# Video Capture Developer's Guide

BlackBerry | QNX

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: March 06, 2020**

# 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 | *Using Video Capture* |
| The video capture header files and libraries | *Header files and libraries* |
| The tasks required to capture video | *Implementing video capture* |
| A sample program you can use as a reference | *Sample video capture program* |
| Arrays used for video capture and the properties that can be applied to them | *Properties applied to arrays* |
| Vidoe capture contexts | *Contexts* |
| Dynamically allocated and statically allocated video capture buffers | *Buffers* |
| Configuring your video capture implementation for different platforms | *Platform-specific considerations* |
| The video capture API | *Video Capture API (capture.h)* |

## 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` |
| Constants | NULL |
| Data types | `unsigned short` |
| 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.

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

**DANGER:** 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 typically use a forward slash (/) as a delimiter in 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
# 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 | *Using Video Capture* |
| The video capture header files and libraries | *Header files and libraries* |
| The tasks required to capture video | *Implementing video capture* |
| A sample program you can use as a reference | *Sample video capture program* |
| Arrays used for video capture and the properties that can be applied to them | *Properties applied to arrays* |
| Vidoe capture contexts | *Contexts* |
| Dynamically allocated and statically allocated video capture buffers | *Buffers* |
| Configuring your video capture implementation for different platforms | *Platform-specific considerations* |
| The video capture API | *Video Capture API (capture.h)* |

# Chapter 2
# Using Video Capture

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 API* section of this guide.

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.

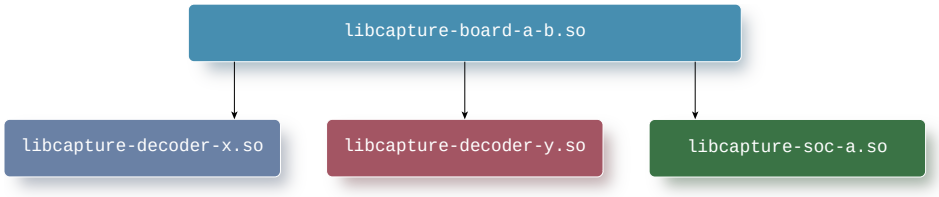


**Figure 1: Overview of video capture libraries**

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* 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:

- 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. Set up your video capture input that's appropriate for your board by calling *capture_set_property_i()* on each of the following properties:
   - CAPTURE_PROPERTY_DEVICE
   - CAPTURE_PROPERTY_SRC_INDEX
2. Call *capture_set_property_i()* for each capture property (of type `integer`) you need to set (e.g., CAPTURE_PROPERTY_DEVICE) to the video capture library.
3. Call *capture_set_property_p()* for each capture property (of type `void` pointer) you need to set (e.g., CAPTURE_PROPERTY_FRAME_FLAGS) 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 don't 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.

4. If your restart video capture (i.e., call *capture_set_property_i()* with the second argument set to >CAPTURE_ENABLE and the third argument set to `1` (one), and then call *capture_update()*) after stopping, then the Video capture framework reclaims all buffers. These buffers include even those that aren't released by the application (i.e., the application hasn't yet called *capture_release_frame()* for these buffers).

## 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...
    }

    // 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 );

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

    capture_set_property_i( context, CAPTURE_PROPERTY_BRIGHTNESS, 10 );
    capture_set_property_i( context, CAPTURE_PROPERTY_FRAME_NBUFFERS, n_pointers );
    capture_set_property_p( context, CAPTURE_PROPERTY_FRAME_BUFFERS, 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.

**DANGER:** 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.)

## 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 (e.g., in the case where the buffer was dynamically allocated)

**DANGER:**

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 3
# Video Capture API (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.

## Definitions:

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

*CAPTURE_PROPERTY()* converts string constants into integers.

This macro is used to help define the capture properties

# Data bus, data lane, and clock

The video capture API includes constants that specify the video capture data bus, data lane and clock properties.

## Definitions:

**`#define CAPTURE_PROPERTY_INTERFACE_TYPE CAPTURE_PROPERTY('Q','P','I','F')`**

Interface type.

Read/Write `uint32_t`

**`#define CAPTURE_PROPERTY_DATA_BUS_WIDTH CAPTURE_PROPERTY( 'Q', 'D', 'B', 'W' )`**

Width of data bus for parallel interfaces.

Valid values are 8, 10, 16, etc.

Read/Write `int`

**`#define CAPTURE_PROPERTY_CSI2_NUM_DATA_LANES CAPTURE_PROPERTY('Q','C','N','D')`**

Number of CSI2 data lanes.

Valid values are 1 to 4, or -1 to default to the number or position that's already set in the hardware.

Read/Write `int`

**`#define CAPTURE_PROPERTY_CSI2_CLK_LANE_POS CAPTURE_PROPERTY('Q','C','C','P')`**

Position of CSI2 clock lane.

Valid values are 1 to 4, or -1 to default to the number or position that's already set in the hardware.

Read/Write `int`

**`#define CAPTURE_PROPERTY_CSI2_DATA0_LANE_POS CAPTURE_PROPERTY('Q','C','D','0')`**

Position of CSI2 data lane 0.

Valid values are 1 to 4, or -1 to default to the number or position that's already set in the hardware.

Read/Write `int`

**`#define CAPTURE_PROPERTY_CSI2_DATA1_LANE_POS CAPTURE_PROPERTY('Q','C','D','1')`**

Position of CSI2 data lane 1.

Valid values are 1 to 4, or -1 to default to the number or position that's already set in the hardware.
Read/Write `int`

**`#define CAPTURE_PROPERTY_CSI2_DATA2_LANE_POS CAPTURE_PROPERTY('Q','C','D','2')`**

Position of CSI2 data lane 2.

Valid values are 1 to 4, or -1 to default to the number or position that's already set in the hardware.

Read/Write `int`

**`#define CAPTURE_PROPERTY_CSI2_DATA3_LANE_POS CAPTURE_PROPERTY('Q','C','D','3')`**

Position of CSI2 data lane 3.

Valid values are 1 to 4, or -1 to default to the number or position that's already set in the hardware.

Read/Write `int`

**`#define CAPTURE_PROPERTY_CSI2_VCHANNEL_NUM CAPTURE_PROPERTY('Q','C','V','N')`**

CSI2 Virtual Channel Identifier number.

Valid values are 0 to 3, or -1 to default to the virtual channel indentifier number that's already set in the hardware.

Read/Write `int`

**`#define CAPTURE_PROPERTY_CSI2_DDR_CLOCK_MHZ CAPTURE_PROPERTY('Q','C','D','C')`**

CSI2 dual date rate (DDR) clock frequency in MHz.

Used on the CSI2 receiver side, to match the DDR clock of the transmitter.

Read/Write `int`

# Debugging

The video capture API includes constants that specify the video capture debugging properties.

### Definitions:

**`#define CAPTURE_PROPERTY_VERBOSITY CAPTURE_PROPERTY( 'Q', 'V', 'B', 'R' )`**

Log verbosity level.

Default is 0; increase this value to increase log verbosity for debugging.

Read/Write `uint32_t`

**`#define CAPTURE_PROPERTY_VERBOSITY_FILE CAPTURE_PROPERTY('Q','V','B','F')`**

File log messages are written to.

Default is `stderr`; A NULL pointer disables logging.

Read/Write `FILE*`

**`#define CAPTURE_PROPERTY_LOGMSG_PREFIX CAPTURE_PROPERTY('Q','L','M','P')`**

User-defined string that log messages will be prefixed with.

Read/Write `char*`

## Deinterlacing

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.

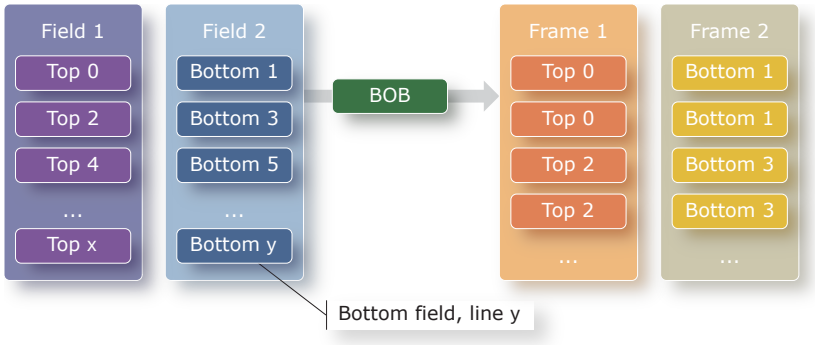


**Figure 2: Deinterlacing in BOB mode**

# capture_deinterlace_mode

*Types of deinterlacing modes*

## Synopsis:

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

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
};
```

## Data:

**CAPTURE_DEINTERLACE_NONE_MODE**

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.

## Library:

**libcapture**

# Deinterlacing

The video capture API includes constants that specify the video capture deinterlacing properties.

### Definitions:

**`#define CAPTURE_PROPERTY_MIN_NBUFFERS CAPTURE_PROPERTY( 'Q', 'M', 'N', 'B' )`**

Minimum number of buffers required for a specific deinterlacing mode.

Read `uint32_t`

# Destination buffer

The video capture API includes constants that specify the video capture destination buffer properties.

### Definitions:

**`#define CAPTURE_PROPERTY_DST_COLOR_SPACE CAPTURE_PROPERTY( 'Q', 'N', 'C', 'D'`**

Set default destination color space and values range of the color space.

Read/Write `uint32_t`

**`#define CAPTURE_PROPERTY_DST_WIDTH CAPTURE_PROPERTY( 'Q', 'D', 'F', 'W' )`**

Width of destination frame, in pixels.

Read/Write `uint32_t`

**`#define CAPTURE_PROPERTY_DST_HEIGHT CAPTURE_PROPERTY( 'Q', 'D', 'F', 'H' )`**

Height of destination frame buffer, in pixels.

Read/Write `uint32_t`

**`#define CAPTURE_PROPERTY_DST_STRIDE CAPTURE_PROPERTY( 'Q', 'D', 'F', 'S' )`**

Stride of destination frame buffer, in bytes.

Read/Write `uint32_t`

**`#define CAPTURE_PROPERTY_DST_NBYTES CAPTURE_PROPERTY( 'Q', 'D', 'F', 'B' )`**

Size of destination frame buffer, in bytes.

For planar YUV formats, the size of each allocated destination buffer must be large enough for all the planes.

For other formats, 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* x *CAPTURE_PROPERTY_DST_HEIGHT*).

Read/Write `uint32_t`

**`#define CAPTURE_PROPERTY_DST_FORMAT CAPTURE_PROPERTY( 'Q', 'D', 'F', 'F' )`**

Destination buffer format.

When setting `CAPTURE_PROPERTY_SRC_FORMAT`, the valid values for this property are defined by Screen pixel format types in `screen.h`.

Read/Write `[] uint32_t`

**`#define CAPTURE_PROPERTY_PLANAR_OFFSETS CAPTURE_PROPERTY( 'Q', 'P', 'L', 'O' )`**

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

Read/Write `int`

## Driver and device

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

# capture_no_signal_mode

*Types of no signal modes*

## Synopsis:

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

enum capture_no_signal_mode {
    CAPTURE_NO_SIGNAL_DEFAULT_MODE = 0,
    CAPTURE_NO_SIGNAL_NOISE_MODE,
    CAPTURE_NO_SIGNAL_BLACK_MODE,
    CAPTURE_NO_SIGNAL_BLUE_MODE,
    CAPTURE_NO_SIGNAL_TABLE_MODE
};
```

## Data:

**CAPTURE_NO_SIGNAL_DEFAULT_MODE**

Default behavior; defined by driver or hardware (unknown).

**CAPTURE_NO_SIGNAL_NOISE_MODE**

Show color or black-and-white noise (snow) on the display if there isn't an input video source.

This is a common mode for hardware that doesn't support detection of video source signals

**CAPTURE_NO_SIGNAL_BLACK_MODE**

Fill all frames with black (or close to black) color if no video source signal state is detected.

**CAPTURE_NO_SIGNAL_BLUE_MODE**

Fill all frames with blue (or close to blue) color if no video source signal state is detected.

**CAPTURE_NO_SIGNAL_TABLE_MODE**

Use a hardware or software pre-defined picture (usually with text "NO SIGNAL") if no video source signal state is detected.

## Library:

**libcapture**

# Driver and device

The video capture API includes constants that specify the video capture driver and device properties.

### Definitions:

**#define CAPTURE_PROPERTY_DEVICE_INFO CAPTURE_PROPERTY( 'Q', 'I', 'N', 'F' )**

Return string information about the video capture driver and device; all drivers support this property.

The content of this property depends on the device that's set in *CAPTURE_PROPERTY_DEVICE* property.

Read `const char *`

**#define CAPTURE_ENABLE CAPTURE_PROPERTY( 'Q', 'E', 'N', 'A' )**

Capture start(1) and stop(0)

Read/Write `uint32_t`

**#define CAPTURE_PROPERTY_NDEVICES CAPTURE_PROPERTY( 'Q', 'N', 'D', 'V' )**

Number of supported capture units A capture driver can support a number of devices (*CAPTURE_PROPERTY_NDEVICES*); each device has at least one input (*CAPTURE_PROPERTY_NSOURCES*).

Selection of the input is done by choosing a device, and then a source.

Read `uint32_t`

**#define CAPTURE_PROPERTY_DEVICE CAPTURE_PROPERTY( 'Q', 'D', 'E', 'V' )**

Capture device in this context.

Once set, on some drivers, this property cannot be changed.

Read/Write `uint32_t`

**#define CAPTURE_PROPERTY_NSOURCES CAPTURE_PROPERTY( 'Q', 'N', 'S', 'R' )**

Number of source inputs available after the device is set.

Read `uint32_t`

**#define CAPTURE_PROPERTY_CONTRAST CAPTURE_PROPERTY( 'Q', 'C', 'O', 'N' )**

Contrast (`-128` to `127`)

Read/Write `int32_t`

**#define CAPTURE_PROPERTY_BRIGHTNESS CAPTURE_PROPERTY( 'Q', 'B', 'R', 'I' )**

Brightness (`-128` to `127`)

Read/Write `int32_t`

**#define CAPTURE_PROPERTY_SATURATION CAPTURE_PROPERTY( 'Q', 'S', 'A', 'T' )**

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

Read/Write `int32_t`

**#define CAPTURE_PROPERTY_HUE CAPTURE_PROPERTY( 'Q', 'H', 'U', 'E' )**

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

Read/Write `int32_t`

**#define CAPTURE_PROPERTY_SHARPNESS CAPTURE_PROPERTY( 'Q', 'S', 'R', 'P' )**

Sharpness (`-128` to `127`); where an image is blurred at `-128` to the sharpest at `127`.

Read/Write `int32_t`

**#define CAPTURE_PROPERTY_DEINTERLACE_FLAGS CAPTURE_PROPERTY( 'Q', 'D', 'E', 'I**

Deinterlacing flags (bit-field)

Use *CAPTURE_DEI_WEAVE_FLAG_BOTTOM_TOP*

Read/Write `uint32_t`

**#define CAPTURE_DEI_WEAVE_FLAG_BOTTOM_TOP (1 << 0)**

Used to specify the field order for weave deinterlacing.

When set, the weave order is to always weave bottom + top, possibly dropping an unexpected first top field. When not set, the weave order is to always weave top + bottom, possibly dropping an unexpected first bottom field.

**#define CAPTURE_PROPERTY_DEINTERLACE_MODE CAPTURE_PROPERTY( 'Q', 'D', 'E', 'M'**

Deinterlacing mode; refer to *Deinterlacing mode types*.

Read/Write `uint32_t`

**#define CAPTURE_PROPERTY_NO_SIGNAL_MODE CAPTURE_PROPERTY( 'Q', 'N', 'O', 'S' )**

No signal mode; set default action if "no carrier" or "no signal" state is detected.

Refer to *No signal mode types*. Read/Write @c uint32_t

**#define CAPTURE_PROPERTY_SRC_INDEX CAPTURE_PROPERTY( 'Q', 'S', 'I', 'D' )**

Video capture unit of the device.

Read/Write `uint32_t`

# External source

The video capture API includes constants that specify the video capture external source properties.

## Definitions:

**#define CAPTURE_FLAG_EXTERNAL_SOURCE UINT32_C(0x0002)**

Bit to set if the context to create is for an external source, instead of a captured source.

The video capture API includes constants for managing the retrieval and deinterlacing of frames brought in from an external source, such as a USB memory stick. When you've set this constant, the following properties are relevant:

- *CAPTURE_BUFFER_USAGE_RDONLY*
- *CAPTURE_FLAG_FREE_BUFFER*
- *CAPTURE_PROPERTY_BUFFER_FLAGS*
- *CAPTURE_PROPERTY_BUFFER_INDEX*
- *CAPTURE_PROPERTY_BUFFER_NFIELDS*
- *CAPTURE_PROPERTY_BUFFER_PLANAR_OFFSETS*
- *CAPTURE_PROPERTY_BUFFER_SEQNO*
- *CAPTURE_PROPERTY_BUFFER_USAGE*

**#define CAPTURE_FLAG_FREE_BUFFER UINT32_C(0x0002)**

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

**#define CAPTURE_BUFFER_USAGE_RDONLY 0x001**

Mark the buffer as read-only.

**#define CAPTURE_PROPERTY_BUFFER_USAGE CAPTURE_PROPERTY( 'Q', 'B', 'U', 'S' )**

An array of buffer usage flags.

Element `i` indicates if the capture driver has read-only or read/write permission for buffer `i`. The default is read/write permission.

Read/Write `[] uint32_t`

**#define CAPTURE_PROPERTY_BUFFER_INDEX CAPTURE_PROPERTY( 'Q', 'B', 'I', 'X' )**

Index of the buffer to be injected by *capture_put_buffer()*

Write `uint32_t`

**#define CAPTURE_PROPERTY_BUFFER_NFIELDS CAPTURE_PROPERTY( 'Q', 'B', 'N', 'F' )**

Number of fields contained in the buffer to be injected by *capture_put_buffer()*

Write `uint32_t`

**#define CAPTURE_PROPERTY_BUFFER_PLANAR_OFFSETS CAPTURE_PROPERTY( 'Q', 'B', 'P'**

A per-buffer array.

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

This property is used for external sources. The library than handles your hardware copies the data for the current frame rather than storing a pointer to the array.

*CAPTURE_PROPERTY_BUFFER_PLANAR_OFFSETS* has a dependency on:

- *CAPTURE_PROPERTY_BUFFER_INDEX*
- *CAPTURE_PROPERTY_BUFFER_NFIELDS*

Write `[] [3] int32_t`

**#define CAPTURE_PROPERTY_BUFFER_FLAGS CAPTURE_PROPERTY( 'Q', 'B', 'F', 'L' )**

A per-buffer array of the following buffer flags:

- *CAPTURE_FRAME_FLAG_ERROR*
- *CAPTURE_FRAME_FLAG_INTERLACED*
- *CAPTURE_FRAME_FLAG_FIELD_BOTTOM*

This property is used for external sources. The library than handles your hardware copies the data for the current frame rather than storing a pointer to the array.

*CAPTURE_PROPERTY_BUFFER_FLAGS* has a dependency on:

- *CAPTURE_PROPERTY_BUFFER_INDEX*
- *CAPTURE_PROPERTY_BUFFER_NFIELDS*

Write `[] uint32_t`

**#define CAPTURE_PROPERTY_BUFFER_SEQNO CAPTURE_PROPERTY( 'Q', 'B', 'S', 'N' )**

A per-buffer array of sequence numbers.

Each element indicates the sequence number of the field contained in the buffer.

This property is used for external sources. The library than handles your hardware copies the data for the current frame rather than storing a pointer to the array.

*CAPTURE_PROPERTY_BUFFER_SEQNO* has a dependency on:

- *CAPTURE_PROPERTY_BUFFER_INDEX*
- *CAPTURE_PROPERTY_BUFFER_NFIELDS*

Write `[] uint32_t`

## Interface, threads, and offsets

The video capture API includes constants that specify the video capture interface type, thread priority, and YUV offsets.

# capture_iface_type

*Types of interfaces*

## Synopsis:

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

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

## Data:

**CAPTURE_IF_PARALLEL**

The interface is parallel.

**CAPTURE_IF_MIPI_CSI2**

The interface is a MIPI CSI2 interface.

## Library:

**libcapture**

# Threads

The video capture API includes constants that specify thread priority.

### Definitions:

**`#define CAPTURE_PROPERTY_THREAD_PRIORITY CAPTURE_PROPERTY( 'Q', 'T', 'P', 'R'`**

Scheduling priority of the capture thread.

Read/Write `int`

# I2C decoder path and slave address

The video capture API includes constants that specify the video capture I2C decoder path and slave address properties.

## Definitions:

**`#define CAPTURE_PROPERTY_DECODER_I2C_PATH CAPTURE_PROPERTY('Q','D','I','P')`**

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

This property is only applicable to decoders that are connected via I2C.

Read/Write `const char *`

**`#define CAPTURE_PROPERTY_DECODER_I2C_ADDR CAPTURE_PROPERTY('Q','D','I','A')`**

The 7-bit Slave address of the I2C decoder.

This property is only applicable to decoders that are connected via I2C.

Read/Write `const char *`

**`#define CAPTURE_PROPERTY_USB_DEVICE_PATH CAPTURE_PROPERTY('Q','U','D','P')`**

Device path of the USB resource manager.

This property is only applicable to cameras that are connected via USB.

Read/Write `const char *`

# Polarity

The video capture API includes constants that specify the video capture polarity properties.

## Definitions:

**#define CAPTURE_PROPERTY_INVERT_FID_POL CAPTURE_PROPERTY( 'Q', 'L', 'F', 'I' )**

Specifies whether the field ID signal polarity is inverted.

- 0: Set polarity to not inverted
- 1: Set polarity to inverted
- -1: Don't set the polarity; use whatever polarity is already set

Read/Write `int`

**#define CAPTURE_PROPERTY_INVERT_VSYNC_POL CAPTURE_PROPERTY( 'Q', 'L', 'H', 'S'**

Specifies whether the vertical synchronization polarity is inverted.

- 0: Set polarity to not inverted
- 1: Set polarity to inverted
- -1: Don't set the polarity; use whatever polarity is already set

Read/Write `int`

**#define CAPTURE_PROPERTY_INVERT_HSYNC_POL CAPTURE_PROPERTY( 'Q', 'L', 'V', 'S'**

Specifies whether the horizontal synchronization polarity is inverted.

- 0: Set polarity to not inverted
- 1: Set polarity to inverted
- -1: Don't set the polarity; use whatever polarity is already set

Read/Write `int`

**#define CAPTURE_PROPERTY_INVERT_CLOCK_POL CAPTURE_PROPERTY( 'Q', 'L', 'P', 'C'**

Specifies whether the clock polarity is inverted.

- 0: Set polarity to not inverted
- 1: Set polarity to inverted
- -1: Don't set the polarity; use whatever polarity is already set

Read/Write `int`

**#define CAPTURE_PROPERTY_INVERT_DATAEN_POL CAPTURE_PROPERTY( 'Q', 'L', 'D', 'E**

Specifies whether the "data_en" pin/signal polarity is inverted.

- 0: Set polarity to not inverted
- 1: Set polarity to inverted
- -1: Don't set the polarity; use whatever polarity is already set

Read/Write `int`

**#define CAPTURE_PROPERTY_INVERT_DATA_POL CAPTURE_PROPERTY( 'Q', 'L', 'D', 'A'**

Specifies whether the data input polarity is inverted.

- 0: Set polarity to not inverted
- 1: Set polarity to inverted
- -1: Don't set the polarity; use whatever polarity is already set

Read/Write `int`

## Source

The video capture API includes definitions for the source properties.

# capture_color_space

*Types of color spaces*

## Synopsis:

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

enum capture_color_space {
    CAPTURE_COLOR_SPACE_BT601 = 0,
    CAPTURE_COLOR_SPACE_BT709,
    CAPTURE_COLOR_SPACE_BT2020,
    CAPTURE_COLOR_SPACE_LRGB,
    CAPTURE_COLOR_SPACE_SRGB,
    CAPTURE_COLOR_SPACE_ADOBERGB
};
```

## Data:

**CAPTURE_COLOR_SPACE_BT601**

Use BT.601 (SDTV) defined primaries.

**CAPTURE_COLOR_SPACE_BT709**

Use BT.709 (HDTV) defined primaries.

**CAPTURE_COLOR_SPACE_BT2020**

Use BT.2020 (UHDTV) defined primaries.

**CAPTURE_COLOR_SPACE_LRGB**

Linear RGB color space.

**CAPTURE_COLOR_SPACE_SRGB**

sRGB color space, use BT.709 (HDTV) defined primaries.

**CAPTURE_COLOR_SPACE_ADOBERGB**

Adobe RGB color space, similar to sRGB but with extended gamut range.

## Library:

**libcapture**

# Source

The video capture API includes constants that specify the video capture source properties.

### Definitions:

**`#define CAPTURE_PROPERTY_SRC_COLOR_SPACE CAPTURE_PROPERTY( 'Q', 'N', 'C', 'S'`**

A bit field that indicates the color space supported by the capture device.

Set this property according to the valid color space types as defined by *Color space types*. The color space types may be used with bitwise operators and the following flags to define *CAPTURE_PROPERTY_SRC_COLOR_SPACE*:

- *CAPTURE_FLAG_COLOR_SPACE_LIMITED*
- *CAPTURE_FLAG_COLOR_SPACE_YEXTEND*
- *CAPTURE_FLAG_COLOR_SPACE_FULL*
- *CAPTURE_FLAG_COLOR_SPACE_MASK*

Read/Write `uint32_t`

**`#define CAPTURE_FLAG_COLOR_SPACE_LIMITED (0x00000000)`**

Use the following ranges for the specified color spaces:

- RGB: 16..235 RGB range
- YUV: 16..235 range for Y and 16..240 range for U and V

**`#define CAPTURE_FLAG_COLOR_SPACE_YEXTEND (0x01000000)`**

Use the following ranges for the specified color spaces:

- RGB: 16..235 RGB range, works as CAPTURE_FLAG_COLOR_SPACE_LIMITED for RGB space.
- YUV: 0..255 range for Y and 16..240 range for U and V

**`#define CAPTURE_FLAG_COLOR_SPACE_FULL (0x80000000)`**

Use the following ranges for the specified color spaces:

- RGB: 0..255 extended RGB range
- YUV: 0..255 extended YUV range for all components

For a 10/12 bits per color component, use full range of 0..1023/0..4095.

**`#define CAPTURE_FLAG_COLOR_SPACE_MASK (0xFF000000)`**

Mask to extract flags from color space data.

**`#define CAPTURE_PROPERTY_SRC_FORMAT CAPTURE_PROPERTY( 'Q', 'S', 'F', 'O' )`**

Source format; a the format of the stream of pixels from the video source.

When setting CAPTURE_PROPERTY_SRC_FORMAT, some valid values for this property are defined by Screen pixel format types in screen.h. If CAPTURE_PROPERTY_SRC_FORMAT needs additional formats beyond those in screen.h, then they must be defined in such a way that they do not conflict with the definitions of the Screen pixel format types.

Read/Write uint32_t

**#define CAPTURE_PROPERTY_SRC_STRIDE CAPTURE_PROPERTY( 'Q', 'S', 'F', 'S' )**

Source buffer stride, in bytes.

Read/Write uint32_t

**#define CAPTURE_PROPERTY_SRC_WIDTH CAPTURE_PROPERTY( 'Q', 'S', 'W', 'I' )**

Width of the source, in pixels.

Read/Write uint32_t

**#define CAPTURE_PROPERTY_SRC_HEIGHT CAPTURE_PROPERTY( 'Q', 'S', 'H', 'E' )**

Height of the source, in pixels.

Read/Write uint32_t

**#define CAPTURE_PROPERTY_CROP_WIDTH CAPTURE_PROPERTY( 'Q', 'C', 'W', 'I' )**

Width of source viewport, in pixels.

Read/Write uint32_t

**#define CAPTURE_PROPERTY_CROP_HEIGHT CAPTURE_PROPERTY( 'Q', 'C', 'H', 'E' )**

Height of source viewport, in pixels.

Read/Write uint32_t

**#define CAPTURE_PROPERTY_CROP_X CAPTURE_PROPERTY( 'Q', 'C', 'X', 'P' )**

Source viewport x offset.

Read/Write uint32_t

**#define CAPTURE_PROPERTY_CROP_Y CAPTURE_PROPERTY( 'Q', 'C', 'Y', 'P' )**

Source viewport y offset.

Read/Write uint32_t

# Video capture behavior

The video capture API includes constants that specify the video capture video capture behavior properties.

## Definitions:

**`#define CAPTURE_TIMEOUT_INFINITE (-1ULL)`**

Never time out; wait for frame indefinitely Typically, drivers set the timeout to 30 days instead of using an actual infinite timeout.

**`#define CAPTURE_TIMEOUT_NO_WAIT (0)`**

Return immediately, even if there is no frame.

**`#define CAPTURE_FLAG_LATEST_FRAME UINT32_C(0x0001)`**

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

# Video standards

The video capture API includes constants that specify the video capture video standard properties.

## Definitions:

**#define CAPTURE_FLAG_ALLOW_UNBLOCK UINT32_C(0x0004)**

Indicate `capture_get_frame` should return immediately (usually with an EINTR error) when *CAPTURE_PROPERTY_UNBLOCK* is set.

Normally this would be used to watch for device-specific events. Verify that `CAPTURE_PROPERTY_UNBLOCK` is supported before using the flag.

**#define CAPTURE_PROPERTY_NORM CAPTURE_PROPERTY( 'Q', 'N', 'O', 'R' )**

Set the video standard.

Refer to *Video standard macros*.

Read/Write `const char *`

**#define CAPTURE_PROPERTY_UNBLOCK CAPTURE_PROPERTY( 'Q', 'U', 'B', 'L' )**

When written, unblock any active capture_get_frame call; if not blocked in this function, cause the next capture_get_frame call to return immediately.

Usually EINTR would be returned, or ESRCH if the capture driver is known to be in an unrecoverable state.

Write `uint32_t` (0, or a device-specific value)

**#define CAPTURE_NORM_AUTO "AUTO"**

Use auto-detection to get the video standard.

Read/Write `const char *`

**#define CAPTURE_PROPERTY_CURRENT_NORM CAPTURE_PROPERTY( 'Q', 'Q', 'N', 'M' )**

Return the current detected video standard.

Refer to *Video standard macros*.

Read `const char *`

**#define CAPTURE_NORM_NONE "NONE"**

Video standard can't be detected; either there's no input, or the video signal is lost.

**#define CAPTURE_NORM_UNKNOWN "UNKNOWN"**

Detected standard is not known.

**#define CAPTURE_NORM_NTSC_M_J "NTSC_M_J"**

The video standard macros are common for *CAPTURE_PROPERTY_NORM* and *CAPTURE_PROPERTY_CURRENT_NORM*.

**#define CAPTURE_NORM_NTSC_4_43 "NTSC_4_43"**

A pseudo-color system that transmits NTSC encoding (not a broadcast format)

**#define CAPTURE_NORM_PAL_M "PAL_M"**

PAL format that uses 525 lines and 59.94 fields per second; this video standard is used in Brazil.

**#define CAPTURE_NORM_PAL_B_G_H_I_D "PAL_B_G_H_I_D"**

PAL format using 625 lines and 50 fields per second with various signal characteristics and color encodings.

**#define CAPTURE_NORM_PAL_COMBINATION_N "PAL_COMBINATION_N"**

PAL format with narrow bandwidth that's used in Argentina, Paraguay, and Uruguay.

**#define CAPTURE_NORM_PAL_60 "PAL_60"**

Multi-system PAL support that uses 525 lines and 60 fields per second (not a broadcast format)

**#define CAPTURE_NORM_SECAM "SECAM"**

Video standard used mainly in France.

**#define CAPTURE_NORM_CEA_640X480P_60HZ "640x480p@60"**

VIC Format 1, based on CEA-861-E specification.

**#define CAPTURE_NORM_CEA_720X480P_60HZ "720x480p@60"**

VIC Format 2, based on CEA-861-E specification.

**#define CAPTURE_NORM_CEA_1280X720P_60HZ "1280x720p@60"**

VIC Format 4, based on CEA-861-E specification.

**#define CAPTURE_NORM_CEA_1920X1080I_60HZ "1920x1080i@60"**

VIC Format 5, based on CEA-861-E specification.

**#define CAPTURE_NORM_CEA_720X480I_60HZ "720x480i@60"**

VIC Format 6, based on CEA-861-E specification.

**#define CAPTURE_NORM_CEA_1920X1080P_60HZ "1920x1080p@60"**

VIC Format 16, based on CEA-861-E specification.

**#define CAPTURE_NORM_CEA_720X576P_50HZ "720x576p@50"**

VIC Format 17, based on CEA-861-E specification.

**#define CAPTURE_NORM_CEA_1920X1080P_50HZ "1920x1080p@50"**

VIC Format 20, based on CEA-861-E specification.

**#define CAPTURE_NORM_CEA_720X576I_50HZ "720x576i@50"**

VIC Format 21, based on CEA-861-E specification.

**#define CAPTURE_NORM_CEA_1920X1080P_24HZ "1920x1080p@24"**

VIC Format 32, based on CEA-861-E specification.

**#define CAPTURE_NORM_CEA_1920X1080P_25HZ "1920x1080p@25"**

VIC Format 33, based on CEA-861-E specification.

**#define CAPTURE_NORM_CEA_1920X1080P_30HZ "1920x1080p@30"**

VIC Format 34, based on CEA-861-E specification.

# Video frame

The video capture API includes constants that specify the video capture video frame properties.

## Definitions:

**`#define CAPTURE_PROPERTY_FRAME_NBUFFERS CAPTURE_PROPERTY( 'Q', 'F', 'B', 'N' )`**

Number of frame buffers that have been specified in *CAPTURE_PROPERTY_FRAME_BUFFERS*.

Read/Write `uint32_t`

**`#define CAPTURE_PROPERTY_FRAME_BUFFERS CAPTURE_PROPERTY( 'Q', 'F', 'B', 'A' )`**

Pointers to the video capture buffers.

Read/Write `[] void*`

**`#define CAPTURE_PROPERTY_FRAME_TIMESTAMP CAPTURE_PROPERTY( 'Q', 'F', 'B', 'T'`**

An array of `CLOCK_MONOTONIC` timestamps indexed by the buffer index.

Read `[] uint64_t`

**`#define CAPTURE_PROPERTY_FRAME_TIMECODE CAPTURE_PROPERTY( 'Q', 'F', 'B', 'C' )`**

An array of SMPTE (Society of Motion Picture and Television Engineers) timecodes indexed by the buffer index; used for streams with metadata.

**`#define CAPTURE_PROPERTY_FRAME_SEQNO CAPTURE_PROPERTY( 'Q', 'F', 'B', 'S' )`**

An array of frame sequence numbers indexed by the buffer index.

Read `[] uint32_t`

**`#define CAPTURE_PROPERTY_FRAME_FLAGS CAPTURE_PROPERTY( 'Q', 'F', 'B', 'F' )`**

An array of frame flags indexed by the buffer index.

Refer to

- *CAPTURE_FRAME_FLAG_ERROR*
- *CAPTURE_FRAME_FLAG_INTERLACED*
- *CAPTURE_FRAME_FLAG_FIELD_BOTTOM*

Read `[] uint32_t`

**`#define CAPTURE_PROPERTY_FRAME_NBYTES CAPTURE_PROPERTY( 'Q', 'F', 'B', 'B' )`**

An array of frame sizes, in bytes, indexed by the buffer index.

Read/Write `[] uint32_t`

**#define CAPTURE_FRAME_FLAG_ERROR 0x0001**

There is an error in the frame.

**#define CAPTURE_FRAME_FLAG_INTERLACED 0x0002**

Frame is interlaced.

For more information about interlacing, refer to *Deinterlacing mode types*.

**#define CAPTURE_FRAME_FLAG_FIELD_BOTTOM 0x0004**

Indicator of the video frame field received.

When the video uses interlacing (i.e., *CAPTURE_FRAME_FLAG_INTERLACED* is set), then when the bottom field (even line numbers) is received, *CAPTURE_FRAME_FLAG_FIELD_BOTTOM* is set. When the top field (odd line numbers) is received, *CAPTURE_FRAME_FLAG_FIELD_BOTTOM* is unset.

**#define CAPTURE_FRAME_FLAG_SHORT 0x0008**

Indicator that the frame or field is shorter than expected.

If the frame or field is shorter because of errors in transmission, the flag *CAPTURE_FRAME_FLAG_ERROR* is also set. Otherwise, a shorter than expected frame of field can be due to either of the following:

- an incorrect selection of analog broadcast standard (e.g., the hardware is expecting a PAL 576i signal, but an NTSC 480i is used instead)
- an incomplete frame

# capture_context_t

*An opaque pointer (or handle) to a video capture context*

## Synopsis:

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

typedef struct _capture_context* capture_context_t;
```

## Library:

**libcapture**

# capture_create_buffers()

*Allocate driver-controlled memory for video capture*

## Synopsis:

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

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

## Arguments:

***context***

The pointer to the video capture context.

***property***

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

## Library:

**libcapture**

## Description:

This function allocates memory for video capture. Allocation of memory for driver allocated buffer memory differs based on whether your hardware supports dynamic buffer allocation. If your hardware doesn't support dynamically-allocated memory, calling `capture_create_buffer()` will fail if *CAPTURE_PROPERTY_FRAME_BUFFERS* is set to anything other than NULL. Refer to *Buffers* from the Video Capture Developer's Guide for more information.

Calls to *capture_create_buffers()* are synchronous. The function frees old buffers and creates the new buffers.

To free an existing buffer without creating a new one:

1. Call *capture_set_property_p()* to set the buffer property *CAPTURE_PROPERTY_FRAME_BUFFERS* to NULL.
2. Call *capture_update()*.

The following code snippet can be 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_buf
...
```

Call this function only when video capture is not in progress.

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 non-existent buffer, with unpredictable results.
- If the buffer in question was initially application-allocated, then no ill effects occur.

## Returns:

0 if successful; or -1 if an error occurred (`errno` is set; refer to `errno.h` for more details). The return value and the value that's observed for `errno` may vary as they are dependent on the implementation by the library that handles your hardware. For example, most drivers return an `errno` value of `ENOTSUP` if they do not support this function.

# capture_create_context()

*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:

This function:

- 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 device ID and input source ID (e.g., device 1, input source 2), and optional settings, such as brightness.

You can create multiple contexts, 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()*.

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

## Returns:

A pointer to a new context if successful; or `NULL` if an error occurred (`errno` is set; refer to `errno.h` for more details). The value that's observed for `errno` may vary as it is hardware dependent.

# capture_destroy_context()

*Disconnect from the video capture device, and destroy the video capture 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:

This function:

- 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! You must create a separate clean-up thread for signal handling. This signal-handling thread can destroy the capture context by instructing another thread to call *capture_destroy_context()*.

## Returns:

Nothing.

# 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,
                      uint32_t flags)
```

## Arguments:

***context***

The pointer to the video capture context.

***timeout***

The 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 indefinitely for a frame (never time out).
- *CAPTURE_TIMEOUT_NO_WAIT* to return immediately, even if there is no frame.

***flags***

A bitmask to change the behavior of this function. Set to 0 or some combination of the following:

- *CAPTURE_FLAG_LATEST_FRAME* to retrieve the latest frame, discarding all the other queued frames (the default behavior is to retrieve all queued frames in sequence).
- *CAPTURE_FLAG_FREE_BUFFER* to retrieve a free buffer instead of a video frame. The *capture_get_free_buffer()* macro can be used to do the same thing.
- *CAPTURE_FLAG_ALLOW_UNBLOCK* to indicate the call should return immediately when some device-specific event occurs, normally with errno set to EINTR.

## Library:

**libcapture**

## Description:

This function 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 the first queued frame in sequence or retrieve only the latest frame, dropping the others.

The index returned by this function corresponds to the sequence number of the *CAPTURE_PROPERTY_FRAME_SEQNO* property, if this array is provided. The application can use this property and the index returned by this function to calculate the number of frames that have been dropped since the last call to *capture_get_frame()*.

It's safe to call this function even if capturing hasn't started (i.e., you haven't set the *CAPTURE_ENABLE* property and called *capture_update()*) yet. Your application must ensure that another thread will enable and disable capturing when required.

The buffer used to get a frame is locked for exclusive use by your application 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 is no longer being displayed, call *capture_release_frame()* to release the buffer for reuse by the capture driver.

## Returns:

The index of the captured buffer (*CAPTURE_PROPERTY_FRAME_BUFFERS*) if successful; or -1 if an error occurred (`errno` is set; refer to `errno.h` for more details). The return value and the value that's observed for `errno` may vary as they are dependent on the implementation by the library that handles your hardware.

# capture_get_free_buffer()

*Get a free video capture buffer when bringing in video from an external source*

**Synopsis:**

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

   #define capture_get_free_buffer( context,

                                    timeout,

                                    flags ) \
                   capture_get_frame( context,

                                    timeout,

                                    (flags) | CAPTURE_FLAG_FREE_BUFFER )
```

## Library:

`libcapture`

## Description:

This macro returns the index to a free buffer. Once your application populates the buffer and sets the appropriate capture properties, the client application calls *capture_put_buffer()* to place the capture buffer in the video capture stream. Use another thread to call *capture_get_frame())* to return a processed frame (for example, the frame is scaled or deinterlaced).

Refer to *capture_get_frame()* for full description for this function.

# capture_get_property_i()

*Get a video frame capture property of type integer*

## Synopsis:

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

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

## Arguments:

***context***

The 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:

This function retrieves a video capture driver or device property, which may have been set by *capture_set_property_p()*.

## Returns:

0 if successful; or -1 if an error occurred (`errno` is set; refer to `errno.h` for more details). The return value and the value that's observed for `errno` may vary as they are dependent on the implementation by the library that handles your hardware.

# capture_get_property_p()

*Get video frame capture property of type void**

## Synopsis:

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

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

## Arguments:

***context***

The pointer to the video capture context.

***prop***

The property to retrieve.

***value***

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

## Library:

**libcapture**

## Description:

This function retrieves a video capture driver or device property.

## Returns:

0 if successful; or -1 if an error occurred (`errno` is set; refer to `errno.h` for more details). The return value and the value that's observed for `errno` may vary as they are dependent on the implementation by the library that handles your hardware.

# capture_is_property()

*Check if the specified property is supported*

## Synopsis:

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

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

## Arguments:

***context***

The pointer to the video capture context.

***prop***

The property that is in question of whether or not it's supported.

## Library:

**libcapture**

## Description:

This function checks if the connected video capture device or driver supports the property specified in the argument *prop*.

Use the following code snippet as reference of 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_no
  }
  fprintf(stderr, "current norm: %s", cur_norm? cur_norm : "unavailable");
  fprintf(stderr, "\n");
}
```

## Returns:

1 if the specified property is supported; or 0 otherwise.

# 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:

***ctx***

The 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 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 fields, 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 fields in the correct sequence. Thus, displaying an interlaced video correctly on a progressive display requires at least 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 if successful; or -1 if an error occurred (`errno` is set; refer to `errno.h` for more details). The return value and the value that's observed for `errno` may vary as they are dependent on the implementation by the library that handles your hardware.

# 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***

The pointer to the video capture context.

***idx***

The index to the frame buffer to release.

## Library:

**libcapture**

## Description:

This function 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 if successful; or -1 if an error occurred (`errno` is set; refer to `errno.h` for more details). The return value and the value that's observed for `errno` may vary as they are dependent on the implementation by the library that handles your hardware.

# capture_set_property_i()

*Set a video frame capture property of type integer*

## Synopsis:

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

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

## Arguments:

***context***

The pointer to the video capture context.

***prop***

The property to set.

***value***

The integer value of the property specified by *prop*.

## Library:

**libcapture**

## Description:

This function sets a video capture property 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:

- 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.
- Call *capture_set_property_p()*, when you have set all the properties you need to specify, 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 if successful; or -1 if an error occurred (`errno` is set; refer to `errno.h` for more details). The return value and the value that's observed for `errno` may vary as they are dependent on the implementation by the library that handles your hardware.

# capture_set_property_p()

*Set a video frame capture property of type (void) pointer*

## Synopsis:

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

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

## Arguments:

***context***

The pointer to the video capture context.

***prop***

The property to set.

***value***

A pointer to a buffer containing the new value of the property to be set. This buffer must be of type `void*`.

## Library:

**libcapture**

## Description:

This function sets an array of video capture properties to pass 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_FRAME_NBUFFERS*.

**Returns:**

0 if successful; or -1 if an error occurred (`errno` is set; refer to `errno.h` for more details). The return value and the value that's observed for `errno` may vary as they are dependent on the implementation by the library that handles your hardware.

# capture_update()

*Apply the capture settings*

## Synopsis:

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

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

## Arguments:

***context***

The pointer to the video capture context.

***flags***

Reserved. Don't use.

## Library:

**libcapture**

## Description:

This function 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.

## Returns:

0 if successful; or -1 if an error occurred (`errno` is set; refer to `errno.h` for more details). The return value and the value that's observed for `errno` may vary as they are dependent on the implementation by the library that handles your hardware. The driver may `slog*`() more detailed error information. Use `slog2info` to retrieve it.

# Helper macros

The video capture API includes macros to help with interval conversions.

### Definitions:

**`#define CAPTURE_INTERVAL_FROM_MS ((x) * 1000000ULL)`**

Convert interval, argument `x`, from milliseconds to nanoseconds.

Required for the `timeout` argument of *capture_get_frame()*.

**`#define CAPTURE_INTERVAL_FROM_US ((x) * 1000ULL)`**

Convert interval, argument `x`, from microseconds to nanoseconds.

Required for the `timeout` argument of *capture_get_frame()*.

**`#define CAPTURE_INTERVAL_FROM_NS (x)`**

Convert interval, argument `x`, from nanoseconds to nanoseconds.

Required for the `timeout` argument of *capture_get_frame()*.

**`#define CAPTURE_INTERVAL_TO_MS ((x) / 1000000ULL)`**

Convert interval, argument `x`, to milliseconds from nanoseconds.

**`#define CAPTURE_INTERVAL_TO_US ((x) / 1000ULL)`**

Convert interval, argument `x`, to microseconds from nanoseconds.

**`#define CAPTURE_INTERVAL_TO_NS (x)`**

Convert interval, argument `x`, to nanoseconds from nanoseconds.

**`#define CAPTURE_INTERVAL_NTSC_FIELD 16668333`**

Amount of time, in nanoseconds, between NTSC fields.

**`#define CAPTURE_INTERVAL_NTSC_FRAME (CAPTURE_INTERVAL_NTSC_FIELD * 2)`**

Amount of time, in nanoseconds, between NTSC frames.

# Index

## S



## T



## V



## W