture device supports a specified property.

Synopsis:

```
#include <vcapture/capture.h>

int capture_is_property(capture_context_t context,
    uint32_t prop)
```

Arguments:

context

Pointer to the video capture context.

prop

The property for which device support is needed.

Library:

`libcapture`

Description:

The *capture_is_property()* function checks if the connected video capture device supports the property specified in *prop*.

Example:

The code snippet below may be a useful reference on how to use *capture_is_property()*.

```
void get_video_info(capture_context_t context)
{
 char *cur_norm = NULL;
 if(capture_is_property(context, CAPTURE_PROPERTY_CURRENT_NORM)) {
  capture_get_property_p(context, CAPTURE_PROPERTY_CURRENT_NORM, (void **)&cur_norm);
 }
 fprintf(stderr, "current norm: %s", cur_norm? cur_norm : "unavailable");
#ifdef CAPTURE_ADV
 int32_t lock = -1, fsclock = -1, freq = -1, wss = -1;
 if(capture_is_property(context, CAPTURE_PROPERTY_ADV_LOCK_STATUS)) {
  capture_get_property_i(context, CAPTURE_PROPERTY_ADV_LOCK_STATUS, &lock);
 }
 if(capture_is_property(context, CAPTURE_PROPERTY_ADV_FSCLOCK_STATUS)) {
  capture_get_property_i(context, CAPTURE_PROPERTY_ADV_FSCLOCK_STATUS, &fsclock);
 }
 if(capture_is_property(context, CAPTURE_PROPERTY_ADV_OUTPUT_FREQ)) {
  capture_get_property_i(context, CAPTURE_PROPERTY_ADV_OUTPUT_FREQ, &freq);
 }
 if(capture_is_property(context, CAPTURE_PROPERTY_ADV_WSS_STATUS)) {
  capture_get_property_i(context, CAPTURE_PROPERTY_ADV_WSS_STATUS, &wss);
```

```
 }
 fprintf(stderr, " lock:%d fsclock: %d freq: %d wss:%d", lock, fsclock, freq, wss);
#endif
 fprintf(stderr, "\n");
}
```

Returns:

1

The device supports the specified property.

0

The device doesn't support the specified property.

capture_put_buffer()

Pass a buffer to the driver for deinterlacing a frame when bringing in video from an external source.

Synopsis:

```
#include <vcapture/capture.h>

int capture_put_buffer( capture_context_t ctx,
    uint32_t idx, uint32_t flags )
```

Arguments:

context

Pointer to the video capture context.

idx

The index to the frame buffer to inject into the capture driver.

flags

Flag specifying how to process the deinterlaced frame.

Library:

`libcapture`

Description:

This function *capture_put_buffer()* passes a buffer to the driver for deinterlacing frames brought in from a video on an external source, such as a USB memory stick. It should be used only when the CAPTURE_FLAG_EXTERNAL_SOURCE flag is set.

Interlaced video frames (typical of analog video) contain two sequential subfields, which doubles the perceived frame rate and improves the video quality. To display interlaced video in a system using a progressive display, the interlaced frames need to be separated into two frames in the correct sequence. Thus, displaying an interlaced video correctly on a progressive display requires two buffers for every interlaced frame. The *capture_put_buffer()* function passes a buffer to the driver, which it can use for the second frame extracted from the interlaced frame.

Returns:

0

Success.

-1

An error occurred (`errno` is set).

Errors:

EINVAL

Invalid index (*idx*) argument, or invalid sequence.

capture_release_frame()

Release a video frame buffer.

Synopsis:

```
#include <vcapture/capture.h>

int capture_release_frame( capture_context_t context,
    uint32_t idx )
```

Arguments:

context

Pointer to the video capture context.

idx

The index to the frame buffer to release.

Library:

`libcapture`

Description:

The function *capture_release_frame()* releases the frame specified in its *idx* argument and returns it to the capture queue. Your application should call this function after it has displayed a captured frame to ensure that the buffer locked by *capture_get_frame()* is made available for reuse.

Returns:

0

Success.

-1

An error occurred (`errno` is set).

Errors:

EINVAL

Invalid index (*idx*) argument.

capture_set_property_i()

Set video frame capture properties.

Synopsis:

```
#include <vcapture/capture.h>

int capture_set_property_i( capture_context_t context,
    uint32_t prop, int32_t value )
```

Arguments:

context

Pointer to the video capture context.

prop

The property to set.

value

The integer value of the property.

Library:

`libcapture`

Description:

The function *capture_set_property_i()* sets video capture properties to pass to the video capture device driver.

Array resources, such as CAPTURE_PROPERTY_FRAME_FLAGS and CAPTURE_PROPERTY_FRAME_SEQNO are not allocated by default. They need to be set, then passed to the video capture driver. To do this you need to:

1. Call *capture_set_property_i()* for each capture property you need to set, using the CAPTURE_PROPERTY_* constants to specify the device, brightness, destination buffers, etc.
2. When you have set all the properties you need to specify, call *capture_set_property_p()* to pass a pointer to this array to the video capture library. The library stores this pointer and will update the array when appropriate.

You can instruct the video capture library to stop collecting and providing data for a property by setting the location for property in the array to NULL.

Returns:

0

Success.

-1

An error occurred (`errno` is set).

Errors:

EINVAL

Bad value or values.

ENOENT

The specified property can't be set by this method.

ENOTSUP

The driver or the device doesn't support the specified property.

EROFS

Unable to change read-only property.

capture_set_property_p()

Set video frame capture properties.

Synopsis:

```
#include <vcapture/capture.h>

int capture_set_property_p( capture_context_t context, uint32_t
 prop, int32_t *value )
```

Arguments:

context

Pointer to the video capture context.

prop

The video capture frame property to set.

****value***

Pointer to the array with the property to set.

Library:

`libcapture`

Description:

The function *capture_set_property_p()* passes the pointer to the array of video capture properties to the video capture library.

Array resources, such as CAPTURE_PROPERTY_FRAME_FLAGS and CAPTURE_PROPERTY_FRAME_SEQNO, are not allocated by default. They need to be set, then passed to the video capture driver. To do this, you must set the properties you need to specify by calling *capture_set_property_i()*, then pass a pointer to the array with these properties by calling *capture_set_property_p()*.

Allocate an array large enough to store the video capture properties. The minimum number of elements in the array must be at least as large as the corresponding capture property associated with the array you are setting. For example, the minimum array sizes of CAPTURE_PROPERTY_FRAME_SEQNO and CAPTURE_PROPERTY_FRAME_FLAGS are determined by the size of CAPTURE_PROPERTY_NBUFFER.

Returns:

0

Success.

-1

An error occurred (`errno` is set).

Errors:

EINVAL

Bad value or values.

ENOENT

The specified property can't be set by this method.

ENOTSUP

The driver or the device doesn't support the specified property.

EROFS

Unable to change read-only property.

capture_update()

Update the video capture device.

Synopsis:

```
#include <vcapture/capture.h>

int capture_update( capture_context_t context,
    uint32_t flags )
```

Arguments:

context

Pointer to the video capture context.

flags

Reserved. Do not use.

value

A pointer to the location where the retrieved property is located.

Library:

`libcapture`

Description:

The function *capture_update()* applies (commits) all updates posted since it was last called to the driver.

Functions such as *capture_set_property_i()* and *capture_set_property_p()* update memory but they don't apply updates to the video capture device. To apply updates, you must call *capture_update()*. This behavior allows the proper combining of discrete properties, such as `*DST_X`, `*DST_Y` and `*DST_WIDTH`. Once these properties are combined, a call to *capture_update()* commits all the changes to the device at the same time.

If you want video capture to start immediately, set `CAPTURE_ENABLE` to `1` (one) when you call *capture_update()*.

Returns:

0

Success

-1

An error occurred (`errno` is set).

Errors:

EINVAL

Some parameters are in conflict; video capture isn't possible.

EIO

Hardware input or output error.

ENODEV

No video capture device with the specified device ID.

The driver may *slog*()* more detailed error information. Use `sloginfo` to retrieve it.

Index

E

F

G

H

I

J

L

M

N

O

P