# Stretch DVR SDK

## API User's Guide

Version 1.2

**Confidential & Proprietary**

Last modified: October 17, 2008

Part #: RU-0129-0001-002

# Contents

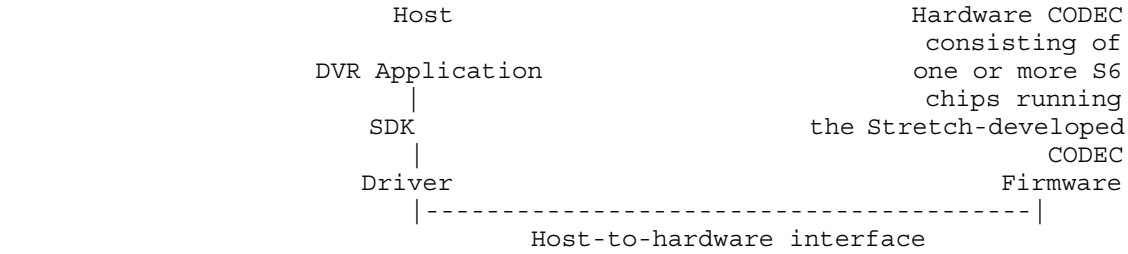
# Chapter 1 Stretch DVR SDK (sdvr_sdk)

## 1.1 Include

```
#include "sdvr_sdk.h"
```

## 1.2 Introduction

Stretch has developed a reference design for PC-based and embedded DVRs. The architecture of a DVR (either PC-based or embedded) using components supplied by Stretch is as follows:




The Stretch DVR (SDVR) hardware consists of one or more S6 processors running the DVR Firmware. Together, the hardware and the firmware are responsible for encoding, decoding, and interfacing to the cameras, microphones, sensors, and relays, and communicating with the host.

On the host there is a driver that handles the low-level communication with the firmware. Sitting on top of the driver is the SDK, which provides an Application-level Programming Interface (API) to the SDVR hardware. It is through this API that an application talks to the hardware. The SDK provides the ability to:

- Discover the capabilities of the hardware CODEC.
- Configure the hardware.
- Encode and decode video and audio.
- Control other capabilities like on-screen display, spot monitor, and so on.

This document describes the Application Programming Interface (API) implemented in the SDK. Your DVR Application can link with the SDK library either statically or dynamically (MS Windows only). Use `sdvr_sdk.lib` to link statically with the SDK library. Use `sdvr_sdk_dll.lib` to link dynamically with the SDK library; the corresponding `sdvr_sdk_dll.dll` needs to be copied with your DVR Application executable.

Throughout this document, we refer to the application that uses the SDK as the *DVR Application*, as it will implement the functionality of a DVR.

## 1.3 Function Groups

The functions in the SDK API are divided into nine major groups.

**System Set Up**. Functions in this group enable you to initialize the SDK, set up the system parameters, and discover and initialize boards at start-up. When you want to exit the application, the APIs in this group allow you to shut down the system gracefully and to free all system resources.

**Channel Set Up**. Functions in this group enable you to initialize each channel at start-up or to reinitialize a channel during operation.

**Encoding**. Functions in this group allow you to start and stop encoding, and to receive encoded and raw audio/video (A/V) buffers.

**Decoding**. Functions in this group allow you to send encoded A/V streams to the decoder for decoding and display.

**Video Display and Sound**. Functions in this group allow you to control raw audio and video streaming.

**On-Screen Display**. Functions in this group allow you to control how on-screen information is displayed and to enable or disable on-screen display (OSD).

**Spot Monitor**. Functions in this group allow you to control the grid pattern and channels appearing in the spot monitor output (SMO).

**RS485 Communication API.** Functions in this group allow you to control the RS-485 interface over which you can implement any PTZ protocol that works over RS-485.

**Sensors and Relays**. Functions in this group allow you to set and reset sensors and relays attached to the hardware.

**Naming conventions.** Functions and data structures start with **`sdvr_`** so as to avoid conflict with symbols in the DVR Application.

All enumerated types end with **`_e`**, and all data structures end with **`_t`**.

## 1.4 Important Notes about the SDK

Note the following:

- The SDK and the firmware on the board is stateless, i.e. parameters used to configure the SDK and the firmware are not stored across sessions (a session starts with initialization and ends when the SDK is closed or the system rebooted). Remember that every time you call `sdvr_sdk_init`() at the beginning of your application you must reinitialize all the system and channel parameters after you connect to the hardware.
- To set the time in firmware, the DVR Application needs to compute the number of seconds elapsed since January 1, 1970 UTC, and to present it to each board using `sdvr_set_date_time`(). When this function is used, the firmware takes the date and time and sets it in hardware. It is required that the DVR Application set the time whenever the time changes (e.g., Daylight Savings Time). We also recommend that the DVR Application monitor the time on the hardware, and if the drift between the host clock and the clock on the hardware gets beyond a certain limit, then reset the time on the hardware.
- Depending on the type of DVR (embedded or PC-based), the hardware may consist of one or more boards, each having one or more S6 chips, and each running a copy of the CODEC firmware. If there are N boards in a system, the SDK numbers them from 0 to N-1.
- When configuring the system, the DVR Application is required to set the system-wide video standard and resolution. This is the maximum video resolution that is supported by the SDK. Lesser video resolutions can be supported, but the lesser resolutions can only be 1/4 of or 1/16 of the maximum resolution.
- If you mark a region as private, that region is blocked out in both the live and the encoded stream.
- When decoding a frame, you must enable raw video or SMO for the decoder channel to receive and display the decoded raw video frames.
- For any alarm detection (i.e., motion detection, blind detection, and so on) to be active, the camera must be either encoding video frames or streaming raw video.
- If the specified decimation and x,y coordinates of an SMO grid do not correctly fit on the monitor, the grid is not displayed.
- While encoding, any of the encoder parameters can be changed, except the video decimation.

## 1.5 Important Restrictions

Note the following restrictions when using the SDK. The restrictions, apart from those explicitly noted, are not permanent and will be removed in future versions of the SDK.

- Only one DVR Application can use the SDK on one host. Multiple copies of the SDK and associated driver cannot reside on the same host. This is a permanent restriction.
- All SDVR cards in a system must have the same video standards and camera resolution, e.g., NTSC D1 720x480 at 30 fps or PAL 720x576 at 25 fps.
- Only two OSD text strings are supported per channel.
- For OSD text only, the ASCII character set is supported. Double-byte and Unicode character sets are supported only if the corresponding font table is loaded in the firmware.
- OSD text color and logo display are currently not supported.
- OSD text for decoder channels is currently not supported.
- Only H.264 decoding is supported.
- Audio decoding is not supported.
- Currently, the APIs that save A/V frames into a file is limited to the .mov file format. Additionally, these APIs are not officially supported in this release.
- Privacy blanking supported only in the VPP analytics mode.
- Blind detection is supported only in the VPP analytics mode.
- SMO dwell time field is not supported. This means only one video channel can be specified in any SMO grid.

## 1.6 Format of Data Buffer Exchange Between the SDK and DVR Application

There are two type of data buffers: `sdvr_av_buffer_t` and `sdvr_yuv_buffer_t`.

`sdvr_av_buffer_t` is used to exchange encoded video, encoded audio, and raw audio data. `sdvr_yuv_buffer_t` is used to exchange raw video data.

The raw A/V frames can be either from the live or decoded video stream. The format of a raw video is YUV 4:2:0. A raw video frame is of type `sdvr_yuv_buffer_t` that consists of a header followed by three payloads. Each payload is a pointer to a Y, U, or V buffer. Each Y, U, or V buffer is of type `sdvr_av_buffer_t`. The format of a raw audio is PCM. A raw audio frame is of type `sdvr_av_buffer_t` that consists of a header followed by a payload.

The encoder encodes the incoming video and audio frame-by-frame and each encoded frame is sent to the host. A frame consists of a header generated by the encoder and a payload. The header format is Stretch proprietary and has information that may be relevant to the DVR Application (e.g., whether motion was detected). The header format `sdvr_av_buffer_t` is described later in this document.

The payload format contains the video stream in elementary stream format. Therefore, if the video payload is stored in its own file, it can be played by any player that supports the elementary stream format and has the corresponding decoder.

## 1.7 Using the SDK API

This section provides a high-level overview of the various DVR Application tasks and the functions used to accomplish those tasks.

### 1.7.1 SDK and Board Initialization and Set-Up

The DVR Application needs to perform the following tasks during initialization and set-up.

Initialize the SDK using `sdvr_sdk_init()`.

Load firmware into every DVR board that needs to be connected by calling `sdvr_upgrade_firmware()`. This is required only in cases when no firmware is burned onto the board, or when you want to use different firmware than that already burned onto the board.

For a PC-based DVR, get the number of boards using `sdvr_get_board_count()`. For an embedded DVR, the board count is almost always 1.

For both PC-based and embedded DVRs, get the board attributes using `sdvr_get_board_attributes()`.

Set the amount of memory allocated for communication with the hardware using `sdvr_set_sdk_params()`. We recommend that you use default vid-

eo buffer settings for that call `sdvr_get_sdk_params`(), and only change the fields of interest to you. During development, we recommend that you enable debugging using many different flags in the SDK parameter structure, and specify a log file to record tracing of communication between the DVR Application and the DVR firmware.

Connect to each board using `sdvr_board_connect`() and set the video standard (NTSC or PAL) and maximum resolution. (You can get a list of supported video standards from `sdvr_get_board_attributes`().) Additionally, specify whether the host PC should perform start code emulation for H.264 CODECs. For performance reasons, it is strongly recommended that you always set this field to 1 except for the embedded DVR Applications, for which it must be set to 0. This establishes a connection to the board for further control and data communication.

If required, the SDK, driver, boot loader, and firmware versions can be obtained using `sdvr_get_sdk_version`(), `sdvr_get_driver_version`() and `sdvr_get_firmware_version_ex`(). Stretch will publish a matrix of which SDK, driver, boot loader, and firmware versions are compatible. This information can be used to check that compatible versions are used.

Get the capabilities of each board, i.e., the number of cameras it can have, the number of sensors, etc. using `sdvr_get_board_config`().

Set the date and time in the firmware using `sdvr_set_date_time`(). Subsequently, the DVR Application should periodically monitor the time on the firmware using `sdvr_get_date_time`(), and if there is drift between the host and the firmware clock, reset the clock on the firmware using `sdvr_set_date_time`().

Set the callback function for the SDK to call when an error is encountered in the firmware using `sdvr_signals_callback`(). This is not mandatory, but is a very useful tool for debugging.

Set the callback function for the SDK to call when sensors are triggered using `sdvr_set_sensor_callback`(). This is mandatory if the DVR Application wants to be notified about sensors being triggered.

Set the callback function for the SDK to call when video alarms are triggered using `sdvr_set_video_alarm_callback`(). This is not mandatory as you can get some of this information from the A/V buffer header, but we highly recommend that you register a callback for video alarms.

Set the callback function for the SDK to call when AV frames are available from the hardware using `sdvr_set_av_frame_callback`(). This callback function is not mandatory, as described later.

If you need to display OSD text in a language other than English, load the corresponding font file using `sdvr_osd_set_font_table`().

If necessary, set the watchdog timer in the hardware using `sdvr_set_watchdog_state()`. The watchdog timer is used to reset the entire DVR system to prevent it from hanging. If the software on the host and the firmware is alive, then the watchdog should be periodically reset before it expires (the current value can be obtained using `sdvr_get_watchdog_state()`).

If necessary, run diagnostics on the hardware using `sdvr_run_diagnostics()`. The diagnostics run pertain only to the S6-part of the system, as described later.

## 1.7.2 Channel Set-Up

Before using the system, you must create different channel types to perform encoding or decoding tasks. An encoder channel provides encoded video and audio, and raw video and audio data.

To create and configure each channel type, the DVR Application needs to perform the following tasks during channel set-up.

If there are N cameras supported by the system, then channels 0 to N-1 are encoding channels. Each encoding channel number corresponds to the camera position number in the back panel. Set each of these channels as encoding channels unless you want to leave some cameras unconnected and use the processing power for decoding. To create an encoding channel, use `sdvr_create_chan()`. You can dynamically turn off encoding channels to free up processing power for decoding. Additionally, if there are channels created with secondary encoding support, you may not be able to connect to all the cameras for recording. You can also specify audio encoding for all cameras that have corresponding audio channels connected. Additionally, you may want to create some streaming-only channels that do not need to have any encoding capabilities. These are called HMO- or SMO-only channels. To create an HMO- or SMO-only channel, set the channel type to `SDVR_CHAN_TYPE_ENCODER` and the primary video format to `SDVR_VIDEO_ENC_NONE`.

> **NOTE:** The HMO- or SMO-only channels reduce the maximum number of encoding channels that are allowed by the system. This means that if there are N cameras supported and you create M HMO- or SMO-only channels, you can only create N minus M number of encoding channels.

After encoding channels are set up, set up decoding channels using `sdvr_create_chan()`. The maximum number of decoding channels depends on the processing power left after setting up the encoder channels. If the call to `sdvr_create_chan()` returns successfully, then the decoding

channel was set up. Otherwise, there is no more processing power available and the decoding channel could not be set up. Currently, the decoding channel number indicates in which S6 chip the decoder is created. Similar to encoder channels, decoder channel numbers run from 0 to N-1. Where N is the maximum number of decoders supported returned by `sdvr_get_board_config`(). Every 4 consecutive decoder channel numbers are created in one S6 chip starting from decoder channel 0 in the first S6 chip. For example, to have one decoder channel created in every S6 chip in a four-S6 chip DVR board, you should assign channel numbers 0, 4, 9, and 13 when creating the decoder channels.

When you set a channel to be encoding or decoding, you receive a unique channel handle of type `sdvr_chan_handle_t`. Subsequently, you use this handle in all function calls requiring a channel identifier.

Existing encoding or decoding channels can be destroyed at any time by calling `sdvr_destroy_chan`(). This call lets you rebalance channel types as needed.

Set the parameters of each video encoding channel using `sdvr_set_video_encoder_channel_params`().

Set the parameters of each alarm-triggered video encoding channel using `sdvr_set_alarm_video_encoder_params`().

Set the parameters of each audio encoding channel using `sdvr_set_audio_encoder_channel_params`(). (Currently, there are no audio encoder parameters.)

Set regions for motion and blind detection, and privacy regions using `sdvr_add_region`().

You can associate application-defined data with each encoding or decoding channel by calling `sdvr_set_chan_user_data`(). This data can be retrieved at a later time by calling `sdvr_get_chan_user_data`().

Enable motion, blind, and night detection using `sdvr_enable_motion_detection`(), `sdvr_enable_blind_detection`(), and `sdvr_enable_night_detection`(), respectively.

Enable privacy regions using `sdvr_enable_privacy_regions`().

> **NOTE:** For privacy blocking and blind detection to be operational, you must create the encoder channel with *vpp_mode* field set to `SDVR_VPP_MODE_ANALYTICS`.

The hardware is now set up for encoding, decoding, and display.

### 1.7.3 Encode

The DVR Application needs to perform the following tasks during encoding.

Enable the encoder for each channel using `sdvr_enable_encoder`().

Get a frame of encoded video for a particular channel using `sdvr_get_av_buffer`(). If a callback was registered for AV frames, then the DVR Application should have received one or more callbacks with information about the channels for which encoded frames are available. (See `sdvr_set_av_frame_callback`() for a usage example of such a callback). That information can be used to determine which channels have data and to request frames only from those channels. If, however, a callback was not registered, `sdvr_get_av_buffer`() can be used as a polling function - when channels have encoded AV frames this function returns valid buffers, but returns appropriate error codes for channels that do not have new frames available.

Stretch has implemented a one-copy buffer management policy. The buffers required to hold incoming data from the board (and data going to the board) are allocated in the SDK. The driver, however, allocates a few buffers that are contiguous in physical memory to enable efficient DMA. Data coming from the board is first stored in the driver buffers and then copied to the SDK buffers. Similarly, data going to the board is first stored in the SDK buffers and then copied to the driver buffers before being DMAed to the board.

It is important to release buffers obtained from the SDK using `sdvr_release_av_buffer`() for encoded frames and `sdvr_release_yuv_buffer`() for raw video frames. The SDK then recycles these buffers and uses them for holding future incoming frames. Under no circumstances should a buffer be freed by the DVR Application. Also, holding on to the buffer for too long causes the SDK to run out of buffers, and frames are lost. It is therefore important that enough buffers be allocated and that buffers are released in a timely manner.

It is the responsibility of the DVR Application to save the data to disk. Stretch does not provide functions to write to disk because disk management and recording policy is dependent on the DVR Application and best managed by it.

### 1.7.4 Decode

The DVR Application needs to perform the following tasks during decoding.

Enable the decoder for each channel using `sdvr_enable_decoder`().

In keeping with Stretch's buffer management policy whereby the SDK manages all the buffers, the DVR Application must request a free frame buffer from the SDK using `sdvr_alloc_av_buffer`().

The DVR Application should fill this buffer with encoded data from the disk. After this buffer is full, it can be sent to the hardware for decoding using `sdvr_send_av_frame`(). Decoder buffers are released as part of the call to `sdvr_send_av_frame`(). In the event that a decoder buffer is acquired, but needs to be released without sending, you can call `sdvr_release_av_buffer`().

The decoded raw video frames can either be displayed on the SMO by calling `sdvr_set_smo_grid`(), or be sent to the DVR Application for display on the host monitor by calling `sdvr_stream_raw_video`(). After streaming of raw video frames is enabled for the decoded channel, the raw video frames for the corresponding decoder will be sent through the `av_frame_callback` function (refer to Section 1.7.5, "Raw Video and Audio Data" for detailed information).

## 1.7.5 Raw Video and Audio Data

The SDK provides raw (unencoded) audio and video for each channel as a separate stream to the DVR Application. These raw A/V frames can be either from the live or decoded video stream. The DVR Application needs to do the following to get raw video and audio data.

Enable streaming of video and audio from the hardware or decoder to the host using `sdvr_stream_raw_video`() and `sdvr_stream_raw_audio`() calls. Although video for various channels can be enabled for streaming, it makes sense to enable streaming for only one audio channel (although the SDK supports streaming of multiple audio channels). To conserve communication bandwidth between the host and the board, we recommend that raw video streaming be enabled only for channels that are being displayed, and not for all channels.

Register a callback function using `sdvr_set_av_frame_callback`() so that the DVR Application can be informed when raw audio and video frames are available. This is not strictly required (see preceding discussion).

Call `sdvr_get_av_buffer`() to get the raw audio or encoded audio/video frames from the SDK. Call `sdvr_get_yuv_buffer`() to get raw video frames. The format of raw video is YUV 4:2:0, and will be received as three separate buffers containing Y, U, and V. Use appropriate rendering and sound playback software and hardware to display the video and to play the sound. Video for each channel is obtained separately, and it is the responsibility of the DVR Application to display the video in its appropriate window. Typically, the DVR Application creates tiled windows for each channel to be displayed, and the video for each channel is rendered in its own window. Stretch provides a UI SDK library that helps you display video frames in any

region within a display window handle. Refer to `sdvr_ui_sdk.html` for details.

Release the frame buffer obtained in the previous step using `sdvr_release_av_buffer`() or `sdvr_release_yuv_buffer`() as appropriate. Raw video buffers are large, so there will usually not be too many of these buffers per channel. Therefore, it is extremely important that these buffers be promptly released.

## 1.7.6 On-Screen Display (OSD)

Each encoder or decoder channel can have two different OSD text items to be displayed at any position in the video frame. To show OSD, the DVR Application needs to do the following:

It first needs to configure each OSD text item using `sdvr_osd_text_config_ex`(). The *osd_text_config* data structure allows the DVR Application to specify the text string, its location, whether date and time should be appended to the string, and if so, the style in which the date and time is displayed. Stretch provides multi-language support through multiple font tables. You can load a non-English language font by calling `sdvr_osd_set_font_table`(). If no font table is loaded, the default English font is used.

After an OSD text item is configured, it can be shown or hidden using `sdvr_osd_text_show`(). In the case of encoder channels, OSD text is blended into the video before the video is encoded. Therefore, the OSD text present at the time of encoding is displayed during decoding. In the case of decoder channels, OSD text is added after the video frame is decoded. Therefore, it is possible to have two different OSD text items on the decoded frame, one when it was encoded and the other when it is decoded.

**NOTE:** New style OSD (OSD with transparent background and Unicode UTF-16 coding scheme supporting Basic Multilingual Plane (BMP)) is only supported for encoder channels with `vpp_mode` fields set to `SDVR_VPP_MODE_SLATERAL`.

## 1.7.7 Spot Monitor Output (SMO)

Stretch provides displaying of the live video output from each encoder channel or the decode frames output of each decode channel anywhere within the spot monitor display. The combination of SMO display location and its video decimation for each channel is called an SMO grid. Each SMO grid can be defined and enabled by calling `sdvr_set_smo_grid`(). You can achieve different display patterns by placing the channel video output in various display locations.

Stretch provides two SMO feature solution. One is analog, which is a direct pass-through of only one video channel to the Spot Monitor. In this mode, only one video channel can be displayed in the full resolution and no OSD text can be viewed on that monitor. The other SMO mode provides full-function SMO capabilities with multiple channels and OSD. If this mode is selected, the last video channel in the last S6 chip will be lost. This means in a 16-channel board only 15 channels are available. To enable SMO, the DVR Application needs to do the following:

Specify the grid pattern using `sdvr_set_smo_grid`(). The SMO grid pattern is flexible and is defined by the user. Each tile can be either enabled or disabled. When enabled, there are one or more channels to display at this tile. If disabled, the tile is not used. The channel specified at each location can be either an encoding or decoding channel. If it is an encoding channel, live video is displayed, and if it is a decode channel then playback video is displayed. Each tile also has a display resolution decimation.

Setting the SMO configuration using `sdvr_set_smo_grid`() enables displaying of SMO. No further action is required on the part of the DVR Application.

## 1.7.8 Pan, Tilt, and Zoom

Typically, the PTZ controller for a camera is connected to the processor implementing the PTZ protocol through an RS-485 interface. The S6 processor does not have an RS-485 interface, so the Stretch reference design provides the mechanism to talk to the RS-485 port on the alarm I/O card.

The SDK does not support any particular PTZ protocol. Instead, it gives access to the RS-485 interface through the RS-232C interface. The PTZ section of the SDK allows the DVR Application to set up the RS23C port on the S6 to talk to the RS-485. The following functions are available:

- `sdvr_init_uart`() allows the DVR Application to set the baud rate, stop bits, etc. for the UART interface.
- `sdvr_read_uart`() returns up to 255 characters from the UART port.
- `sdvr_write_uart`() writes up to 255 characters to the UART port.

## 1.7.9 Sensors and Relays

Sensors are external inputs that can be triggered, for example, by a door opening. The SDK allows the DVR Application to register a callback so that any time one or more sensors are triggered, the callback function is called. This callback is registered using `sdvr_set_sensor_callback`() and the type of the callback function is `sdvr_sensor_callback`. Some sensors are edge triggered,

whereas others are level sensitive. You can specify how each sensor should be triggered, as well as its initial enable status by calling `sdvr_config_sensors`(). You can also enable or disable an individual sensor at different times by calling `sdvr_enable_sensor`().

Relays are actuators that are activated by the DVR Application. To activate or deactivate a relay, the DVR Application calls `sdvr_trigger_relay`() with the proper value for the *is_triggered* flag.

### 1.7.10 System Shutdown

The DVR Application needs to perform the following tasks during shutdown:

Disable encoding on all channels using `sdvr_enable_encoder`(), with the *enable* flag set to false.

Disable decoding on all channels using `sdvr_enable_decoder`(), with the *enable* flag set to false.

Disable all relays using `sdvr_trigger_relay`(), with the *is_triggered* flag set to false.

Disconnect from boards using `sdvr_board_disconnect`(). This frees up all board-specific resources in the SDK and driver.

> **NOTE:** After you call `sdvr_board_disconnect`(), you must load the firmware by calling `sdvr_upgrade_firmware`() every time prior to reconnecting to the DVR board when no firmware is loaded into the board's non-volatile memory.

Close the SDK and free up all system resources using `sdvr_sdk_close`(). Although disabling and destroying all the channels, as well as disconnecting from all boards is good programming practice, it is not required because `sdvr_sdk_close`() performs these actions.

## 1.8 Types

**sx_uint8**

```
typedef unsigned char sx_uint8;
```

8-bit unsigned integer.

**sx_uint16**

```
typedef unsigned short sx_uint16;
```

16-bit unsigned integer.

**sx_uint32**

```
typedef unsigned int sx_uint32;
```

32-bit unsigned integer.

**sx_uint64**

```
typedef unsigned long long sx_uint64;
```

64-bit unsigned integer.

**sx_bool**

```
typedef unsigned int sx_bool;
```

Boolean value.

**sx_int8**

```
typedef signed char sx_int8;
```

8-bit signed integer.

**sx_int16**

```
typedef short sx_int16;
```

16-bit signed integer.

**sx_int32**

```
typedef int sx_int32;
```

32-bit signed integer.

**sx_int64**

```
typedef long long sx_int64;
```

64-bit signed integer.

**time_t**

```
typedef long time_t;
```

32 bit time value, the number of seconds elapsed since midnight 01/01/1970.

**_sdvr_err_e**

```
typedef enum _sdvr_err_e {
  SDVR_ERR_NONE = 0,
  SDVR_FRMW_ERR_WRONG_CAMERA_NUMBER,
  SDVR_FRMW_ERR_WRONG_CAMERA_TYPE,
  SDVR_FRMW_ERR_WRONG_CODEC_FORMAT,
  SDVR_FRMW_ERR_WRONG_CODEC_RESOLUTION,
  SDVR_FRMW_ERR_WRONG_CHANNEL_TYPE,
  SDVR_FRMW_ERR_WRONG_CHANNEL_ID,
  SDVR_FRMW_ERR_WRONG_VIDEO_FORMAT,
  SDVR_FRMW_ERR_WRONG_AUDIO_FORMAT,
  SDVR_FRMW_ERR_EXCEED_CPU_LIMIT,
  SDVR_FRMW_ERR_CHANNEL_NOT_CREATED,
  SDVR_FRMW_ERR_CHANNEL_ALREADY_CREATED,
  SDVR_FRMW_ERR_CHANNEL_NOT_ENABLED,
  SDVR_FRMW_ERR_CHANNEL_NOT_DISABLED,
  SDVR_FRMW_ERR_SMO_NOT_CREATED,
  SDVR_FRMW_ERR_INVALID_TIME,
  SDVR_FRMW_ERR_ILLEGAL_SMO_PARAMS,
  SDVR_FRMW_ERR_SMO_NOT_SUPPORTED,
  SDVR_FRMW_ERR_VDET_ERROR,
  SDVR_FRMW_ERR_RUNTIME_ERROR,
  SDVR_FRMW_ERR_VPP_RUNTIME_ERROR,
  SDVR_FRMW_ERR_ENCODER_RUNTIME_ERROR,
  SDVR_FRMW_ERR_DECODER_RUNTIME_ERROR,
  SDVR_FRMW_ERR_ILLEGAL_PARAMETER,
  SDVR_FRMW_ERR_INTERNAL_ERROR,
  SDVR_FRMW_ERR_ILLEGAL_COMMAND,
  SDVR_FRMW_ERR_SMO_NOT_DISABLED,
```

```
SDVR_FRMW_ERR_OUT_OF_MEMORY,
SDVR_DRV_ERR_MSG_RECV = 255,

SDVR_DRV_ERR_INVALID_PARAMETER = 1000,
SDVR_DRV_ERR_DEVICE_IN_USE,
SDVR_DRV_ERR_DEVICE_OPEN,
SDVR_DRV_ERR_DEVICE_CLOSE,
SDVR_DRV_ERR_DEVICE_RESET,
SDVR_DRV_ERR_IPC_INIT,
SDVR_DRV_ERR_NO_CHANNELS,
SDVR_DRV_ERR_CHANNEL_IN_USE,
SDVR_DRV_ERR_CHANNEL_CREATE,
SDVR_DRV_ERR_CHANNEL_CONNECT,
SDVR_DRV_ERR_CHANNEL_CLOSE,
SDVR_DRV_ERR_CHANNEL_NOT_ACTIVE,
SDVR_DRV_ERR_CHANNEL_DEAD,
SDVR_DRV_ERR_NO_RECV_BUFFERS,
SDVR_DRV_ERR_NO_SEND_BUFFERS,
SDVR_DRV_ERR_MSG_SEND,

SDVR_ERR_OUT_OF_MEMORY = 2000,
SDVR_ERR_INVALID_HANDLE,
SDVR_ERR_INVALID_ARG,
SDVR_ERR_INVALID_BOARD,
SDVR_ERR_BOARD_CONNECTED,
SDVR_ERR_INVALID_CHANNEL,
SDVR_ERR_CHANNEL_CLOSED,
SDVR_ERR_BOARD_CLOSED,
SDVR_ERR_NO_VFRAME,
SDVR_ERR_NO_AFRAME,
SDVR_ERR_INTERNAL,
SDVR_ERR_BOARD_NOT_CONNECTED,
SDVR_ERR_IN_STREAMING,
SDVR_ERR_NO_DVR_BOARD,
SDVR_ERR_WRONG_DRIVER_VERSION,
SDVR_ERR_DBG_FILE,
SDVR_ERR_ENCODER_NOT_ENABLED,
SDVR_ERR_ENCODER_NOT_DISABLED,
SDVR_ERR_SDK_NO_FRAME_BUF,
SDVR_ERR_INVALID_FRAME_TYPE,
SDVR_ERR_NOBUF,
SDVR_ERR_CALLBACK_FAILED,
SDVR_ERR_INVALID_CHAN_HANDLE,
SDVR_ERR_COMMAND_NOT_SUPPORTED,
SDVR_ERR_ODD_SMO_COORDINATES,
SDVR_ERR_LOAD_FIRMWARE,
SDVR_ERR_WRONG_CHANNEL_TYPE,
SDVR_ERR_DECODER_NOT_ENABLED,
SDVR_ERR_BUF_NOT_AVAIL,
SDVR_ERR_MAX_REGIONS,
SDVR_ERR_INVALID_REGION,
SDVR_ERR_INVALID_GOP,
SDVR_ERR_INVALID_BITRATE,
SDVR_ERR_INVALID_BITRATE_CONTROL,
SDVR_ERR_INVALID_QUALITY,
SDVR_ERR_INVALID_FPS,
SDVR_ERR_UNSUPPORTED_FRIMWARE,
SDVR_ERR_INVALID_OSD_ID,
```

```
  SDVR_ERR_OSD_LENGTH,
  SDVR_ERR_OSD_FONT_FILE,
  SDVR_ERR_FONT_ID
  SDVR_ERR_CAMERA_IN_REC,
  SDVR_ERR_OPEN_REC_FILE,
  SDVR_ERR_FAILED_ADD_VIDEO_TRACK,
  SDVR_ERR_FAILED_ADD_AUDIO_TRACK,
  SDVR_ERR_SDK_BUF_EXCEEDED

  } sdvr_err_e;
```

Typedef for the errors returned by the SDK.

SDVR_ERR_NONE - No error, or in other words, success!

### DVR Firmware Error Codes

The following error codes are generated by the DVR firmware:

SDVR_FRMW_ERR_WRONG_CAMERA_NUMBER - The given camera number is invalid.

SDVR_FRMW_ERR_WRONG_CAMERA_TYPE - Error code if the specified video standard is not supported by the firmware.

SDVR_FRMW_ERR_WRONG_CODEC_FORMAT - Error code if the specified video codec is not supported.

SDVR_FRMW_ERR_WRONG_CODEC_RESOLUTION

SDVR_FRMW_ERR_WRONG_CHANNEL_TYPE

SDVR_FRMW_ERR_WRONG_CHANNEL_ID

SDVR_FRMW_ERR_WRONG_VIDEO_FORMAT

SDVR_FRMW_ERR_WRONG_AUDIO_FORMAT

SDVR_FRMW_ERR_EXCEED_CPU_LIMIT

SDVR_FRMW_ERR_CHANNEL_NOT_CREATED

SDVR_FRMW_ERR_CHANNEL_ALREADY_CREATED

SDVR_FRMW_ERR_CHANNEL_NOT_ENABLED

SDVR_FRMW_ERR_CHANNEL_NOT_DISABLED

SDVR_FRMW_ERR_INVALID_TIME

SDVR_FRMW_ERR_ILLEGAL_SMO_PARAMS

SDVR_FRMW_ERR_SMO_NOT_SUPPORTED - There is no SMO support in the firmware.

SDVR_FRMW_ERR_VDET_ERROR - Error code if the firmware detects cameras connected with mixed video standard connected. (i.e., some NTSC cameras and some PAL cameras).

SDVR_FRMW_ERR_RUNTIME_ERROR

SDVR_FRMW_ERR_VPP_RUNTIME_ERROR

SDVR_FRMW_ERR_ENCODER_RUNTIME_ERROR

SDVR_FRMW_ERR_DECODER_RUNTIME_ERROR

SDVR_FRMW_ERR_ILLEGAL_PARAMETER

SDVR_FRMW_ERR_INTERNAL_ERROR

SDVR_FRMW_ERR_ILLEGAL_COMMAND

SDVR_FRMW_ERR_SMO_NOT_DISABLED - Error code if you tried to reset channel parameters to factory default while SMO was enabled.

SDVR_FRMW_ERR_OUT_OF_MEMORY - Error code if the firmware runs out of memory in the middle of the current operation.

SDVR_FRMW_ERR_NO_IO_BOARD - Error code if the current command requires an I/O board for the operation but the I/O board is not connected to the DVR board.

### PCI Driver Interface Error Codes

The following error codes are generated by the PCI driver interface:

SDVR_DRV_ERR_INVALID_PARAMETER - Refer to the function prototype for information.

SDVR_DRV_ERR_BOARD_IN_USE - The given board index is already in use.

SDVR_DRV_ERR_BOARD_CONNECT - Failed to connect to the board.

SDVR_DRV_ERR_BOARD_CLOSE - Failed while trying to close the board.

SDVR_DRV_ERR_BOARD_RESET - Failed to reset the DVR board.

SDVR_DRV_ERR_IPC_INIT

SDVR_DRV_ERR_NO_CHANNELS

SDVR_DRV_ERR_CHANNEL_IN_USE - If a receive channel has not been closed by the firmware.

SDVR_DRV_ERR_CHANNEL_CREATE

SDVR_DRV_ERR_CHANNEL_CONNECT

SDVR_DRV_ERR_CHANNEL_CLOSE - If an error occurred while closing the channel, or the firmware did not respond correctly to the close request.

SDVR_DRV_ERR_CHANNEL_NOT_ACTIVE

SDVR_DRV_ERR_CHANNEL_DEAD

SDVR_DRV_ERR_NO_RECV_BUFFERS

SDVR_DRV_ERR_NO_SEND_BUFFERS

SDVR_DRV_ERR_MSG_SEND - Error code if the driver failed to send the command to the firmware.

SDVR_DRV_ERR_MSG_RECV - Error code if the driver timed out while waiting to receive a response from the firmware. This error could be an indication that the code on the firmware hung.

### DVR SDK Error Codes

The following error codes are generated by the DVR SDK:

SDVR_ERR_OUT_OF_MEMORY - System is out of memory.

SDVR_ERR_INVALID_HANDLE - Invalid buffer handle.

SDVR_ERR_INVALID_ARG - Invalid argument to a function call.

SDVR_ERR_INVALID_BOARD - Invalid board number.

SDVR_ERR_BOARD_CONNECTED - The current operation is invalid while connecting to a board (i.e., setting SDK parameters or connect to a board that is already connected).

SDVR_ERR_INVALID_CHANNEL - Invalid channel number.

SDVR_ERR_CHANNEL_CLOSED - Channel is closed - cannot communicate with it.

SDVR_ERR_BOARD_CLOSED - Board is closed - cannot communicate with it.

SDVR_ERR_NO_VFRAME - No video frame is available.

SDVR_ERR_NO_AFRAME - No audio frame is available.

SDVR_ERR_INTERNAL - Internal error in the SDK. Please contact Stretch support for assistance.

SDVR_ERR_BOARD_NOT_CONNECTED - The specified board index was not connected.

SDVR_ERR_IN_STREAMING - Failed to close the board because some channels are still active (encoding or decoding).

SDVR_ERR_NO_DVR_BOARD - No PCIe DVR board was found on the PC.

SDVR_ERR_WRONG_DRIVER_VERSION - The current DVR PCIe driver is not supported.

SDVR_ERR_DBG_FILE - Failed to open the debug file.

SDVR_ERR_ENCODER_NOT_ENABLED - Failed to start the encoder on the given channel.

SDVR_ERR_ENCODER_NOT_DISABLED - Failed to stop the encoder on the given channel.

SDVR_ERR_SDK_NO_FRAME_BUF - There is not enough buffer allocated to receive encoded or raw frame buffers. Or the frame size is zero.

SDVR_ERR_INVALID_FRAME_TYPE - The given frame type is not supported.

SDVR_ERR_NOBUF - No A/V buffer is available.

SDVR_ERR_CALLBACK_FAILED - Failed to register callback with the driver.

SDVR_ERR_INVALID_CHAN_HANDLE - The given channel handle is invalid.

SDVR_ERR_COMMAND_NOT_SUPPORTED - The function is not implemented.

SDVR_ERR_ODD_SMO_COORDINATES - Error code if either of the x or y value of the SMO grid is an odd number.

SDVR_ERR_LOAD_FIRMWARE - Error code if failed to load the firmware. This could be as result of invalid file path or failure to load from PCIe driver.

SDVR_ERR_WRONG_CHANNEL_TYPE - The channel handle belongs to wrong channel type for the current operation.

SDVR_ERR_DECODER_NOT_ENABLED - Error code if we are trying to send frames to be decoded but the decoder is not enabled.

SDVR_ERR_BUF_NOT_AVAIL - Error code if no buffer is available to send frames.

SDVR_ERR_MAX_REGIONS - Error code if the maximum allowed regions is reached when you request to add a new motion or blind detection, as well as privacy region.

SDVR_ERR_INVALID_REGION - Error code if the given region either does not exist or is invalid.

SDVR_ERR_INVALID_GOP - Error code if the specified GOP value of the encoder parameter to be set is zero.

SDVR_ERR_INVALID_BITRATE - Error code if the specified maximum bit rate is less than the average bit rate while setting the encoder parameters.

SDVR_ERR_INVALID_BITRATE_CONTROL - Error code if an unknown encoder bit rate control is specified.

SDVR_ERR_INVALID_QUALITY - Error code if the encoder quality parameter is out of range for the current video encoder.

SDVR_ERR_INVALID_FPS - Error code if the specified encoder frame rate is not supported by the current video standard.

SDVR_ERR_UNSUPPORTED_FRIMWARE - Error code if the DVR firmware version is not supported by the current version of SDK.

SDVR_ERR_INVALID_OSD_ID - Error code if the specified OSD ID has not been configured before using it or is out of the valid range of OSD items on a channel.

SDVR_ERR_OSD_LENGTH - Error code if the given OSD text length is too large.

SDVR_ERR_OSD_FONT_FILE - Error code if the given font table file does not exist, is invalid, or cannot be opened.

SDVR_ERR_FONT_ID - Error code if the given font ID does not fall within the valid range, or the font table with this ID does not exist.

SDVR_ERR_CAMERA_IN_REC - Error code if the camera that is being requested for recording, is currently recording.

SDVR_ERR_OPEN_REC_FILE - Error code for failure to open the given file for recording.

SDVR_ERR_FAILED_ADD_VIDEO_TRACK - Error code for failure to add the video track to the recording file.

SDVR_ERR_FAILED_ADD_AUDIO_TRACK - Error code for failure to add the audio track to the recording file.

SDVR_ERR_SDK_BUF_EXCEEDED - Error code if any buffer size, while setting up SDK buffers, exceeds the maximum allowed.

**_sdvr_diag_code_e**

```
typedef enum _sdvr_diag_code_e {
  SDVR_DIAG_OK = 0x00000000,
  SDVR_DIAG_DDR_WRITEREAD_FAIL = 0xb007e001,
  SDVR_DIAG_DDR_ADDRLINES_FAIL = 0xb007e002,
  SDVR_DIAG_DDR_BITFLIP_FAIL = 0xb007e003,
  SDVR_DIAG_DDR_DMA_FAIL = 0xb007e004,
  SDVR_DIAG_DDR_READ_DMA_FAIL = 0xb007e005,
  SDVR_DIAG_PLL_TEST_MHZ = 0x1000e001,
  DVR_DIAG_PLL_TEST_SYS = 0x1000e002,
  SDVR_DIAG_PLL_TEST_IO = 0x1000e003,
  SDVR_DIAG_PLL_TEST_AIM = 0x1000e004,
  SDVR_DIAG_PLL_TEST_DP0 = 0x1000e005,
  SDVR_DIAG_PLL_TEST_DP2 = 0x1000e006,
  SDVR_DIAG_PLL_TEST_DDR = 0x1000e007,
  SDVR_DIAG_SPI_TEST_READ = 0x1001e001,
  SDVR_DIAG_SPI_TEST_ERASE = 0x1001e002,
  SDVR_DIAG_SPI_TEST_PROG = 0x1001e003,
  SDVR_DIAG_SPI_TEST_UNLOCK = 0x1001e004,
  SDVR_DIAG_SPI_TEST_COMPARE = 0x1001e005,
  SDVR_DIAG_SPI_TEST_MAINT = 0x1001e006,
  SDVR_DIAG_SPI_TEST_MISC = 0x1001e007,
  SDVR_DIAG_TWI_EEPROM_TEST_READ = 0x1002e001,
  SDVR_DIAG_TWI_EEPROM_TEST_WRITE = 0x1002e002,
  SDVR_DIAG_TWI_EEPROM_TEST_INIT = 0x1002e003,
```

```
  SDVR_DIAG_TWI_EEPROM_TEST_COMPARE = 0x1002e004,
  SDVR_DIAG_EPSON_REG_TEST_INIT = 0x1003e001,
  SDVR_DIAG_EPSON_REG_TEST_WALKING = 0x1003e002,
  SDVR_DIAG_TW2815_AUDIO_TEST_INIT = 0x1004e001,
  SDVR_DIAG_TW2815_AUDIO_TEST_NO_AUDIO = 0x1004e002,
  SDVR_DIAG_TW2815_REG_TEST = 0x1004e003,
  SDVR_DIAG_TW2815_VIDEO_TEST_INIT = 0x1004e004,
  SDVR_DIAG_TW2815_VIDEO_TEST_NO_VIDEO = 0x1004e005,
  SDVR_DIAG_TW28XX_VIDEO_TEST_TIMEOUT = 0x1004e015,
  SDVR_DIAG_TW28XX_VIDDET_TEST_INIT_ERR = 0x1004e006,
  SDVR_DIAG_TW28XX_VIDDET_TEST_UNKNOWN_CHIP = 0x1004e007,
  SDVR_DIAG_TW28XX_VIDDET_TEST_NO_INPUT_ERR = 0x1004e008,
  SDVR_DIAG_TW28XX_VIDDET_TEST_CONFLICT_ERR = 0x1004e009,
  SDVR_DIAG_TW28XX_VIDDET_TEST_NO_SYNC_ERR = 0x1004e00a,
  SDVR_DIAG_TW28XX_AUDDET_TEST_NO_SYNC_ERR = 0x1004e00b,
  SDVR_DIAG_PCIE_EYEMASK_TEST_NO_CBB = 0x1005e001,
  SDVR_DIAG_PCIE_EYEMASK_TEST_ERR = 0x1005e002,
  SDVR_DIAG_PCIE_EYEMASK_TEST_TIMEOUT = 0x1005e003
} sdvr_diag_code_e;
```

Typedef for the board diagnostics codes.

`SDVR_DIAG_OK` - All the diagnostic tests passed.

**Diagnostics failure codes for boot loader DDR test**

`SDVR_DIAG_DDR_WRITEREAD_FAIL` - DDR write/read test failed.

`SDVR_DIAG_DDR_ADDRLINES_FAIL` - DDR address lines test failed.

`SDVR_DIAG_DDR_BITFLIP_FAIL` - DDR bit-flip test failed.

`SDVR_DIAG_DDR_DMA_FAIL` - DDR DMA test failed.

`SDVR_DIAG_DDR_READ_DMA_FAIL` - DDR read/DMA test failed.

**Diagnostics failure codes for PLL test**

`SDVR_DIAG_PLL_TEST_MHZ` - Processor speed test failed.

`SDVR_DIAG_PLL_TEST_SYS` - PLL_SYS test failed.

`SDVR_DIAG_PLL_TEST_IO` - PLL_IO test failed.

`SDVR_DIAG_PLL_TEST_AIM` - PLL_AIM test failed.

`SDVR_DIAG_PLL_TEST_DP0` - PLL_DP0 test failed.

`SDVR_DIAG_PLL_TEST_DP2` - PLL_DP2 test failed.

`SDVR_DIAG_PLL_TEST_DDR` - DLL_DDR test failed.

**Diagnostics failure codes for SPI Flash test**

`SDVR_DIAG_SPI_TEST_READ` - Flash read error.

`SDVR_DIAG_SPI_TEST_ERASE` - Flash erase error.

`SDVR_DIAG_SPI_TEST_PROG` - Flash program error.

`SDVR_DIAG_SPI_TEST_UNLOCK` - Flash unlock error.

SDVR_DIAG_SPI_TEST_COMPARE - Flash data compare error.

SDVR_DIAG_SPI_TEST_MAINT - Flash maintenance command error.

SDVR_DIAG_SPI_TEST_MISC - Miscellaneous Flash error.

**Diagnostics failure codes for TWI EEPROM test**

SDVR_DIAG_TWI_EEPROM_TEST_READ - TWI EEPROM read error.

SDVR_DIAG_TWI_EEPROM_TEST_WRITE - TWI EEPROM write error.

SDVR_DIAG_TWI_EEPROM_TEST_INIT - TWI EEPROM initialization error.

SDVR_DIAG_TWI_EEPROM_TEST_COMPARE - TWI EEPROM data compare error.

**Diagnostics failure codes for Epson test**

SDVR_DIAG_EPSON_REG_TEST_INIT - Epson test initialization error.

SDVR_DIAG_EPSON_REG_TEST_WALKING - Epson register bit-walk error.

**Diagnostics failure codes for Techwell test**

SDVR_DIAG_TW2815_AUDIO_TEST_INIT - Techwell audio test init error.

SDVR_DIAG_TW2815_AUDIO_TEST_NO_AUDIO - Techwell audio not received error.

SDVR_DIAG_TW2815_REG_TEST - Techwell register test error.

SDVR_DIAG_TW2815_VIDEO_TEST_INIT - Techwell video test init error.

SDVR_DIAG_TW2815_VIDEO_TEST_NO_VIDEO - Techwell video not received error.

SDVR_DIAG_TW28XX_VIDEO_TEST_TIMEOUT - Techwell video test timeout error.

SDVR_DIAG_TW28XX_VIDDET_TEST_INIT_ERR - Techwell video detect test init error.

SDVR_DIAG_TW28XX_VIDDET_TEST_UNKNOWN_CHIP - Techwell video detect test unknown chip error.

SDVR_DIAG_TW28XX_VIDDET_TEST_NO_INPUT_ERR - Techwell video detect test no input error.

SDVR_DIAG_TW28XX_VIDDET_TEST_CONFLICT_ERR - Techwell video detect test conflict error.

SDVR_DIAG_TW28XX_VIDDET_TEST_NO_SYNC_ERR - Techwell video detect test no sync error.

SDVR_DIAG_TW28XX_AUDDET_TEST_NO_SYNC_ERR - Techwell audio detect test no sync error.

**Diagnostics failure codes for PCIe test**

`SDVR_DIAG_PCIE_EYEMASK_TEST_NO_CBB` - PCIe did not detect the CBB test board.

`SDVR_DIAG_PCIE_EYEMASK_TEST_ERR` - PCIe eye mask test failure.

`SDVR_DIAG_PCIE_EYEMASK_TEST_TIMEOUT` - PCIe eye mask test timeout.

**sdvr_chan_handle_t**

```
typedef sx_int32 sdvr_chan_handle_t;
```

A handle to a channel.

**_sdvr_signals_type_e**

```
typedef enum _sdvr_signals_type_e {
  SDVR_SIGNAL_RUNTIME_ERROR = 1,
  SDVR_SIGNAL_FATAL_ERROR
  } __sdvr_signals_type_e;
```

This enumerated type defines the types of asynchronous messages that can be sent from the DVR firmware to the DVR Application.

`SDVR_SIGNAL_RUNTIME_ERROR` - Indicates that a non-fatal runtime error has occurred on the board. There is extra data associated with this signal that gives more information regarding the error. The meaning of those values varies depending on where the error occurred.

`SDVR_SIGNAL_FATAL_ERROR` - Indicates that a fatal error has occurred on the board. If this signal is received, the board must be reset. There is extra data associated with this signal that gives more information regarding the error. The meaning of those values varies depending on where the error occurred.

**sdvr_signals_type_e**

```
typedef sx_uint8 sdvr_signals_type_e;
```

Microsoft compiler work around, `sdvr_signals_type_e` cannot be enum because it is being used as :8 bit in the data structure.

**_sdvr_video_alarm_e**

```
typedef enum _sdvr_video_alarm_e {
  SDVR_VIDEO_ALARM_NONE,
  SDVR_VIDEO_ALARM_MOTION,
  SDVR_VIDEO_ALARM_BLIND,
  SDVR_VIDEO_ALARM_NIGHT,
  SDVR_VIDEO_ALARM_LOSS,
  SDVR_VIDEO_ALARM_DETECTED
} sdvr_video_alarm_e;
```

This enumerated type defines the types of events that can be detected by the video encoder.

`SDVR_VIDEO_ALARM_NONE` - No event or alarm.

`SDVR_VIDEO_ALARM_MOTION` - Motion detected.

`SDVR_VIDEO_ALARM_BLIND` - Blind detected.

`SDVR_VIDEO_ALARM_NIGHT` - Night detected.

SDVR_VIDEO_ALARM_LOSS - Video loss detected. If a channel is set to be an encoding channel, and no camera is connected to it, this video alarm is triggered whenever the encoder is enabled.

SDVR_VIDEO_ALARM_DETECTED - This alarm is sent when a video signal is detected on an encoding channel that was created but had no video signal previously.

**_sdvr_regions_type_e**

```
typedef enum _sdvr_regions_type_e {
  SDVR_REGION_MOTION,
  SDVR_REGION_BLIND,
  SDVR_REGION_PRIVACY
} sdvr_regions_type_e;
```

This enumerated type defines the types of regions that can be defined for a camera.

SDVR_REGION_MOTION - Motion detected region.

SDVR_REGION_BLIND - Blind detected region.

SDVR_REGION_PRIVACY - Privacy region.

**_sdvr_frame_type_e**

```
typedef enum _sdvr_frame_type_e {
  SDVR_FRAME_RAW_Y_VIDEO = 0,
  SDVR_FRAME_RAW_U_VIDEO,
  SDVR_FRAME_RAW_V_VIDEO,
  SDVR_FRAME_RAW_AUDIO,
  SDVR_FRAME_H264_IDR,
  SDVR_FRAME_H264_I,
  SDVR_FRAME_H264_P,
  SDVR_FRAME_H264_B,
  SDVR_FRAME_H264_SPS,
  SDVR_FRAME_H264_PPS,
  SDVR_FRAME_JPEG,
  SDVR_FRAME_G711,
  SDVR_FRAME_MPEG4_I,
  SDVR_FRAME_MPEG4_P,
  SDVR_FRAME_MPEG4_B,
  SDVR_FRAME_MPEG4_VOL,
  SDVR_FRAME_RAW_VIDEO = 100,
  SDVR_FRAME_VIDEO_ENCODED_PRIMARY,
  SDVR_FRAME_VIDEO_ENCODED_SECONDARY,
  SDVR_FRAME_AUDIO_ENCODED
} __sdvr_frame_type_e;
```

This enumerated type defines the kind of frames that can be exchanged between the SDK and the DVR Application.

SDVR_FRAME_RAW_VIDEO - Generic Raw video frame type. This is the type to pass to sdvr_get_yuv_buffer() to get raw video frames.

SDVR_FRAME_RAW_AUDIO - Generic Raw audio PCM frame. This is the type to pass to sdvr_get_av_buffer() to get raw audio frames.

SDVR_FRAME_H264_IDR - Encoded H.264 IDR frame.

SDVR_FRAME_H264_I - Encoded H.264 I frame.

SDVR_FRAME_H264_P - Encoded H.264 P frame.

SDVR_FRAME_H264_B - Encoded H.264 B frame.

SDVR_FRAME_H264_SPS - Encoded H.264 SPS frame.

SDVR_FRAME_H264_PPS - Encoded H.264 PPS frame.

SDVR_FRAME_JPEG - Encoded JPEG image frame.

SDVR_FRAME_G711 - Encoded G.711 audio frame.

SDVR_FRAME_MPEG4_I - Encoded MPEG4 I frame.

SDVR_FRAME_MPEG4_P - Encoded MPEG4 P frame.

SDVR_FRAME_MPEG4_VOL - Encoded MPEG4 VOL frame, which contains the video frame header information.

SDVR_FRAME_VIDEO_ENCODED_PRIMARY - Any encoded video frame from the primary encoder—this is the type to pass to sdvr_get_av_buffer() to get the primay encoded video frames.

SDVR_FRAME_VIDEO_ENCODED_SECONDARY - Any encoded video frame from the secondary encoder—this is the type to pass to sdvr_get_av_buffer() to get the secondary encoded video frames.

SDVR_FRAME_AUDIO_ENCODED - Any encoded audio frame—this is the type to pass to sdvr_get_av_buffer() to get an encoded audio frame.

**sdvr_frame_type_e**

```
typedef sx_uint8 sdvr_frame_type_e;
```

Microsoft compiler work around, sdvr_frame_type_e cannot be enum because it is being used as :8 bit in the data structure.

**_sdvr_location_e**

```
typedef enum _sdvr_location_e {
  SDVR_LOC_TOP_LEFT = 0,
  SDVR_LOC_BOTTOM_LEFT,
  SDVR_LOC_TOP_RIGHT,
  SDVR_LOC_BOTTOM_RIGHT,
  SDVR_LOC_CUSTOM
} sdvr_location_e;
```

Typedef describing locations for OSD.

SDVR_LOC_TOP_LEFT - Top left of the screen.

SDVR_LOC_TOP_RIGHT - Top right of the screen.

SDVR_LOC_BOTTOM_LEFT - Bottom left of the screen.

SDVR_LOC_BOTTOM_RIGHT - Bottom right of the screen.

SDVR_LOC_CUSTOM - A user-defined position. The upper left corner of the video frame is the origin (0,0). This location cannot be used with the sdvr_set_osd_text() function.

**_sdvr_video_std_e**

```
typedef enum _sdvr_video_std_e {
  SDVR_VIDEO_STD_NONE = 0,
  SDVR_VIDEO_STD_D1_PAL = (1 << 0),
  SDVR_VIDEO_STD_D1_NTSC = (1 << 1),
  SDVR_VIDEO_STD_CIF_PAL = (1 << 2),
  SDVR_VIDEO_STD_CIF_NTSC = (1 << 3),
  SDVR_VIDEO_STD_2CIF_PAL = (1 << 4),
  SDVR_VIDEO_STD_2CIF_NTSC = (1 << 5),
  SDVR_VIDEO_STD_4CIF_PAL = (1 << 6),
  SDVR_VIDEO_STD_4CIF_NTSC = (1 << 7),
  SDVR_VIDEO_STD_QCIF_PAL = (1 << 8),
  SDVR_VIDEO_STD_QCIF_NTSC = (1 << 9)
} sdvr_video_std_e;
```

This enumerated type describes the video standards supported by SDVR.

`SDVR_VIDEO_STD_NONE` - No standard defined.

`SDVR_VIDEO_STD_D1_PAL` - PAL 720x576 at 25 fps.

`SDVR_VIDEO_STD_D1_NTSC` - NTSC 720x480 at 30 fps.

`SDVR_VIDEO_STD_CIF_PAL` - PAL 352x288 at 25 fps.

`SDVR_VIDEO_STD_CIF_NTSC` - NTSC 352x240 at 30 fps.

`SDVR_VIDEO_STD_2CIF_PAL` - PAL 704x288 at 25 fps.

`SDVR_VIDEO_STD_2CIF_NTSC` - NTSC 704x240 at 30 fps.

`SDVR_VIDEO_STD_4CIF_PAL` - PAL 704x576 at 25 fps.

`SDVR_VIDEO_STD_4CIF_NTSC` - NTSC 704x480 at 30 fps.

`SDVR_VIDEO_STD_QCIF_PAL` - PAL 176x144 at 25 fps.

`SDVR_VIDEO_STD_QCIF_NTSC` - NTSC 176x120 at 30 fps. **NOTE**: This is not the standard QCIF size 176x120.

**_sdvr_video_size_e**

```
typedef enum _sdvr_video_size_e {
  SDVR_VIDEO_SIZE_720x576 = (1 << 0),
  SDVR_VIDEO_SIZE_720x480 = (1 << 1),
  SDVR_VIDEO_SIZE_352x288 = (1 << 2),
  SDVR_VIDEO_SIZE_352x240 = (1 << 3),
  SDVR_VIDEO_SIZE_704x288 = (1 << 4),
  SDVR_VIDEO_SIZE_704x240 = (1 << 5),
  SDVR_VIDEO_SIZE_704x576 = (1 << 6),
  SDVR_VIDEO_SIZE_704x480 = (1 << 7),
  SDVR_VIDEO_SIZE_176x144 = (1 << 8),
  SDVR_VIDEO_SIZE_176x112 = (1 << 9)
} sdvr_video_size_e;
```

This enumerated type describes the supported decoding video sizes. It is needed when creating a decoder channel.

`SDVR_VIDEO_SIZE_720x576` - D1-PAL video width of 720 and number of lines of 576.

`SDVR_VIDEO_SIZE_720x480` - D1-NTSC video width of 720 and number of lines of 480.

SDVR_VIDEO_SIZE_352x288 - CIF-PAL video width of 352 and number of lines of 288.

SDVR_VIDEO_SIZE_352x240 - CIF-NTSC video width of 352 and number of lines of 240.

SDVR_VIDEO_SIZE_704x288 - 2CIF-PAL video width of 704 and number of lines of 288.

SDVR_VIDEO_SIZE_704x240 - 2CIF-NTSC video width of 704 and number of lines of 240.

SDVR_VIDEO_SIZE_704x576 - 4CIF-PAL video width of 704 and number of lines of 576.

SDVR_VIDEO_SIZE_704x480 - 4CIF-NTSC video width of 704 and number of lines of 480.

SDVR_VIDEO_SIZE_176x144 - QCIF_PAL video width of 176 and number of lines of 144.

SDVR_VIDEO_SIZE_176x112 - QCIF_NTSC video width of 176 and number of lines of 112. **NOTE**: This is not the standard QCIF size 176x120.

**_sdvr_chan_type_e**

```
typedef enum _sdvr_chan_type_e {
  SDVR_CHAN_TYPE_NONE = 255,
  SDVR_CHAN_TYPE_ENCODER = 0,
  SDVR_CHAN_TYPE_DECODER = 2
} __sdvr_chan_type_e;
```

This enumerated type describes the kinds of channels supported by SDVR. To create a channel that only allows it to be used in HMO or SMO, you must use SDVR_CHAN_TYPE_ENCODER, and set the encoder type to SDVR_VIDEO_ENC_NONE.

SDVR_CHAN_TYPE_NONE - Channel type not specified.

SDVR_CHAN_TYPE_ENCODER - Encoder channel.

SDVR_CHAN_TYPE_DECODER - Decoder channel.

**sdvr_chan_type_e**

```
typedef sx_uint8 sdvr_chan_type_e;
```

Microsoft compiler work around, sdvr_chan_type_e cannot be enum because it is being used as :8 bit in the data structure.

**_sdvr_vpp_mode_e**

```
enum _sdvr_vpp_mode_e {
  SDVR_VPP_MODE_SLATERAL = 0,
  SDVR_VPP_MODE_ANALYTICS = 1
} ;
```

Enumerated type describing video preprocessing (VPP) modes.

SDVR_VPP_MODE_ANALYTICS Run VPP in analytics mode.

SDVR_VPP_MODE_SLATERAL Run VPP in Stretch-lateral-filter mode.

**sdvr_vpp_mode_e**

```
typedef sx_uint8 sdvr_vpp_mode_e;
```

Microsoft compiler work around, `sdvr_vpp_mode_e` cannot be enum because it is being used as :8 bit in the data structure.

**_sdvr_sub_encoders_e**

```
typedef enum _sdvr_sub_encoders_e {
  SDVR_ENC_PRIMARY,
  SDVR_ENC_SECONDARY
} sdvr_sub_encoders_e;
```

Enumerated type describing encoder subchannels supported by the SDVR.

`SDVR_ENC_PRIMARY` - The primary encoder

`SDVR_ENC_SECONDARY` - The secondary encoder

**_sdvr_venc_e**

```
typedef enum _sdvr_venc_e {
  SDVR_VIDEO_ENC_NONE,
  SDVR_VIDEO_ENC_H264,
  SDVR_VIDEO_ENC_JPEG,
  SDVR_VIDEO_ENC_MPEG4
} sdvr_venc_e;
```

Enumerated type describing video encoders supported by the SDVR.

`SDVR_VIDEO_ENC_NONE` - No video encoder specified

`SDVR_VIDEO_ENC_H264` - H.264 encoder

`SDVR_VIDEO_ENC_JPEG` - Motion JPEG encoder

`SDVR_VIDEO_ENC_MPEG4` - MPEG4 encoder

**_sdvr_aenc_e**

```
typedef enum _sdvr_aenc_e {
  SDVR_AUDIO_ENC_NONE,
  SDVR_AUDIO_ENC_G711,
  SDVR_AUDIO_ENC_G726_16K,
  SDVR_AUDIO_ENC_G726_24K,
  SDVR_AUDIO_ENC_G726_32K,
  SDVR_AUDIO_ENC_G726_48K
} sdvr_aenc_e;
```

Enumerated type describing audio encoders supported by the SDVR.

`SDVR_AUDIO_ENC_NONE` - No audio encoder specified

`SDVR_AUDIO_ENC_G711` - G.711 audio encoder

`SDVR_AUDIO_ENC_G726_16` - G.726 audio encoder at 16K bits/sec

`SDVR_AUDIO_ENC_G726_24` - G.726 audio encoder at 24K bits/sec

`SDVR_AUDIO_ENC_G726_32` - G.726 audio encoder at 32K bits/sec

`SDVR_AUDIO_ENC_G726_48` - G.726 audio encoder at 48K bits/sec

**_sdvr_video_res_decimation_e**

```
typedef enum _sdvr_video_res_decimation_e {
  SDVR_VIDEO_RES_DECIMATION_NONE,
  SDVR_VIDEO_RES_DECIMATION_EQUAL = 1,
  SDVR_VIDEO_RES_DECIMATION_FOURTH = 2,
```

```
  SDVR_VIDEO_RES_DECIMATION_SIXTEENTH = 4
} sdvr_video_res_decimation_e;
```

Enumerated type describing the various encoding and display resolution decimation.

When configuring the DVR, the DVR Application sets the maximum system-wide resolution. The video resolution for a particular channel is described in terms of this maximum resolution. The video resolution for a channel can be the same as, 1/4 of, or 1/16 of the maximum system-wide resolution.

`SDVR_VIDEO_RES_DECIMATION_NONE` - No resolution set.

`SDVR_VIDEO_RES_DECIMATION_EQUAL` - Same resolution as the maximum.

`SDVR_VIDEO_RES_DECIMATION_FOURTH` - 1/4 of the maximum resolution.

`SDVR_VIDEO_RES_DECIMATION_SIXTEENTH` - 1/16 of the maximum resolution.

### _sdvr_br_control_e

```
typedef enum _sdvr_br_control_e {
  SDVR_BITRATE_CONTROL_NONE = 255,
  SDVR_BITRATE_CONTROL_VBR = 0,
  SDVR_BITRATE_CONTROL_CBR,
  SDVR_BITRATE_CONTROL_CQP,
  SDVR_BITRATE_CONTROL_CONSTANT_QUALITY
} sdvr_br_control_e;
```

Enumerated type describing various bit rate control schemes available in the SDVR.

`SDVR_BITRATE_CONTROL_NONE` - No bit rate control.

`SDVR_BITRATE_CONTROL_VBR` - Variable bit rate.

`SDVR_BITRATE_CONTROL_CBR` - Constant bit rate.

`SDVR_BITRATE_CONTROL_CQP` - Quantization parameter - Not Supported.

`SDVR_BITRATE_CONTROL_CONSTANT_QUALITY` - Constant Quality bit rate.

### _sdvr_dts_style_e

```
typedef enum _sdvr_dts_style_e {
  SDVR_OSD_DTS_NONE = 0,
  SDVR_OSD_DTS_DEBUG,
  SDVR_OSD_DTS_MDY_12H,
  SDVR_OSD_DTS_DMY_12H,
  SDVR_OSD_DTS_YMD_12H,
  SDVR_OSD_DTS_MDY_24H,
  SDVR_OSD_DTS_DMY_24H,
  SDVR_OSD_DTS_YMD_24H
} sdvr_dts_style_e;
```

Enumerated type describing the date and time display styles supported by the SDVR.

SDVR_OSD_DTS_NONE - No date and time is displayed after the OSD text.

SDVR_OSD_DTS_DEBUG - Enables a special debug display mode.

SDVR_OSD_DTS_MDY_12H - Displays the date in Month/Day/Year format followed by time in HH:MM:SS am/pm format.

SDVR_OSD_DTS_DMY_12H - Displays the date in Day/Month/Year format followed by time in HH:MM:SS am/pm format.

SDVR_OSD_DTS_YMD_12H - Displays the date in Year/Month/Day format followed by time in HH:MM:SS am/pm format.

SDVR_OSD_DTS_MDY_24H - Displays the date in Month/Day/Year format followed by time in~~in~~ 24 hour HH:MM:SS format.

SDVR_OSD_DTS_DMY_24H - Displays the date in Day/Month/Year format followed by time in 24 hour HH:MM:SS format.

SDVR_OSD_DTS_YMD_24H - Displays the date in Year/Month/Day format followed by time in 24 hour HH:MM:SS format.

**_sdvr_board_e**

```
typedef enum _sdvr_board_e {
   SDVR_BOARD_REV_UNKNOWN = -1,
   SDVR_S6D1X16_BOARD_REV_0 = 0,
   SDVR_S6D1X16_BOARD_REV_1,
   SDVR_S6D1X16_BOARD_REV_2,
   SDVR_S6D1X16_BOARD_REV_3
} sdvr_board_e;
```

This enumerated type describes the types of SDVR boards available from Stretch.

SDVR_S6D1X16_BOARD_REV_0 - This board has the following limitations and capabilities:

- 16 channels of D1 encoding
- 16 decoding supported
- No spot monitor supported

SDVR_S6D1X16_BOARD_REV_1 SDVR_S6D1X16_BOARD_REV_2 SDVR_S6D1X16_BOARD_REV_3 These are the production boards with the following capabilities:

- 16 channels of D1 encoding without full SMO support. Has analog SMO support.
- 15 channels of D1 encoding with full SMO support
- 16 decoding supported
- 16 audio supported

**_sdvr_chip_rev_e**

```
typedef enum _sdvr_chip_rev_e {
  SDVR_CHIP_S6100_3_REV_C = 0,
  SDVR_CHIP_S6105_3_REV_C,
  SDVR_CHIP_S6106_3_REV_C,
  SDVR_CHIP_S6100_3_REV_D = 16,
```

```
  SDVR_CHIP_S6105_3_REV_D,
  SDVR_CHIP_S6106_3_REV_D,
  SDVR_CHIP_S6100_3_REV_F = 32,
  SDVR_CHIP_S6105_3_REV_F,
  SDVR_CHIP_S6106_3_REV_F,
  SDVR_CHIP_S6100_3_UNKNOWN = 48,
  SDVR_CHIP_S6105_3_UNKNOWN,
  SDVR_CHIP_S6106_3_UNKNOWN,
  SDVR_CHIP_UNKNOWN = 255
} sdvr_chip_rev_e;
```

Chip revision definitions.

**_sdvr_board_attrib_t**

```
typedef struct _sdvr_board_attrib_t {
  sx_uint32 pci_slot_num;
  sx_uint32 board_type;
  sx_uint32 supported_video_stds;
  sdvr_chip_rev_e chip_revision;
  sdvr_board_e board_revision;
} sdvr_board_attrib_t;
```

This data structure holds board attributes.

*pci_slot_num* - The PCI slot number in which the board is located.

*board_type* - The SDVR board type.

*supported_video_stds* - Specifies what cameras are supported by the board. It is a bit-wise OR of video standards, such as `SDVR_VIDEO_STD_D1_PAL` or `SDVR_VIDEO_STD_D1_NTSC`.

*chip_revision* - The Stretch chip number and revision. Refer to `sdvr_chip_rev_e` for a list of Stretch chip numbers.

*board_revision* - The board revision.

**_sdvr_pci_attrib_t**

```
typedef struct _sdvr_pci_attrib_t {
  sx_uint32 pci_slot_num;
  sx_uint32 board_type;
  sx_uint16 vendor_id;
  sx_uint16 device_id;
  sx_uint16 subsystem_vendor;
  sx_uint16 subsystem_id;
  sx_uint8 serial_number[SDVR_BOARD_SERIAL_LENGTH + 1];
} sdvr_pci_attrib_t;
```

This data structure holds PCIe board attributes. You get these attributes by calling `sdvr_get_pci_attrib`(). This function can be called before or after loading of the firmware into the DVR board.

*pci_slot_num* - The PCI slot number in which the board is located.

*board_type* - The board type, which is combination of *device_id* and the *subsystem_vendor.*

*vendor_id* - Always Stretch (`0x18A2`)

*device_id* - Board ID per each vendor.

*subsystem_vendor* - Vendor ID. For Stretch boards it is (`0x18A2`).

*subsystem_id* - Currently is always set to zero.

*serial_number* - A null terminated serial number string.

**_sdvr_sdk_params_t**

```
typedef struct _sdvr_sdk_params_t {
  sx_uint32 enc_buf_num;
  sx_uint32 raw_buf_num;
  sx_uint32 dec_buf_num;
  sx_uint32 dec_buf_size;
  sx_uint32 timeout;
  sx_uint32 debug_flag;
  char *debug_file_name;
} sdvr_sdk_params_t;
```

This data structure is used to hold SDK configuration parameters.

To exchange encoded and raw video with the board, the SDK needs to allocate buffers to hold the data on its way to and from the DVR Application and the board. The number of buffers to be allocated on each of these paths and the sizes of these buffers are set using this data structure.

We recommend that you use the default values for each frame buffer as if they are optimized for streaming at 30 fps for NTSC, and 25fps for PAL video standards.

*enc_buf_num* - The number of buffers to be allocated for each encoder channel. Each encoder channel will have the same number of buffers allocated. It is important that you allocate enough buffers for encoded frames to be held between the times you can process them. Maximum number of encoder buffers is 40. The default value is 20.

**NOTE:** The size of each buffer is determined by the firmware.

*raw_buf_num* - The number of buffers to be allocated for each channel that will be sending raw video to the DVR Application. Each channel sending raw video will have the same number of buffers. Typically, you will need to display raw frames 30 times per second. Therefore, you only need between 2 and 4 buffers to hold raw video. Maximum number of raw buffers is 5. The default value is 2.

**NOTE:** The size of each buffer is determined by the firmware.

*dec_buf_num* - The number of buffers to be allocated for each decoder channel. Each decoder channel will have the same number of buffers allocated. The maximum number of decoder buffers allowed is 5. The default value is 5.

*dec_buf_size* - The size of each buffer used to hold encoded frames on the way to the decoder hardware. Typically, this is the size of the largest encoded

frame that needs to be decoded. The same buffer size is used across all decoder channels. The default value is 414720.

*timeout* - For a variety of reasons, it is possible for the board to hang. By setting the timeout value, you specify when the SDK will give up on a response from the board and inform the DVR Application that the board is hung. The value of this parameter is in seconds. A value of 0 indicates that there is no timeout and the SDK will wait indefinitely for the board to respond. Do not set the value of timeout too low, or during times of heavy traffic on the bus, you might get a false warning that the board has hung. Setting the value to 0 (no timeout), may cause the PC to hang if the firmware on the board dies. The default value is 5 seconds.

*debug_flag* - This is a bit field of flags that can be set to enable various levels of debugging as defined by the debug flags. See the defines for `DEBUG_FLAG_`*xxx* for the definition of each field. Setting debugging flags has a noticeable effect on system performance. The default value is zero.

*debug_file_name* - The name of the file where the debug information is stored. This string should include the full path name to the file, or a file in the current working directory (depends on the OS as to how this is defined) is created. If the file does not exist, it is created. If it already exists, it will be truncated. You also must enable the `DEBUG_FLAG_WRITE_TO_FILE` bit in the *debug_flag* field to save the debugging information into the file. In addition to the given specified *debug_file_name*, which includes all the SDK tracing, a new file will be created with **`_fw`** appended to the *debug_file_name*, which contains all the low level commands sent to the DVR firmware.

**_sdvr_board_config_t**

```
typedef struct _sdvr_board_config_t {
  sx_uint32 num_cameras_supported;
  sx_uint32 num_microphones_supported;
  sx_bool has_smo;
  sx_uint32 num_sensors;
  sx_uint32 num_relays;
  sdvr_video_std_e camera_type;
  sx_uint32 num_decoders_supported;
} sdvr_board_config_t;
```

This data structure defines the capabilities of the SDVR boards.

*num_cameras_supported* - Number of cameras supported by the board, i.e., the number of physical camera connectors on the board.

*num_microphones_supported* - Number of microphones supported by the board, i.e., the number of physical microphone connectors on the board.

*has_smo* - If true, this board has a spot monitor output.

*num_sensors* - Number of sensors on this board.

*num_relays* - Number of relays on this board.

*camera_type* - The maximum resolution that was specified by the DVR Application when connecting to the board. Typically, all cameras connected to the DVR have this resolution, hence the name of the field.

*num_decoders_supported* - The number of decoding channels supported by the board. This value is zero if the board does not support decoding.

**_sdvr_firmware_ver_t**

```
typedef struct _sdvr_firmware_ver_t {
  sx_uint8 fw_major;
  sx_uint8 fw_minor;
  sx_uint8 fw_revision;
  sx_uint8 fw_build;
  sx_uint16 fw_build_year;
  sx_uint8 fw_build_month;
  sx_uint8 fw_build_day;
  sx_uint8 bootloader_major;
  sx_uint8 bootloader_minor;
  sx_uint8 bsp_major;
  sx_uint8 bsp_minor;
} sdvr_firmware_ver_t;
```

This data structure defines the firmware, boot loader and BSP version, and build information.

Stretch follows the convention of using four numbers for version control. A change in the major number indicates major changes to functionality, a change in the minor number indicates minor changes to functionality, and a change in the revision number indicates significant bug fixes that were introduced in the minor change functionality. A change to the build number indicates only bug fixes that do not change functionality.

*fw_major* - The firmware major version number. A change in this field indicates major changes to functionality.

*fw_minor* - The firmware minor version number. A change in this field indicates minor changes to functionality.

*fw_revision* - The firmware revision version number. A change in this field indicates significant bug fixes that were introduced in the minor change functionality.

*fw_build* - The firmware build version number. A change in this field indicates only bug fixes that do not change functionality.

*fw_build_year* - The date of firmware build.

*fw_build_month* - The date of firmware build.

*fw_build_day* - The date of firmware build.

*bootloader_major* - The major version number of boot loader.

*bootloader_minor* - The minor version number of boot loader.

*bsp_major* - The major version number of BSP.

*bsp_minor* - The minor version number of BSP.

**sdvr_signal_info**

```
typedef struct sdvr_signal_info {
  sdvr_signals_type_e sig_type;
  sdvr_chan_type_e chan_type;
  sx_uint8 chan_num;
  sx_uint8 reserved1;
  sx_uint32 data;
  sx_uint32 extra_data;
  sx_uint32 reserved3;
} sdvr_signal_info_t;
```

The DVR firmware sends asynchronous signal messages to the DVR Application as it encounters errors not related to any direct function call. This data structure defines parameters associated with the asynchronous signals sent from the DVR firmware to the host DVR Application.

*sig_type* - The type identifying the signal cause.

*chan_type* - The type of channel causing the signal.

*chan_num* - The ID of channel causing the signal.

*data* - The error code associated with the `sig_type`.

*extra_data* - Optional data information associated with the `signal_data`.

**_sdvr_chan_def_t**

```
typedef struct _sdvr_chan_def_t {
  sx_uint8 board_index;
  sx_uint8 chan_num;
  sdvr_chan_type_e chan_type;
  sdvr_venc_e video_format_primary;
  sdvr_aenc_e audio_format;
  sdvr_venc_e video_format_secondary;
  sdvr_video_size_e video_size;
  sdvr_vpp_mode_e vpp_mode;
} sdvr_chan_def_t;
```

This data structure defines parameters that are needed to create a new encoder, decoder, or HMO-only channel.

*board_index* - The zero-based index of the board where this channel resides.

*chan_num* - The channel number identifier. In the case of encoding, the range is 0 to M-1, where M is the number of cameras supported by the board that are designated encoding channels. In the case of decoding, the range is 0 to N-1, where N is the number of decoders supported by the board that are designated decoding channels.

**NOTE:** The channel number specifies on which S6 chip the channel is going to be created. The first set of four (0 - 3) is created on the first S6 chip, the second set of four (4 - 7) on the second S6 chip, and so on.

*chan_type* - The type channel to create as specified in `sdvr_chan_type_e`, i.e., encoder or decoder.

**NOTE:** To create an HMO- or SMO-only channel, set the chan_type to SDVR_CHAN_TYPE_ENCODER and the video_format_primary to SDVR_VIDEO_ENC_NONE.

*video_format_primary* - The primary encode or decode video format, e.g., H.264

*audio_format* - The encode or decode audio format, e.g., G.711. If no audio is associated with this channel, you can specify SDVR_AUDIO_ENC_NONE. This field is ignored in version 1.0.

*video_format_secondary* - The secondary encode video format, e.g., H.264 This field is ignored when creating a decoder channel, or if the primary video format is SDVR_VIDEO_ENC_NONE.

**NOTE:** Specifying a secondary encoder on a camera takes away processing power from other channels. As result you may not be able to create some encoder channels if secondary encoding is used in all the encoder channels.

*video_size* - This enum specifies the size of video to be decoded. It is ignored if the channel type is SDVR_CHAN_TYPE_ENCODER.

*vpp_mode* - This field specifies whether to run VPP in analytics or Stretchlateral-filter mode for an encoder channel. Blind detection, privacy blocking, and old style OSD (black background, only English characters) are disabled in SDVR_VPP_MODE_SLATERAL *vpp_mode*, but new style OSD (transparent background and Unicode UTF-16 coding scheme supporting Basic Multilingual Plane (BMP)) is supported and encoder quality is higher. In SDVR_VPP_MODE_ANALYTICS, only old style OSD is supported.

**_sdvr_video_enc_chan_params_t**

```
typedef struct _sdvr_video_enc_chan_params_t {
  sx_uint8 frame_rate;
  sx_uint8 res_decimation;
  sx_uint16 reserved1;
  union {
    struct {
       sx_uint16 avg_bitrate;
       sx_uint16 max_bitrate;
       sx_uint8 bitrate_control;
       sx_uint8 gop;
       sx_uint8 quality;
       sx_uint8 reserved1;
    } h264;
    struct {
       sx_uint16 quality;
       sx_uint8 is_image_style;
       sx_uint8 reserved1;
       sx_uint32 reserved2;
    } jpeg;
    struct {
       sx_uint16 avg_bitrate;
       sx_uint16 max_bitrate;
```

```
        sx_uint8 bitrate_control;
        sx_uint8 gop;
        sx_uint8 quality;
        sx_uint8 reserved1;
    } mpeg4;
  } encoder;
} sdvr_video_enc_chan_params_t;
```

This data structure defines video encoder parameters. For each channel used for encoding, use this data structure to set its parameters.

*frame_rate* - The frame rate of the channel in frames per second. Valid values are 1-30 for NTSC video and 1-25 for PAL. The default is 30 for NTSC and 25 for PAL.

*res_decimation* - Resolution decimation of the channel. You can specify the resolution of the encoded channel to be the same as, 1/4, or 1/16 of the system-wide maximum resolution. The default is `SDVR_VIDEO_RES_DECIMATION_EQUAL`.

The parameters for each encoder are set in the following union.

**H.264 Parameters**

*avg_bitrate* - The average bit rate in Kbits per second if CBR or VBR is selected. The default is 2000.

*max_bitrate* - The maximum target bit rate in Kbits per second if VBR is selected. The default is 4000.

*bitrate_control* - Choose one of CBR, VBR, QP (not supported), or constant quality. The default is CBR.

*gop_size* - GOP size for the H.264 encoder. GOP size must be greater than zero. The default is 15.

*quality* - A number between 0 and 100 to control the quality to be maintained while `bitrate_control` is set to constant quality. The default is 50.

**JPEG Parameters**

*quality* - A number in the range of 10 - 300 to control the quality of the compression. A higher number implies better quality. The default is 50.

*is_image_style* - The JPEG encoder generates an image-style JPEG frame header to be used for RTP when this value is 1. The JPEG encoder generates a video-style JPEG (Motion JPEG) frame header when this value is zero. This is suitable in AVI MJPEG or still JPEG image files. This field must always be 0. The default is 0.

**MPEG4 Parameters**

*avg_bitrate* - The average bit rate in Kbits per second if CBR or VBR is selected. The default is 2000.

*max_bitrate* - The maximum target bit rate in Kbits per second if VBR is selected. The default is 4000.

*bitrate_control* - Choose one of CBR, VBR, QP (not supported), or constant quality. The default is VBR.

*gop_size* - GOP size for the MPEG4 encoder. GOP size must be greater than zero. The default is 15.

*quality* - A number between 0 and 100 to control the quality to be maintained while bitrate_control is set to constant quality. The default is 50.

**_sdvr_alarm_video_enc_params_t**

```
typedef struct _sdvr_alarm_video_enc_params_t {
    sx_uint8 frame_rate;
    sx_uint8 min_on_seconds;
    sx_uint8 min_off_seconds;
    sx_uint8 enable;
    union {
        struct {
            sx_uint16 avg_bitrate;
            sx_uint16 max_bitrate;
            sx_uint8 bitrate_control;
            sx_uint8 gop;
            sx_uint8 quality;
            sx_uint8 reserved1;
        } h264;
        struct {
            sx_uint16 quality;
            sx_uint8 is_image_style;
            sx_uint8 reserved1;
            sx_uint32 reserved2;
        } jpeg;
        struct {
            sx_uint16 avg_bitrate;
            sx_uint16 max_bitrate;
            sx_uint8 bitrate_control;
            sx_uint8 gop;
            sx_uint8 quality;
            sx_uint8 reserved1;
        } mpeg4;
    } encoder;
} sdvr_alarm_video_enc_params_t;
```

This data structure defines video encode channel parameters for alarm video streaming. After any of the alarms reaches its specified threshold, the video encoded frame is streamed using these new parameters for the given minimum duration.

**NOTE:** Currently these parameters are used for all triggered alarms.

*frame_rate* - The new on-alarm recording frame rate of the channel in frames per second. Valid values are 1-30 for NTSC video and 1-25 for PAL. The default is 30 for NTSC and 25 for PAL.

*min_on_seconds* - Minimum number of seconds to stream using the new encoder parameter after the alarm is triggered.

*min_off_seconds* - Minimum number of quiet periods between each alarm streaming condition.

*enable* - A flag to enable or disable on-alarm video streaming. 0 turns off the on-alarm streaming. 1 turns on the on-alarm streaming.

*encoder* - The encoder-specific parameters. See the encoder union in `sdvr_video_enc_chan_params_t` for a detailed description.

**_sdvr_audio_enc_chan_params_t**

```
typedef struct _sdvr_audio_enc_chan_params_t {
    sdvr_aenc_e audio_enc_type;
} sdvr_audio_enc_chan_params_t;
```

Structure defining audio encoder channel parameters.

*audio_enc_type* - The type of audio encoder to use.

**_sdvr_region_t**

```
typedef struct _sdvr_region_t {
    sx_uint8 region_id;
    sx_uint16 upper_left_x;
    sx_uint16 upper_left_y;
    sx_uint16 lower_right_x;
    sx_uint16 lower_right_y;
} sdvr_region_t;
```

Data structure for a region. A region is specified by its upper left and lower right coordinates in pixels. The upper left corner of an image is the origin (0,0).

*region_id* - The region identifier. It is needed to change or remove a region. When adding a new region, this field will be set by the system.

*upper_left_x* - X-coordinate of the upper left corner.

*upper_left_y* - Y-coordinate of the upper left corner.

*lower_right_x* - X-coordinate of the lower right corner.

*lower_right_y* – Y- coordinate of the lower right corner.

**_sdvr_motion_detection**

```
typedef struct _sdvr_motion_detection {
    sx_uint8 threshold;
    sx_uint8 enable;
    sx_uint8 num_regions;
    sdvr_region_t regions[SDVR_MAX_MD_REGIONS];
} sdvr_motion_detection_t;
```

Data structure for motion detection alarm.

*threshold* - The threshold value for motion detection. Motion above the threshold is reported. A threshold of zero means the motion detection alarm is triggered constantly. A threshold of 99 disables the motion detection. The valid range is 0 - 99. The default is 20.

*enable* - A value of 1 enables motion detection for all the specified regions. If no region is defined, the entire picture will be used for motion detection. A value of 0 disables the motion detection.

*num_regions* - This field specifies the number of motion detection regions to be added to the current video channel. A value of 0 means motion detection is applied to the entire picture. This is a read-only field.

*regions* - An array of regions definition. The regions in this array are not in any order. Each array item has an ID to identify the region. This is a read-only field.

**_sdvr_blind_detection**

```
typedef struct _sdvr_blind_detection {
    sx_uint8 threshold;
    sx_uint8 enable;
    sx_uint8 num_regions;
    sdvr_region_t regions[SDVR_MAX_BD_REGIONS];
} sdvr_blind_detection_t;
```

Data structure for blind detection alarm.

*threshold* - The threshold value for blind detection. Blinding above the threshold is reported. Setting this value to 99 disables blind detection. The valid range is 0 - 99. The default is 60.

*enable* - A value of 1 enables blind detection for all the specified regions. If no region is defined, the entire picture will be used for blind detection. A value of 0 disables the blind detection.

*num_regions* - This field specifies the number of blind detection regions added to the current video channel. A value of 0 means blind detection is applied to the entire picture. This is a read-only field.

*regions* - An array of regions definition. The regions in this array are not in any order. Each array item has an ID to identify the region. This is a read-only field.

**_sdvr_night_detection**

```
typedef struct _sdvr_night_detection {
    sx_uint8 threshold;
    sx_bool enable;
} sdvr_night_detection_t;
```

Data structure for night detection alarm.

*night_detect_threshold* - The threshold value for night detection. Values below the threshold are reported. Setting this value to 255 disables night detection. The valid range is 0 - 255. The default is 40.

*enable* - A value of 1 enables night detection. A value of 0 disables the night detection.

**_sdvr_privacy_region**

```
typedef struct _sdvr_privacy_region {
    sx_uint8 enable;
    sx_uint8 num_regions;
    sdvr_region_t regions[SDVR_MAX_PR_REGIONS];
} sdvr_privacy_region_t;
```

Data structure for privacy regions.

*enable* - A value of 1 enables block out for all the specified regions. If no region is defined, the entire picture will be blocked out. A value of 0 turns off privacy.

*num_regions* - This field specifies the number of blocked out regions added to the current video channel. A value of 0 means the entire picture will be blocked out. This is a read-only field.

*regions* - An array of regions definition. The regions in this array are not in any order. Each array item has an ID to identify the region. This is a read-only field.

**_sdvr_av_buffer_t**

```
typedef struct _sdvr_av_buffer_t {
    sx_uint8 board_id;
    sdvr_chan_type_e channel_type;
    sx_uint8 channel_id;
    sdvr_frame_type_e frame_type;
    sx_uint8 motion_detected;
    sx_uint8 blind_detected;
    sx_uint8 night_detected;
    sx_uint8 av_state_flags;
    sx_uint8 stream_id;
    sx_uint8 reserved1;
    sx_uint16 reserved2;
    sx_uint32 payload_size;
    sx_uint32 timestamp;
    sx_uint32 timestamp_high;
    sx_uint32 reserved4;
    sx_uint32 reserved5;
    sx_uint8 payload[1];
} sdvr_av_buffer_t;
```

The AV buffer structure used in the SDK for A/V frames.

The AV buffer has a header followed by the payload. The header contains the following information:

*board_id* - The board ID from where the frame was received.

*channel_type* - The type of channel (encoder or decoder).

*channel_id* - The channel ID.

*frame_type* - The type of video or audio frame associated with this buffer.

*motion_detected* - The motion value detected on this channel. You must compare this value against the motion threshold to decide if there are any motions on this video frame.

*blind_detected* - The blind value on this channel. You must compare this value against the blind threshold to decide if the camera is blinded.

*night_detected* - The night value on this channel. You must compare this value against the night threshold to decide if night is detected.

*av_state_flags* - A set of one-bit flags about the audio/video state of the camera. The loss occurs if the corresponding bit is set. The possible flags are `SDVR_AV_STATE_VIDEO_LOST` and `SDVR_AV_STATE_AUDIO_LOST`.

*stream_id* - This field is only used for encoded frames. It indicates whether the payload corresponds to a video frame from the primary (0) or the secondary (1) encoder on this encoded channel.

*payload_size* - The size of the payload, in bytes, that follows the header.

*timestamp* - A hardware-generated time stamp of when the frame was captured. Low 32 bits of hardware-generated time stamp.

*timestamp_high* - A hardware-generated time stamp of when the frame was captured. High 32 bits of hardware-generated time stamp.

**NOTE:** This field is valid only in firmware build versions 3.2.0.1 or later.

For video, the time stamp is generated using a 100 KHz clock, and can be used for A/V synchronization.

For audio, the time stamp is generated using an 100 KHz clock, and can be used for A/V synchronization.

In addition to the header, there is the payload that contains the data for the frame.

We highly recommend that when storing each encoded frame to disk, both the payload and the header be stored. The header provides information that can be useful when searching through the file.

Additionally, we recommend that the DVR Application create a separate data and tag file for each channel. The data file contains the payload, and the tag file contains the header information. This makes searching stored video a lot faster.

The payload field points to the beginning of data for a frame. If you have a pointer to an `sdvr_av_buffer_t` object (call this pointer p), then you can access the payload by:

```
... = (int) p->payload[0]; // First word of buffer
... = (int) p->payload[1]; // Second word of buffer
```

**_sdvr_yuv_buffer_t**

```
typedef struct _sdvr_yuv_buffer_t {
    sx_uint8 board_id;
    sdvr_chan_type_e channel_type;
    sx_uint8 channel_id;
    sdvr_frame_type_e frame_type;
    sx_uint8 motion_detected;
    sx_uint8 blind_detected;
    sx_uint8 night_detected;
    sx_uint8 reserved1;
    sx_uint32 timestamp;
    sx_uint32 y_data_size;
```

```
    sx_uint32 u_data_size;
    sx_uint32 v_data_size;
    sx_uint8 *y_data;
    sx_uint8 *u_data;
    sx_uint8 *v_data;
} sdvr_yuv_buffer_t;
```

The YUV buffer structure used for holding a raw video frame. The structure contains three pointers to the Y, U, and V parts of a raw video frame. The raw video is in YUV 4:2:0 format.

The AV buffer has a header followed by the payload. The header contains the following information:

*board_id* - The board ID from where the frame was received.

*channel_type* - The type of channel (encoder or decoder).

*channel_id* - The channel ID.

*frame_type* - The type of video/audio frame associated with this buffer.

*motion_detected* - The motion value detected on this channel. You must compare this value against the motion threshold to decide if there are any motions on this video frame.

*blind_detected* - The blind value on this channel. You must compare this value against the blind threshold to decide if the camera is blinded.

*night_detected* - The night value on this channel. You must compare this value against the night threshold to decide if night is detected.

*reserved1* - Reserved for future use. This field should not be used.

*y_buff_size* - The size of the `y_buffer` in bytes that follows the header.

*u_buff_size* - The size of the `u_buffer` in bytes that follows the header.

*v_buff_size* - The size of the `v_buffer` in bytes that follows the header.

*timestamp* - A hardware-generated time stamp of when the frame was captured.

For live video, the time stamp is generated using a 100 KHz clock, and can be used for A/V synchronization.

For live audio, the time stamp is generated using an 8 KHz clock, and can be used for A/V synchronization.

For decoded audio and video, the time stamp generated during encoding (using the 100 KHz) clock is passed to the decoded audio and video frames.

In addition to the header, there are `y_buff`, `u_buff`, and `v_buff` that contain the data for the three part of a YUV frame.

*y_data* - Pointer to a buffer containing the Y data of a raw frame.

*u_data* - Pointer to a buffer containing the U data of a raw frame.

*v_data* - Pointer to a buffer containing the V data of a raw frame.

**NOTE:** When you are ready to release the frame, you must pass this structure to `sdvr_release_yuv_buffer()`.

**NOTE:** Raw audio is stored in `sdvr_av_buffer_t`, and does not need a separate data structure.

**_sdvr_osd_text_config_t**

```
typedef struct _sdvr_osd_text_config_t {
    char display_text[SDVR_MAX_OSD_TEXT + 1];
    sdvr_location_e text_location;
    sx_bool append_dts;
    sdvr_dts_style_e dts_style;
    sx_bool enable;
} sdvr_osd_text_config_t;
```

Data structure to store a single-byte string of OSD text support.

*display_text* - String to display as OSD text. This is a single byte NULL terminated string, and cannot be longer than `SDVR_MAX_OSD_TEXT` characters.

*text_location* - Location of the string on the window.

*append_dts* - True means date and time are appended to the text.

*dts_style* - The style in which DTS is shown (time format).

*enable* - A non-zero value indicates that the given OSD text is part of the video stream. A zero value disables OSD.

**NOTE:** This data structure, and all of its supporting functions, will be replaced with a set of more general functions starting with version 3.4.x. You should plan to stop using this data structure.

**_sdvr_osd_config_ex_t**

```
typedef struct _sdvr_osd_config_ex_t {
    sx_uint8 translucent;
    sdvr_location_e location_ctrl;
    sx_uint16 top_left_x;
    sx_uint16 top_left_y;
    sdvr_dts_style_e dts_format;
    sx_uint8 text_len;
    sx_uint16 text[SDVR_MAX_OSD_EX_TEXT];
} sdvr_osd_config_ex_t;
```

This data structure is used to configure each one of the OSD items associated with video frames of any camera or player. The number of OSD items is currently limited to two lines of 100 double-byte character strings per OSD item.

After an OSD item is configured, it can be shown or hidden at any time.

*translucent* - This field specifies the intensity of translucence when overlay OSD text is on the active video. 0 means least translucent, 255means most translucent.

*position_ctrl* - The position of OSD text. It can be any of the predefined locations in `sdvr_location_e` or a custom defined location.

*top_left_x*, *top_left_y* - The top left coordinates of the OSD text when the custom `postion_ctrl` is specified, otherwise these fields are ignored. The upper left corner of the video frame is the origin (0,0).

*dts_format* - The format of date and time to be appended optionally to the end of the OSD text.

*text_len* - The number of unsigned double-byte characters in the text field.

*text* - Up to 100 unsigned double-byte Unicode characters to be displayed.

**_sdvr_font_table_t**

```
typedef struct _sdvr_font_table_t {
    char *font_file;
    sx_uint8 font_table_id;
    sx_uint8 font_table_format;
    sx_uint32 start_font_code;
    sx_uint32 end_font_code;
    sx_uint8 color_y;
    sx_uint8 color_u;
    sx_uint8 color_v;
} sdvr_font_table_t;
```

This data structure is used to specify a new OSD font table. You can either use all the characters or a subset of characters within the font file. Additionally, you can choose a color to be used for all the characters. The same OSD font is used for all DVR boards connected at the time `sdvr_osd_set_font_table`() is called.

*font_file* - Full path to a the font file. Currently this must be a `.bdf` file.

*font_table_id* - The font table ID. User-defined font table IDs are 8–5; IDs 0–7 are reserved for system fonts. This field is ignored in this release.

*font_table_format* - The format of font file. Currently, the only supported font format is `SDVR_FT_FORMAT_BDF`. This field is ignored in this release.

*start_font_code* - The first font character to use within the font file. Use 0 for the lowest character code.

*end_font_code* - The last font character to use within the font file. Use 65535 for the highest character code.

*color_y* - Y component color of the character in YUV space. Use 255 for white.

*color_u* - U component color of the character in YUV space. Use 128 for white.

*color_v* - V component color of the character in YUV space. Use 128 for white.

**_sdvr_smo_grid_t**

```
typedef struct _sdvr_smo_grid_t {
    sx_uint16 top_left_mb_x;
    sx_uint16 top_left_mb_y;
```

```
    sdvr_video_res_decimation_e res_decimation;
    sx_uint8 dwell_time;
    sx_bool enable;
} sdvr_smo_grid_t;
```

This data structure is used to specify the spot monitor output grid pattern. The SMO display is divided into different grids specified by a top left location and a resolution decimation of the original video camera assigned to that grid.

Each encode or decode video channel can display its raw image at a particular pixel position on the SMO display.

Each grid on the SMO screen consists of one or more encode or decode channel outputs with a specific resolution decimation.

After a grid is defined, you can temporarily enable or disable its output.

*top_left_mb_x*, *top_left_mb_y* - These two numbers specify the top-left macro block coordinates of the display position and must be even number values. The coordinate of the top-left corner of SMO is (0,0).

*res_decimation* - The resolution to use to display the channels at this grid position. The image is resized based on this number.

*dwell_time* - If you put multiple channels in the same grid, then you must specify the length of time each channel is displayed before switching to the next channel. The firmware cycles through all the channels periodically, and switches them every N seconds, where N is the dwell time.

*enable* - To disable and not display this grid on the SMO, set the value of this field to zero. Otherwise, the channel assigned to this grid is displayed on the SMO.

**sdvr_video_alarm_callback**

```
typedef void (*sdvr_video_alarm_callback)
                              (sdvr_chan_handle_t handle,
                               sdvr_video_alarm_e alarm_type,
                               sx_uint32 data);
```

Typedef for video alarm callback function. This function has to be written by the DVR Application writer and be registered as a callback. Through this callback mechanism, the SDK will alert the DVR Application when video alarms happen above the specified threshold for the specific alarm.

The video alarm callback function is called whenever the encoder detects motion, blinding, nighttime light conditions, or video loss or detection. The callback function takes as its arguments the channel handle, the type of alarm, and alarm data. The meaning of alarm data varies depending on the type of alarm (i.e., for motion alarm, the alarm data is the actual amount of motion over the given threshold). These arguments are set by the SDK so that the callback function can determine which board and which video channel the alarm is coming from, and the type of alarm.

**sdvr_sensor_callback**

```
typedef void (*sdvr_sensor_callback)
            (sx_uint32 board_index, sx_uint32 sensor_map);
```

Typedef for sensor callback function. This function has to be written by the DVR Application writer and be registered as a callback. Through this callback mechanism, the SDK alerts the DVR Application when sensors trigger.

The sensor callback is called whenever one or more sensors on each board are triggered. The callback function takes as its arguments the board index and the sensor map. These arguments are set by the SDK so that the callback function can determine which board and which sensors have triggered. The sensors that have triggered are in `sensor_map`, with bit 0 corresponding to sensor 0, bit 1 to sensor 1, and so on.

**sdvr_av_frame_callback**

```
typedef void (*sdvr_av_frame_callback)
(sdvr_chan_handle_t handle, sdvr_frame_type_e frame_type,
 sx_bool primary_frame);
```

Typedef for AV frame callback function. This function has to be written by the DVR Application writer and be registered as a callback. Through this callback mechanism, the SDK alerts the DVR Application whenever encoded or raw AV frames are available.

The AV frame callback function is called whenever a new encoded AV frame is available, or when a raw video or audio frame is available. The callback function takes as its arguments the channel handle and the frame type to determine what the frame is and where it came from. The last argument is to distinguish between primary and secondary subchannels in a particular encoding channel. This parameter is only valid for encoded video frames and has no meaning for audio or raw video frames.

**sdvr_display_debug_callback**

```
typedef void (*sdvr_display_debug_callback)
                                              (char *message);
```

Typedef for displaying debug callback function. This function has to be written by the DVR Application writer and be registered as a callback. Through this callback mechanism, the SDK alerts the DVR Application whenever tracing error messages need to be displayed on the screen.

The callback function takes as its argument the string buffer to display.

**sdvr_signals_callback**

```
typedef void (*sdvr_signals_callback)
                          (sx_uint32 board_index,
                           sdvr_signal_info_t *signal_info);
```

Typedef for firmware asynchronous send message callback function. This function has to be written by the DVR Application writer and be registered as a callback. Through this callback mechanism, the SDK will signal the DVR Application when firmware needs to send messages to the DVR Application with-

out the application initiating a request. These messages are currently only error conditions generated on the firmware side, but in the future it could include other types of messages.

The callback function takes as its argument the signal type and a pointer to the signal information data structure.

## 1.9 Defines

`#define true`
`true` is a non-zero value

`#define false`
`false` is a zero value

**A/V State**

```
#define SDVR_AV_STATE_VIDEO_LOST 0x01
#define SDVR_AV_STATE_AUDIO_LOST 0x02
```

The current audio/video signal state on the channel in the `sdvr_av_buffer_t` structure.

`SDVR_AV_STATE_VIDEO_LOST` - No video signal is being detected on the channel.

`SDVR_AV_STATE_AUDIO_LOST` - No audio signal is being detected on the channel.

`#define SDVR_BOARD_SERIAL_LENGTH 16`
Length of the serial number string.

**Debug Flags**

```
#define DEBUG_FLAG_DEBUGGING_ON 0x1
#define DEBUG_FLAG_ALL 0xFFFFFFFE
#define DEBUG_FLAG_WRITE_TO_FILE 0x2
#define DEBUG_FLAG_OUTPUT_TO_SCREEN 0x4
#define DEBUG_FLAG_ENCODER 0x8
#define DEBUG_FLAG_DECODER 0x10
#define DEBUG_FLAG_VIDEO_ALARM 0x20
#define DEBUG_FLAG_SENSORS_RELAYS 0x40
#define DEBUG_FLAG_AUDIO_OPERATIONS 0x80
#define DEBUG_FLAG_DISPLAY_OPERATIONS 0x100
#define DEBUG_FLAG_BOARD 0x200
#define DEBUG_FLAG_GENERAL_SDK 0x400
#define DEBUG_FLAG_SMO 0x800
#define DEBUG_FLAG_OSD 0x1000
#define DEBUG_FLAG_CHANNEL 0x2000
#define DEBUG_FLAG_VIDEO_FRAME 0x4000
#define DEBUG_FLAG_FW_WRITE_TO_FILE 0x8000
#define DEBUG_FLAG_RECORD_TO_FILE 0x00010000
```

These debug flags are available to help you turn on debugging in the SDK.

DEBUG_FLAG_DEBUGGING_ON - Turn on debugging.

DEBUG_FLAG_ALL - Turn on all debugging flags.

DEBUG_FLAG_WRITE_TO_FILE - Turn on writing debug information to file.

DEBUG_FLAG_OUTPUT_TO_SCREEN - Write debug information to TTY.

DEBUG_FLAG_ENCODER - Turn on debugging for encoder.

DEBUG_FLAG_DECODER - Turn on debugging for decoder.

DEBUG_FLAG_VIDEO_ALARM - Turn on debugging for video alarm.

DEBUG_FLAG_SENSORS_RELAYS - Turn on debugging for sensors and relays.

DEBUG_FLAG_AUDIO_OPERATIONS - Turn on debugging for audio operations.

DEBUG_FLAG_DISPLAY_OPERATIONS - Turn on debugging for video operations.

DEBUG_FLAG_BOARD - Turn on debugging for board configuration.

DEBUG_FLAG_GENERAL_SDK - Turn on debugging for SDK configuration.

DEBUG_FLAG_SMO - Turn on debugging for SMO configuration.

DEBUG_FLAG_OSD - Turn on debugging for OSD configuration.

DEBUG_FLAG_CHANNEL - Turn on debugging for channel configuration.

DEBUG_FLAG_VIDEO_FRAME - Turn on debugging of video frames.

DEBUG_FLAG_FW_WRITE_TO_FILE - Turn on firmware only messages.

DEBUG_FLAG_RECORD_TO_FILE - Turn on debugging of recording the audio and video frame to a file.

### Maximum Regions

```
#define SDVR_MAX_MD_REGIONS 4
#define SDVR_MAX_BD_REGIONS 4
#define SDVR_MAX_PR_REGIONS 4
```

These defines specify the different maximum regions.

SDVR_MAX_MD_REGIONS - Maximum number of motion detection regions.

SDVR_MAX_BD_REGIONS - Maximum number of blind detection regions.

SDVR_MAX_PR_REGIONS - Maximum number of privacy regions.

### Maximum Number of OSD Characters

```
#define SDVR_MAX_OSD_TEXT 10
#define SDVR_MAX_OSD_EX_TEXT 100
```

This defines the maximum number of OSD text characters that can be displayed as an OSD text string.

SDVR_MAX_OSD_TEXT - The maximum size of OSD text for single byte OSD APIs is 10 characters.

`SDVR_MAX_OSD_EX_TEXT` - The maximum length of OSD text in OSD APIs supporting double byte characters is 100 unsigned short. Available only in firmware version 3.2.0.0 or later.

**Maximum OSD Items**

```
#define SDVR_MAX_OSD 2
```

This defines the maximum number of OSD items that can be configured per each channel. Available only in firmware version 3.2.0.0 or later.

**Invalid Channel Handle**

```
#define INVALID_CHAN_HANDLE (sx_int32)0xFFFFFFFF
```

This defines the invalid channel handle.

**Pre-loaded Font Tables**

```
#define SDVR_FT_FONT_ENGLISH 1
```

Stretch DVR pre-loaded font tables.

`SDVR_FT_FONT_ENGLISH` - Fonts for English character sets.

**Font Table Format**

```
#define SDVR_FT_FORMAT_BDF 1
```

`SDVR_FT_FORMAT_BDF` - The Glyph Bitmap Distribution Format (BDF) by Adobe is a file format for storing bitmap fonts. BDF is the most commonly used font file within the lLinux operating system. Currently, this is the only supported format.

# Chapter 2 API Syntax Definitions

## 2.1 System Set Up, SDK, and Board Initialization API

### 2.1.1 sdvr_get_error_text

```
char * sdvr_get_error_text (sdvr_err_e error_no);
```

This function returns the text string mnemonic for the given error number. If the given error number is not valid, a string with the number is returned.

Parameters

| Name | Description |
|---|---|
| *error_no* | The error number for the text string |

Returns

A pointer to a null terminated string with the text of the error.

**NOTE:** This string should not be **freed**.

## 2.1.2 sdvr_sdk_init

```
sdvr_err_e sdvr_sdk_init ();
```

This function initializes the SDK and the driver, allocates system resources required by them, and discovers all SDVR cards in the system.

**NOTE:** the cards are not initialized by this function.

This is the first call that you must make in your application before you can use any of the other API functions with the exception of get version or set callback calls. Also, this function should be called only once per session.

**Parameters** None

**Returns** `SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.1.3 sdvr_sdk_close

```
sdvr_err_e sdvr_sdk_close ();
```

This function closes the SDK and the driver, and frees up the system resources used by them. You must call this function prior to exiting your DVR Application for a clean shutdown of your system. No other API function calls, except `sdvr_sdk_init()`, are allowed after this function is called.

**Parameters** None

**Returns** `SDVR_ERR_NONE` - On success, Otherwise, see the error code list.

## 2.1.4 sdvr_get_board_count

```
sx_uint32 sdvr_get_board_count ();
```

This function returns the number of SDVR boards in the system. Only boards that are functioning properly are discovered and reported by this function.

**Parameters** None

**Returns** The number of SDVR boards discovered.

## 2.1.5 sdvr_get_board_attributes

```
sdvr_err_e sdvr_get_board_attributes (sx_uint32 board_index,
                                      sdvr_board_attrib_t * board_attrib);
```

This function returns the board attributes of each board. Use this function to discover which PCI slot the board is connected to and its type, as well as all the video standards supported by the firmware.

You must call this function before calling `sdvr_board_connect()` to select one of the video standards supported by the firmware.

**Parameters**

| Name | Description |
|---|---|
| *board_index* | The number of the board whose attributes you want. This is a zero-based number; that is, the first board number is index zero, the second board number is index one, and so on. |
| *board_attrib* | A pointer to a variable that will hold the attributes when this function returns. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.1.6 sdvr_get_pci_attrib

```
sdvr_err_e sdvr_get_pci_attrib (sx_uint32 board_index,
                                sdvr_pci_attrib_t * pci_attrib);
```

This function returns the board PCI attributes of each board. Use this function to discover which PCI slot the board is connected to, its type, vendor and device ID, and serial number.

Parameters

| Name | Description |
|---|---|
| *board_index* | The board number for which to get PCI information. This is a zero-based number. It means the first board number is index zero, second board number is index one, and so on. |
| *pci_attrib* | A pointer to a variable that will hold the PCI information when this function returns successfully. |

**Returns** `SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

**Remarks** This function can be called before the call to `sdvr_board_connect`() or before loading a firmware into the DVR board.

## 2.1.7 sdvr_board_reset

```
sdvr_err_e sdvr_board_reset (sx_uint32 board_index);
```

This function resets the firmware on the given board. We recommend that you not connect to any board prior calling this function. After calling `sdvr_board_reset`(), you must reload the firmware if it is not already burned onto the board. Additionally, your system also needs to be reconfigured after board reset.

The usage of this function is heavily discouraged because it can put the SDK in an indeterminate stage. In general, if the firmware is not already burned onto the DVR board, this function should not be called.

**Parameters**

| Name | Description |
|---|---|
| *board_index* | The number of the board that you want to reset. This is a zero-based number; that is, the first board number is index zero, the second board number is index one, and so on. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.1.8 sdvr_board_connect

```
sdvr_err_e sdvr_board_connect (sx_uint32 board_index, sdvr_video_std_e video_std,
sc_bool is_h26r_SCE);
```

This function connects to a board and sets up communication channels and other system resources required to handle the board.

This function should only be called once per board.

**NOTE:** To change and set a new video standard, the DVR firmware is reset every time the DVR board is connected.

**Parameters**

| Name | Description |
|---|---|
| *board_index* | The number of the board to which you want to connect. This is a zero-based number; that is, the first board number is index zero, the second board number is index one, and so on. |
| *video_std* | The video standard and maximum system-wide resolution of all the cameras connected to the board. |
| *is_h264_SCE* | This field specifies whether the SDK or the H.264 encoder should perform start code emulation (SCE) for H.264 video frames. For performance reasons, it is strongly recommended that you always set this field to 1 so that the SDK performs SCE, except for the embedded DVR Applications for which it must be set to 0, which means the encoder performs this task.<br>This field is ignored if connected to any firmware version prior to 3.2.0.19. |

**Returns** `SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

**Remarks** All the cameras connected to the board must be of the same video standard (NTSC or PAL). In the event that you have connected mixed video standard cameras, the video frames for any camera that is different from the specified video standard will be distorted.

## 2.1.9 sdvr_board_disconnect

```
sdvr_err_e sdvr_board_disconnect (sx_uint32 board_index);
```

This function disconnects from a board, and releases all the board specific resources allocated in the SDK and driver. After this call, all attempts to communicate with the board fail.

To shut down your system cleanly, you must call this function for every SDVR board to which you were connected before you exit your application.

**Parameters**

| Name | Description |
|---|---|
| *board_index* | The number of the board from which you want to disconnect. This is a zero-based number; that is, the first board number is index zero, the second board number is index one, and so on. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list. If disconnect fails, you must restart your application before you can connect to the same board again.

**Remarks**

You must call `sdvr_upgrade_firmware` every time prior to connecting the the DVR board if no firmware is loaded into the board's non-volatile memory.

## 2.1.10 sdvr_upgrade_firmware

```
sdvr_err_e sdvr_upgrade_firmware (sx_uint32 board_index,
                                  char * firmware_file_name);
```

This function is used to load a firmware onto the board. This function loads the contents of the given file (which is required to be in a `.rom` format) into the board memory, and directs the board to burn it into volatile memory. The board then automatically *reboots*, i.e., starts up with the new firmware, without requiring a PC reboot.

Before you load the firmware, you must disconnect from the board using `sdvr_board_disconnect`(). You must shut down all board functionality before attempting to load new firmware. Also, after you upgrade firmware, you must reconnect to the board using `sdvr_board_connect`().

**NOTE:** You must load the firmware every time prior to connecting to the DVR board if no firmware is loaded into the board's none-volatile memory.

**Parameters**

| Name | Description |
|---|---|
| *board_index* | The number of the board that you want to upgrade. This is a zero-based number; that is, the first board number is index zero, the second board number is index one, and so on. |
| *firmware_file_name* | A string containing the name of the new firmware file in `.rom` format. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.1.11 sdvr_set_sdk_params

```
sdvr_err_e sdvr_set_sdk_params (sdvr_sdk_params_t *sdk_params);
```

This function sets the SDK parameters that control resource allocation in the SDK. It must be called before you connect to any boards to assign buffer resources. Otherwise, the default buffer resource values will be used by the SDK. Any subsequent calls to this function after being connected will only result in changing the debug information or the PCI response timeout value.

We highly recommend that you not change the default buffer size or numbers unless it is needed for performance reasons.

To change only some SDK parameters and keep the default setting for the rest, you should call `sdvr_get_sdk_params`() to get the existing parameter settings, change the desired parameters, and then call `sdvr_set_sdk_params`().

There must be at least one buffer specified for each of the system buffers.

The SDK parameters contain the following fields, Guidelines on how to set their values are:

*enc_buf_num* — The number of buffers to be allocated for each encoder channel. Each encoder channel will have the same number of buffers allocated. It is important that you allocate enough buffers for encoded frames to be held between the times you can process them. Maximum number of encoder buffers is 25. Default value is 20.

*raw_buf_num* — The number of buffers to be allocated for each channel that will be sending raw video to the DVR Application. Each channel sending raw video will have the same number of buffers. Typically, you will need to display raw frames 30 times per second. Therefore, you only need between 2 to 4 buffers to hold raw video per channel. Maximum number of raw buffers is 5. Default value is 2.

*dec_buf_num* — The number of buffers to be allocated for each decoder channel. Each decoder channel will have the same number of buffers allocated.
Each decoder channels is limited to a maximum of 5 buffers. Default value is 5.

*dec_buf_size* — The size of each buffer used to hold encoded frames on the way to the decoder hardware. This is the size of the largest encoded frame on any channel, as the same buffer size is used across all channels. Default value is 414720 which is PAL D1 720x576.

| | |
|---|---|
| *timeout* | For a variety of reasons, it is possible for the board to hang. By setting the timeout value, you specify when the SDK will give up on a response from the board and inform the DVR Application that the board is hung. The value of this parameter is in seconds. A value of 0 indicates that there is no timeout and the SDK will wait indefinitely for the board to respond. Do not set the value of timeout too low, or during times of heavy traffic, you might get a false warning that the board has hung. Setting the value to 0 (no timeout), however, may cause the DVR Application to hang if the firmware on the board dies. Default value is 5. |
| *debug_flag* | This is a bit field of flags that can be set to enable various levels of debugging as defined by the debug flags. See the defines for `DEBUG_FLAG_xxx` for definition of each field. You can turn on tracing of different part of SDK by enabling the subsystem tracing bit. The tracing messages can be redirected to a log file or screen. |
| *debug_file_name* | The name of log file to write the tracing information according to the debug_flag. The log file will not be update if no bit in the debug_flag field is set. |

**Parameters**

| Name | Description |
|---|---|
| *sdk_params* | A pointer to a structure containing the SDK information that you want to set. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.1.12 sdvr_get_sdk_params

```
sdvr_err_e sdvr_get_sdk_params (sdvr_sdk_params_t *sdk_params);
```

This function gets the SDK parameters that control resource allocation in the SDK.

**Parameters**

| Name | Description |
|---|---|
| *sdk_params* | A pointer to a structure containing the SDK information that will be filled when this function returns. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

**Remarks**

We recommend that you call this function prior to the call to `sdvr_set_sdk_params`() to preserve the default values of any SDK parameters that do not need to be changed.

## 2.1.13 sdvr_get_sdk_version

```
void sdvr_get_sdk_version (sx_uint8 * major, sx_uint8 * minor, sx_uint8 * revision,
                           sx_uint8 * build);
```

This function returns the SDK version.

Stretch follows the convention of using four numbers for version control. A change in the major number indicates major changes to functionality, a change in the minor number indicates minor changes to functionality, and a change in the revision number indicates significant bug fixes that were introduced in the minor change functionality. A change to the build number indicates only bug fixes that do not change functionality.

This function can be called before or after `sdvr_sdk_init`();

**Parameters**

| Name | Description |
|---|---|
| *major* | Pointer to a variable that will hold the major version of the SDK when this function returns. |
| *minor* | Pointer to a variable that will hold the minor version of the SDK when this function returns. |
| *revision* | Pointer to a variable that will hold the revision version of the SDK when this function returns. |
| *build* | Pointer to a variable that will hold the build or bug fix version of the SDK when this function returns. |

**Returns** Nothing

## 2.1.14 sdvr_get_driver_version

```
void sdvr_get_driver_version (sx_uint8 * major, sx_uint8 * minor,
                              sx_uint8 * revision, sx_uint8 * build);
```

This function returns the driver version.

Stretch follows the convention of using four numbers for version control. A change in the major number indicates major changes to functionality, a change in the minor number indicates minor changes to functionality, and a change in the revision number indicates significant bug fixes that were introduced in the minor change functionality. A change to the build number indicates only bug fixes that do not change functionality.

**Parameters**

| Name | Description |
|---|---|
| *major* | Pointer to a variable that will hold the major version of the driver when this function returns. |
| *minor* | Pointer to a variable that will hold the minor version of the driver when this function returns. |
| *revision* | Pointer to a variable that will hold the revision version of the driver when this function returns. |
| *build* | Pointer to a variable that will hold the build or bug fix version of the driver when this function returns. |

**Returns** Nothing

## 2.1.15 sdvr_get_firmware_version

```
sdvr_err_e sdvr_get_firmware_version (sx_uint32 board_index, sx_uint8 * major,
                                      sx_uint8 * minor, sx_uint8 * build);
```

This function returns the firmware version.

Stretch follows the convention of using four numbers for version control. A change in the major number indicates major changes to functionality, a change in the minor number indicates minor changes to functionality, and a change in the revision number indicates significant bug fixes that were introduced in the minor change functionality. A change to the build number indicates only bug fixes that do not change functionality.

**NOTE:** This function will be removed in version 3.4.0.0.

**Parameters**

| Name | Description |
|---|---|
| *board_index* | The number of the board whose firmware version we want. This is a zero-based number; that is, the first board number is index zero, the second board number is index one, and so on. |
| *major* | Pointer to a variable that will hold the major version of the firmware when this function returns. |
| *minor* | Pointer to a variable that will hold the minor version of the firmware when this function returns. |
| *build* | Pointer to a variable that will hold the build or version of the firmware when this function returns. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

**Remarks**

Call `sdvr_get_firmware_version_ex()` to get more detailed version information including the revision number.

## 2.1.16 sdvr_get_firmware_version_ex

```
sdvr_err_e sdvr_get_firmware_version_ex (sx_uint32 board_index,
                                         sdvr_firmware_ver_t * version_info);
```

This function returns the firmware and boot loader version.

Stretch follows the convention of using four numbers for version control. A change in the major number indicates major changes to functionality, a change in the minor number indicates minor changes to functionality, and a change in the revision number indicates significant bug fixes that were introduced in the minor change functionality. A change to the build number indicates only bug fixes that do not change functionality.

This function provides more detailed information over `sdvr_get_firmware_version`(). You can access the build date of the firmware in addition to the firmware version and version of the boot loader.

This function must be called after calling `sdvr_upgrade_firmware`() to get the boot loader information and after `sdvr_board_connect`() to get both the firmware and boot loader versions.

This function can still be used to get the BSP and boot loader versions even if the call to `sdvr_upgrade_firmware`() returns an error. In such cases, no other version information is available.

**Parameters**

| Name | Description |
|---|---|
| *board_index* | The number of the board whose firmware version you want. This is a zero-based number; that is, the first board number is index zero, the second board number is index one, and so on. |
| *version_info* | A pointer to a structure that holds the firmware and boot loader version numbers. |

**Returns**

`SDVR_ERR_NONE` - Both the firmware and boot loader version information are returned.

`SDVR_ERR_BOARD_NOT_CONNECTED` - The firmware version is not returned because the given board number is not connected, but the boot loader information might have been returned if it is non-zero.

Otherwise, see the error code list.

## 2.1.17 sdvr_set_sensor_callback

```
sdvr_sensor_callback sdvr_set_sensor_callback
                                    (sdvr_sensor_callback sensor_callback);
```

This function is used to register the sensor callback function. There can be only one function registered for this callback.

For the DVR Application to be notified of sensor events, it has to register a callback. The callback function has as its arguments the board index and the sensor map that contains a bit pattern describing the sensors that have triggered. This information can be used in the callback function to determine which sensors have triggered and take the appropriate action. Bit 0 in the sensor map corresponds to sensor 0, bit 1 to sensor 1, and so on.

**Parameters**

| Name | Description |
|---|---|
| *sensor_callback* | A pointer to the callback function. |

**Returns**

Returns a pointer to the previous function that was registered for this callback. If no function was registered previously, the return value is a NULL function pointer. This can be used to temporarily override a callback function with another and then to restore the original callback.

## 2.1.18 sdvr_set_video_alarm_callback

```
sdvr_video_alarm_callback sdvr_set_video_alarm_callback
                                (sdvr_video_alarm_callback video_alarm_callback);
```

This function is used to register the video alarm callback. There can be only one function registered for this callback.

For the DVR Application to be notified of video alarm events, it has to register a callback. The callback function has as its arguments the video channel handle, the alarm type, and the value associated with the alarm. This information can be used in the callback function to determine the type of event that occurred and, to take the appropriate action. There is one callback per video event, even if the events happen simultaneously.

**Parameters**

| Name | Description |
|---|---|
| *alarm_callback* | A pointer to the callback function. |

**Returns**

Returns a pointer to the previous function that was registered for this callback. If no function was registered previously, the return value is a NULL function pointer. This can be used to temporarily override a callback function with another and then to restore the original callback.

**Remarks**

See the `sdvr_video_alram_callback` definition for more information

## 2.1.19 sdvr_set_av_frame_callback

```
sdvr_av_frame_callback sdvr_set_av_frame_callback
                                    (sdvr_av_frame_callback av_frame_callback);
```

This function is used to register the AV frame callback function. There can be only one function registered for this callback. The callback is called every time encoded AV, raw video, and raw audio frames are received from the SDVR boards. The function has as its arguments the board index, the channel number, the frame type, and whether the frame is from a primary channel. This information can be used in the callback function to perform the appropriate action; encoded frames are saved to disk, raw video frames are displayed, and raw audio frames are played.

The callback function is used for asynchronous notification, somewhat like a hardware interrupt. We recommend that you note the information provided in the callback, but not process the A/V frame itself in the callback function. That should be done in the appropriate routine or thread dedicated to processing encoded and raw frames.

**NOTE:** It is not necessary to register this callback. A polling mechanism can be used with `sdvr_get_av_buffer()` to get the buffers.

**Parameters**

| Name | Description |
|---|---|
| *av_frame_callback* | A pointer to the callback function. |

**Returns**

Returns a pointer to the previous function that was registered for this callback. If no function was registered previously, the return value is a NULL function pointer. This can be used to temporarily override a callback function with another and then to restore the original callback.

**Example**

```
void av_frame_callback(sdvr_chan_handle_t handle, sdvr_frame_type_e frame_type,
                       sx_bool is_primary)
{
    sdvr_av_buffer_t  *av_buffer;
    sdvr_yuv_buffer_t *yuv_buf;
    switch (frame_type)
    {
    case SDVR_FRAME_AUDIO_ENCODED:
        if (sdvr_get_av_buffer(handle, SDVR_FRAME_AUDIO_ENCODED, &av_buffer) ==
            SDVR_ERR_NONE)
        {
            <add the buffer to an audio queue to be processed from a different
             thread>
            <The buffer should be released from that thread when you processed
             the audio frame.>
        }
        break;
    case SDVR_FRAME_RAW_VIDEO:
        if (sdvr_get_yuv_buffer(handle, &yuv_buf) == SDVR_ERR_NONE
```

```
        {
             <add the buffer to a YUV queue to be processed from
              a different thread>
            <The YUV buffer should be released from that thread when you
              processed the frame.>
        }
        break;
     case SDVR_FRAME_H264_IDR:
     case SDVR_FRAME_H264_I:
     case SDVR_FRAME_H264_P:
     case SDVR_FRAME_H264_B:
     case SDVR_FRAME_H264_SPS:
     case SDVR_FRAME_H264_PPS:
     case SDVR_FRAME_JPEG:
     case SDVR_FRAME_MPEG4_I:
     case SDVR_FRAME_MPEG4_P:
     case SDVR_FRAME_MPEG4_VOL:
        if (sdvr_get_av_buffer(handle,
          is_primary ? SDVR_FRAME_VIDEO_ENCODED_PRIMARY :
          SDVR_FRAME_VIDEO_ENCODED_SECONDARY, &av_buffer) == SDVR_ERR_NONE)
        {
             <add the buffer to a video encoded queue to be processed
              from a different thread.>
            <The buffer should be released from that thread when you processed the
             video frame.>
        }
        break;
    }
    NOTE: Any buffer that is not retrieved(i.e. no call  to the
          "sdvr_get_xxx_buffer") will be freed by the SDK once the frame queue is
          full.
}
```

## 2.1.20 sdvr_set_display_debug

```
sdvr_display_debug_callback sdvr_set_display_debug
                            (sdvr_display_debug_callback display_debug_callback);
```

This function is used to register the callback function that displays the debug message.

Through this callback mechanism, the SDK alerts the DVR Application whenever tracing error messages need to be displayed on the screen.

The callback function takes as its argument the string buffer to display.

**NOTE:** If this callback is not registered and you enable the display-to-screen flag, the message is written to `stdout`.

**Parameters**

| Name | Description |
|---|---|
| *display_debug_callback* | A pointer to the callback function. |

**Returns**

Returns a pointer to the previous function that was registered for this callback. If no function was previously registered, the return value is a NULL function pointer. This can be used to temporarily override a callback function with another and then to restore the original callback.

## 2.1.21 sdvr_set_signals_callback

```
sdvr_signals_callback sdvr_set_signals_callback
                                    (sdvr_signals_callback signals_callback);
```

This function is used to register the signal callback function. There can only be one function registered for this callback.

For the DVR Application to be notified of asynchronous events such as error conditions in the DVR firmware, it has to register a callback. The callback function has as its arguments the board index and a pointer to data structure associated with the signal event. This information can be used in the callback function to determine which device caused the signal to be triggered and whether the DVR Application should be closed or continue execution.

**Parameters**

| Name | Description |
|---|---|
| *signals_callback* | A pointer to the callback function. |

**Returns**

Returns a pointer to the previous function that was registered for this callback. If no function was registered previously, the return value is a NULL function pointer. This can be used to temporarily override a callback function with another and then to restore the original callback.

## 2.1.22 sdvr_get_board_config

```
sdvr_err_e sdvr_get_board_config (sx_uint32 board_index,
                                  sdvr_board_config_t * board_config);
```

This function returns the board configuration parameters, i.e., the number of cameras, number of sensors and relays, and so on.

There is no corresponding function to set these parameters because the configuration information returned by this function cannot be changed. The configuration information essentially is a description of the capabilities of the board.

**Parameters**

| Name | Description |
|---|---|
| *board_index* | The index of the board for which you want this information. This is a zero-based number; that is, the first board number is index zero, the second board number is index one, and so on. |
| *board_config* | A pointer to a configuration data structure that is filled appropriately when this function returns. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

**Remarks**

It is not necessary to connect to the board to call this function.

## 2.1.23 sdvr_get_supported_vstd

```
sdvr_err_e sdvr_get_supported_vstd (sx_uint32 board_index, sx_uint32 * video_stds);
```

This function gets a list of all video standards supported by the firmware for a given board.

**Parameters**

| Name | Description |
|---|---|
| *board_index* | The index of the board. This is a zero-based number. It means the first board number is index zero, the second board number is index one, and so on. |
| *video_stds* | A pointer to a variable that contains all the supported video standards. It is a bit `OR` of `sdvr_video_std_e`. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise see the error code list.

## 2.1.24 sdvr_get_video_standard

```
sdvr_err_e sdvr_get_video_standard (sx_uint32 board_index,
                                    sdvr_video_std_e * video_std_type);
```

This function returns the current video standard for a particular board.

Parameters

| Name | Description |
|---|---|
| *board_index* | The index of the board. This is a zero-based number; that is, the first board number is index zero, the second board number is index one, and so on. |
| *video_std_type* | A pointer to a variable that will contain the video standard when this function returns. |

**Returns** `SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

**Remarks** All boards in a system must have the same video standard.

## 2.1.25 sdvr_set_watchdog_state

```
sdvr_err_e sdvr_set_watchdog_state (sx_uint32 board_index, sx_uint32 seconds);
```

This function enables or disables the watchdog timer. Each board is equipped with a watchdog timer that expires 10 seconds after it is enabled. When the watchdog timer expires, the board is reset and a reset signal is issued from the board. If this reset signal is tied to the reset signal on the PC's motherboard, the entire PC is reset and rebooted.

> **NOTE:** When the watchdog timer expires, data on the way to the disk may-be lost and the system may be left in an inconsistent state. Therefore, use this function with caution.

If you are going to use this function to guard against the system hanging indefinitely, then set the watchdog timer on all boards in your system.

**Parameters**

| Name | Description |
|---|---|
| *board_index* | Index of the board on which to set the watchdog timer. This is a zero-based number; that is, the first board number is index zero, the second board number is index one, and so on. |
| *enable* | Enable or disable the watchdog timer. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

**Remarks**

The watchdog timer is set at 10 seconds, therefore we recommend that you call `sdvr_set_watchdog_state()` every 5 seconds to prevent the timer from expiring.

## 2.1.26 sdvr_get_watchdog_state

```
sdvr_err_e sdvr_get_watchdog_state (sx_uint32 board_index,
                                    sx_uint32 * num_seconds);
```

This function returns the current enable state of the watchdog timer.

Parameters

| Name | Description |
|---|---|
| *board_index* | The index of the board from which to get watchdog timer value. This is a zero-based number; that is, the first board number is index zero, the second board number is index one, and so on. |
| *enable* | A pointer to a variable that will contain the enable state of the watchdog timer on a successful return by this function. |

**Returns** `SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.1.27 sdvr_set_date_time

```
sdvr_err_e sdvr_set_date_time (sx_uint32 board_index, time_t time);
```

This function sets the date and time in the firmware of a particular SDVR board. The time is specified as a 32-bit value, which is the number of seconds elapsed since midnight of January 1, 1970.

Parameters

| Name | Description |
|---|---|
| *board_index* | Index of the board on which to set the time. This is a zero-based number; that is, the first board number is index zero, the second board number is index one, and so on. |
| *time* | The time value to be set. |

**Returns** `SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.1.28 sdvr_get_date_time

```
sdvr_err_e sdvr_get_date_time (sx_uint32 board_index, time_t *time);
```

This function returns the date and time set in the firmware of a particular SDVR board. The time returned is a 32-bit value, which gives the number of seconds elapsed since midnight of January 1, 1970.

**Parameters**

| Name | Description |
|---|---|
| *board_index* | Index of the board for which the time is requested. This is a zero-based number; that is, the first board number is index zero, the second board number is index one, and so on. |
| *time* | A pointer to a `time_t` type that will hold the value of time when this function returns. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.1.29 sdvr_run_diagnostics

```
sdvr_err_e sdvr_run_diagnostics (sx_uint32 board_index, char * diag_file_name,
                                 sdvr_diag_code_e * diag_code);
```

This function is used to run diagnostics on the board. It accepts the name of the diagnostics firmware file and returns the diagnostics result in the given `diag_code` parameter. This function overwrites the existing DVR firmware, so you must reload the DVR firmware for the DVR to function properly.

Running diagnostics can take some time.

**Parameters**

| Name | Description |
|---|---|
| *board_index* | Index of the board for which to run the diagnostics. This is a zero-based number; that is, the first board number is index zero, the second board number is index one, and so on. |
| *diag_file_name* | A string containing the name of the file containing the diagnostics firmware. |
| *diag_code* | A pointer to an array of four 32-bit variables that hold the diagnostics results for each PE. Up on return, `diag_code[0]` holds the result for PE0, `diag_code[1]` has the result for PE1, and so on. Refer to `sdvr_diag_code_e` for the possible values. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.1.30 sdvr_get_board_index

```
sx_uint8 sdvr_get_board_index (sdvr_chan_handle_t handle);
```

This function returns the board number, given a channel handle.

**Parameters**

| Name | Description |
|---|---|
| *handle* | A channel handle. |

**Returns**

Zero-based board index corresponding to this handle. A return value of `0xFF` indicates that the channel handle is invalid.

## 2.1.31 sdvr_get_chan_num

```
sx_uint8 sdvr_get_chan_num (sdvr_chan_handle_t handle);
```

This function returns the channel number, given a channel handle.

**Parameters**

| Name | Description |
|---|---|
| *handle* | A channel handle. |

**Returns**

Channel number corresponding to this handle. A return value of `0xFF` indicates that the channel handle is invalid.

## 2.1.32 sdvr_get_chan_type

```
sdvr_chan_type_e sdvr_get_chan_type (sdvr_chan_handle_t handle);
```

This function returns the channel type, given a handle.

**Parameters**

| Name | Description |
|---|---|
| *handle* | A channel handle. |

**Returns**

`SDVR_CHAN_TYPE_ENCODER` - For an encoding channel.

`SDVR_CHAN_TYPE_DECODER` - For a decoding channel.

`SDVR_CHAN_TYPE_NONE` - For a channel whose type is either not set, or the channel handle is invalid.

# 2.2 Channel Set Up API

## 2.2.1 sdvr_create_chan

```
sdvr_err_e sdvr_create_chan (sdvr_chan_def_t * chan_def,
                             sdvr_chan_handle_t * handle_ptr);
```

This function is used to create an encoding, decoding, or HMO-only channel.

Each board can have a certain number of cameras connected to it. Call this number M. The encoding channels on the board are then numbered from 0 to M-1. Any channel number in this range is an encoding channel and corresponds to the physical camera connection on the board. Additionally, you can have a certain number of decoders. Call this number N. The decoding channels on the board are numbered from 0 to N-1. These decoder channel number are virtual because there is no physical hardware attached to them unlike the encoder channels.

Each Stretch chip is capable of encoding or decoding four D1 video channels, but the sum of encoding and decoding channels may not exceed six. This means that channel numbers 0 to 3 are assigned to first Stretch chip, the next four channel numbers are assigned to the second Stretch chip and so on.

The number of decoding channels that can be set up depends on the processing power left after all encoding channels are assigned. When this function is called to set up a decoding channel, it may fail because there is no more processing power left for decoding. If the call is successful in setting up a decoding channel, another call to this function can be tried to set up an additional decoding channel. This procedure can be repeated until the calls fail, which indicates that there is no more processing power left.

Even though you can have 0 to M-1 encoding channels, you do not have to use all of them for encoding (perhaps because not that many cameras are connected to the board). This lets you have more decoding channels. Use this function to set the encoding channels to the ones that have cameras connected to them.

A video encoding channel can have up to two different video encoders, as well as an optional audio encoder. By specifying a secondary video encoder for a channel, you are taking away more computing power, which may not allow you to have four D1 video encoding or decoding channels.

Upon success, this call returns a handle to a channel that is used for all other calls that need a channel identifier.

Parameters

| Name | Description |
|---|---|
| *chan_def* | Data structure defining the new channel attributes. |
| *handle_ptr* | Holds the channel handle upon successful return. |

**Returns** `SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.2.2 sdvr_set_channel_default

```
sdvr_err_e sdvr_set_channel_default (sdvr_chan_handle_t handle);
```

This function is used to reset an encoding or decoding channel to its factory default settings. Make sure to close down the channel (stop encoding or decoding) before resetting the channel.

The default settings of an encoding or decoding channel can be found in the description of the `sdvr_video_enc_chan_params_t` data structure.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | An encoding channel handle. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.2.3 sdvr_destroy_chan

```
sdvr_err_e sdvr_destroy_chan (sdvr_chan_handle_t handle);
```

This function is used to destroy the given channel handle.

After you have set up some encoder or decoder channels, and you need to add more, you may get an error message that there is not enough CPU power to add the new channel. In this case, you may decide to temporarily destroy one or more channels and add the new channel type.

**NOTE:** You must stop all A/V streaming on this channel before destroying it.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | The channel handle to be destroyed. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.2.4 sdvr_set_chan_user_data

```
sdvr_err_e sdvr_set_chan_user_data (sdvr_chan_handle_t handle,
                                    sx_uint64 user_data);
```

In your DVR Application, you may need to access a certain data structure associated with an encoder or decoder channel.

This function allows you to associate a 64-bit variable with a specified channel handle. This data can be retrieved at any time by calling `sdvr_get_chan_user_data()`.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | An encoding or decoding video channel handle. |
| *user_data* | Specifies the data to be associated with the channel. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.2.5 sdvr_get_chan_user_data

```
sdvr_err_e sdvr_get_chan_user_data (sdvr_chan_handle_t handle,
                                    sx_uint64 * user_data);
```

This function lets you retrieve the application-defined data associated with the specified channel handle. This 64-bit variable was set by calling `sdvr_get_chan_user_data()`.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | An encoding or decoding video channel handle. |
| *user_data* | A pointer to hold the data associated with this channel. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.2.6 sdvr_set_video_encoder_channel_params

```
sdvr_err_e sdvr_set_video_encoder_channel_params (sdvr_chan_handle_t handle,
                                                  sdvr_sub_encoders_e sub_chan_enc,
                                                  sdvr_video_enc_chan_params_t * video_enc_params);
```

This function is used to set the video parameters for either the primary or secondary encoder of a video encoder channel. The encoder channel parameters are determined by the type of connected camera, the resolution, frame rate, and so on. All these parameters are defined in `sdvr_video_enc_chan_params_t`, and are set using this function. Any video encoder parameter except the video resolution can be changed regardless of the encoder enable status. This means that to change the video resolution of a channel, the encoder must be disabled before calling this function.

There are no corresponding functions for setting the parameters for a video decoding channel. This is because the video decoding channel takes its parameters from the encoded stream.

The encoder-specific parameters are ignored for any channel that has its primary encoder set to `SDVR_VIDEO_ENC_NONE`.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | An encoding channel handle. |
| *sub_chan* | Specifies whether to set the parameters for the primary or secondary encoder. |
| *video_enc_params* | The parameters for the video encoder channel. See `sdvr_video_enc_chan_params_t` for detailed information of each parameter. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

**Remarks**

Any video encoder parameters can be changed while the encoder is enabled except video resolution. If the encoder is enabled, all the encoder parameter changes take place at the end of the GOP, with the exception of frame rate, which takes place immediately.

## 2.2.7 sdvr_get_video_encoder_channel_params

```
sdvr_err_e sdvr_get_video_encoder_channel_params (sdvr_chan_handle_t handle,
                                                  sdvr_sub_encoders_e sub_chan_enc,
                                                  sdvr_video_enc_chan_params_t * video_enc_params);
```

This function is used to get the parameters of a video encoder channel.

The encoder-specific parameters for any channel that has its primary encoder set to SDVR_VIDEO_ENC_NONE have no meaning.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | An encoding channel handle. |
| *sub_chan_enc* | Specifies whether to set the parameters for the primary or secondary encoder. |
| *video_enc_params* | A pointer to a variable that will be filled with encoder channel parameters when this function returns. |

**Returns**

SDVR_ERR_NONE - On success. Otherwise, see the error code list.

## 2.2.8 sdvr_set_alarm_video_encoder_params

```
sdvr_err_e sdvr_set_alarm_video_encoder_params (sdvr_chan_handle_t handle,
                                                sdvr_sub_encoders_e sub_chan_enc,
                                                sdvr_alarm_video_enc_params_t * alarm_video_enc_params);
```

This function is used to set the parameters of a video encoder channel after any of the alarms are triggered. You can specify two sets of video encoder parameters. One to be used on streaming of video encoded frames before alarms are triggered. The other to be used after any of the alarms is triggered for a minimum duration.

**NOTE:** These parameters are used only for encoded channels and while they are enabled. Otherwise, the value of the encoder parameters for both pre- and post-alarm conditions is ignored.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | Handle of an encoding channel. |
| *sub_chan_enc* | Specifies whether to set the parameters for the primary or secondary encoder. |
| *alarm_video_enc_params* | The parameters for the video encoder channel after an alarm is triggered. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.2.9 sdvr_get_alarm_video_encoder_params

```
sdvr_err_e sdvr_get_alarm_video_encoder_params (sdvr_chan_handle_t handle,
                                                sdvr_sub_encoders_e sub_chan_enc,
                                                sdvr_alarm_video_enc_params_t * alarm_video_enc_params);
```

This function is used to get the parameters of a video encoder channel after an alarm has triggered.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | Handle of an encoding channel. |
| *sub_chan_enc* | Specifies whether to set the parameters for the primary or secondary encoder. |
| *alarm_video_enc_params* | A pointer to a variable that will be filled with post-alarm encoder channel parameters when this function returns. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.2.10 sdvr_set_audio_encoder_channel_params

```
sdvr_err_e sdvr_set_audio_encoder_channel_params (sdvr_chan_handle_t handle,
                                    sdvr_audio_enc_chan_params_t * audio_enc_params);
```

This function is used to set the parameters of an audio encoder channel. There are no corresponding functions for setting the parameters for an audio decoding channel. This is because the decoding channel takes its parameters from the encoded stream.

The parameters of an audio encoding channel can only be changed when encoding is stopped.

**NOTE:** This feature is not implemented in this release. Only G.711 audio encoding is supported for this release.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | An encoding channel handle. |
| *audio_enc_params* | The parameters for the audio encoder channel. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.2.11 sdvr_get_audio_encoder_channel_params

```
sdvr_err_e sdvr_get_audio_encoder_channel_params (sdvr_chan_handle_t handle,
                                                  sdvr_audio_enc_chan_params_t * audio_enc_params);
```

This function is used to get the parameters of an audio encoder channel.

You must set the parameters before you can get them.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | An encoding channel handle. |
| *audio_enc_params* | A pointer to a variable that will be filled with encoder channel parameters when this function returns. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.2.12 sdvr_add_region

```
sdvr_err_e sdvr_add_region (sdvr_chan_handle_t handle,
                            sdvr_regions_type_e region_type,
                            sdvr_region_t * region);
```

Regions can be added to any encoding or decoding video channel for motion or blind detection, as well as privacy.

This function adds a new region type to a video channel.

After regions are created, you must enable them by calling `sdvr_enable_motion_detection`(), `sdvr_enable_blind_detection`(), or `sdvr_enable_privacy_regions`() as appropriate.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | An encoding or decoding video channel handle. |
| *region_type* | The type of region to be added. |
| *region* | A structure to specify the coordinates of the region as input. If the region was added successfully, the region_id field of the structure holds the region ID as output. This ID needs to be set to remove or change the coordinates of the region. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.2.13 sdvr_change_region

```
sdvr_err_e sdvr_change_region (sdvr_chan_handle_t handle,
                               sdvr_regions_type_e region_type,
                               sdvr_region_t * region);
```

This function is used to change the coordinates of a defined region for motion or blind detection, or privacy.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | An encoding or decoding video channel handle. |
| *region_type* | The type of region. |
| *region* | A structure that specifies the new coordinates of the region. The region_id field of the structure holds the region to be changed. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.2.14 sdvr_remove_region

```
sdvr_err_e sdvr_remove_region (sdvr_chan_handle_t handle,
                               sdvr_regions_type_e region_type,
                               sx_uint8 region_id);
```

This function is used to remove an existing region that was added for motion or blind detection, or privacy.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | An encoding or decoding video channel handle. |
| *region_type* | The type of region. |
| *region_id* | The region ID to be removed from the given region type. |

**Returns** `SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.2.15 sdvr_get_motion_detection

```
sdvr_err_e sdvr_get_motion_detection (sdvr_chan_handle_t handle,
                                      sdvr_motion_detection_t * motion_detection);
```

This function is used to get the motion threshold, motion detection state, and regions for a particular channel. A motion region is where motion is detected by the encoder. If no motion regions are defined, the entire video picture is used for motion detection.

Parameters

| Name | Definition |
|---|---|
| *handle* | An encoding or decoding video channel handle. |
| *motion_detection* | A pointer to a structure describing the motion threshold, regions, and enable state. |

**Returns** `SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.2.16 sdvr_get_blind_detection

```
sdvr_err_e sdvr_get_blind_detection (sdvr_chan_handle_t handle,
                                     sdvr_blind_detection_t * blind_detection);
```

This function is used to get the blind threshold and regions for a particular channel. A blind detection region is where constant video image is detected by the encoder in the specified region.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | An encoding or decoding channel handle. |
| *blind_detection* | A pointer to a structure describing the blind threshold, regions, and enable state. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.2.17 sdvr_get_privacy_regions

```
sdvr_err_e sdvr_get_privacy_regions (sdvr_chan_handle_t handle,
                                     sdvr_privacy_region_t * privacy_regions);
```

This function is used to get the privacy regions for a particular channel. Privacy regions are blanked in live and encoded video.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | Handle of an encoding or decoding channel. |
| *privacy_region* | A pointer to a data structure containing the region(s) to blanked, as well as their enable state(s). |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.2.18 sdvr_get_night_detection

```
sdvr_err_e sdvr_get_night_detection (sdvr_chan_handle_t handle,
                                     sdvr_night_detection_t * night_detection);
```

This function is used to get the threshold and enable state of night detection.

Night detection refers to the entire picture, and not to any particular region of the picture.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | An encoding or decoding channel handle. |
| *night_detection* | Pointer to a structure containing the threshold and enable state for night detection. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.2.19 sdvr_enable_privacy_regions

```
sdvr_err_e sdvr_enable_privacy_regions (sdvr_chan_handle_t handle, sx_bool enable);
```

This function is used to enable or disable all the privacy regions that are defined by calling the function `sdvr_add_region`(). After a privacy region is enabled, all the regions are blacked out in both the live/decoded and encoded video. If no privacy region is defined, the entire video picture will be blacked out.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | An encoding or decoding channel handle. |
| *enable* | A non-zero value enables privacy regions; zero disables privacy regions. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.2.20 sdvr_enable_motion_detection

```
sdvr_err_e sdvr_enable_motion_detection (sdvr_chan_handle_t handle, sx_bool enable,
                                         sx_uint8 threshold);
```

After regions are specified for motion detection by calling the function `sdvr_add_region`(), you must enable motion detection.

This function is used to enable and disable motion detection on all the specified regions. The motion detection uses the entire picture if no motion region is specified. As you enable the motion detection, you must specify a threshold above which motion should be detected.

After motion detection is enabled, the video alarm callback function is called every time motion is detected.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | An encoding or decoding channel handle. |
| *enable* | A non-zero value enables motion detection; zero disables motion detection. |
| *threshold* | The threshold value for motion detection. This field is ignored if you are disabling motion detection. The valid range is 0–99. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.2.21 sdvr_enable_blind_detection

```
sdvr_err_e sdvr_enable_blind_detection (sdvr_chan_handle_t handle, sx_bool enable,
                                        sx_uint8 threshold);
```

After regions are specified for blind detection by calling the function `sdvr_add_region`(), you must enable blind detection.

This function is used to enable and disable blind detection on all the specified regions. Blind detection uses the entire picture if no region is specified. As you enable the blind detection, you must specify a threshold above which blind detection is triggered.

After blind detection is enabled, the video alarm callback function is called every time video blind is detected.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | Handle of an encoding or decoding channel. |
| *enable* | A non-zero value enables blind detection; zero disables blind detection. |
| *threshold* | The threshold value for blind detection. This field is ignored if you are disabling blind detection. The valid range is 0–99. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.2.22 sdvr_enable_night_detection

```
sdvr_err_e sdvr_enable_night_detection (sdvr_chan_handle_t handle, sx_bool enable,
                                        sx_uint8 threshold);
```

This function is used to enable and disable night detection.

Night detection refers to the entire picture, and not to any particular region of the picture. As you enable the night detection, you must specify a threshold below which night should be detected.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | An encoding or decoding channel handle. |
| *enable* | A non-zero value enables night detection; zero disables night detection. |
| *threshold* | The threshold value for night detection. This field is ignored if you are disabling night detection. The valid range is 0–255. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

# 2.3 Encoding and Raw Audio/Video API

## 2.3.1 sdvr_enable_encoder

```
sdvr_err_e sdvr_enable_encoder (sdvr_chan_handle_t handle,
                                sdvr_sub_encoders_e sub_chan_enc,
                                sx_bool enable);
```

This function enables the Nth encoder on a particular channel. After an encoder is enabled for a particular channel, the associated video encoding on alarms will be activated.

Parameters

| Name | Definition |
|---|---|
| *handle* | An encoding channel handle. |
| *sub_chan_enc* | The encoder subchannel to enable or disable. |
| *enable* | If true, encoding is enabled; disabled otherwise. |

**Returns** `SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

**Remarks** After the encoder is enabled, both audio (if the channel has one) and video frames are sent.

**NOTE:** There are no audio frames for secondary encoders.

## 2.3.2 sdvr_get_av_buffer

```
sdvr_err_e sdvr_get_av_buffer (sdvr_chan_handle_t handle,
                               sdvr_frame_type_e frame_type,
                               sdvr_av_buffer_t ** frame_buffer);
```

This function is used to get one of:

- An encoded video frame from the encoder
- An encoded audio frame from the encoder
- A raw audio PCM frame

This function is called by the DVR Application to get a frame of the appropriate type from the SDK. Typically, the DVR Application registers a callback so that it can be informed when a frame is available. A callback, however, is not required. This function can be called at any time by the DVR Application to request a particular frame type. If the frame is not available, an error code indicates that no frame is available and the buffer pointer is NULL.

This is the typical flow of operation in the encoding path:

- DVR Application registers a AV frame callback using `sdvr_set_av_frame_callback`().
- When this callback function is called, the DVR Application notes the channels and the frame types available in a data structure.
- In a separate thread that saves encoded streams to disk, the DVR Application uses the information from the previous step to retrieve the encoded AV frames and saves them to disk.
- In a separate thread that displays live video or audio, the DVR Application uses the information saved earlier to retrieve the raw audio and video frames, and renders them on the display or plays them.

When this function is called, an `sdvr_av_buffer_t` * pointer to a frame is returned. The DVR Application should treat this buffer as read-only, and not modify any field of the structure. After the buffer is used, it must be released to the SDK using `sdvr_release_av_buffer`(). This release should happen as soon as possible because there are limited buffers in the SDK and released buffers are used to hold incoming data. If the DVR Application holds a buffer for too long, frame loss could result.

This function can be called repeatedly to retrieve frames for different channels. It can be called repeatedly for the same channel and it will return as many frames as are available before returning an error code indicating no more frames are available.

**What to do with the buffer header information?** The `sdvr_av_buffer_t` structure contains a header in addition to the frame buffer (payload). This header contains information about the size of the frame buffer, the type of the

frame, whether motion was detected in this frame, and so on. This information may be useful later to search through the stored video file to locate events quickly, e.g., frames where motion happened. We highly recommend that you store the header and the frame buffer. We also recommend that you save the header in a separate file (perhaps with other information) from the frame buffer file, so that you can search the header file to locate a particular section of encoded video and then jump to that location in the frame buffer file. The reserved field in the buffer data structure need not be saved. Also, the reserved field should not be changed or else `sdvr_release_av_buffer`() will have trouble recycling the buffer.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | An encoding channel handle. |
| *frame_type* | The type of frame you want.<br>`SDVR_FRAME_VIDEO_ENCODED_PRIMARY` - For an encoded video frame associated with the primary encoder.<br>`SDVR_FRAME_VIDEO_ENCODED_SECONDARY` - For an encoded video frame associated with the secondary encoder.<br>`SDVR_FRAME_AUDIO_ENCODED` - For an encoded audio frame<br>`SDVR_FRAME_RAW_AUDIO` - For a raw audio PCM frame |
| *frame_buffer* | The pointer to a pointer to an A/V buffer structure. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.3.3 sdvr_get_yuv_buffer

```
sdvr_err_e sdvr_get_yuv_buffer (sdvr_chan_handle_t handle,
                                sdvr_yuv_buffer_t ** frame_buffer);
```

This function is called by the DVR Application to get a raw video frame from the SDK. Typically, the DVR Application registers a callback so that it can be informed when a frame is available. A callback, however, is not required. This function can be called at any time by the DVR Application to request a particular frame type. If the frame is not available, an error code indicates that no frame is available and that the buffer pointer is NULL.

This is the typical flow of operation in an encoding path:

The DVR Application registers an AV frame callback using `sdvr_set_av_frame_callback()`.

When this callback function is called, the DVR Application notes the channels and the frame types available in a data structure.

In a separate thread that displays live video, the DVR Application uses the information saved earlier to retrieve the raw video frames and renders them on the display or plays them.

When this function is called, an `sdvr_yuv_buffer_t` pointer to a raw frame is returned. The DVR Application should treat this buffer as read-only and not modify any field of the structure. After the buffer is used, it must be released to the SDK using `sdvr_release_yuv_buffer()`. This release should happen as soon as possible because there are limited buffers in the SDK, and released buffers are used to hold incoming data. If the DVR Application holds a buffer for too long, frame loss could result.

This function can be called repeatedly to retrieve frames for different channels. It can be called repeatedly for the same channel and it will return as many frames as are available before returning an error code indicating no more frames are available.

**Parameters**

| Name | Definition |
|---|---|
| *handle* | An encoding or decoding channel handle. |
| *frame_buffer* | The pointer to a pointer to a YUV buffer structure. NULL if no YUV buffer is available. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.3.4 sdvr_release_av_buffer

```
sdvr_err_e sdvr_release_av_buffer (sdvr_av_buffer_t *frame_buffer);
```

This function is used to release an A/V frame to the SDK. Buffers should be released as quickly as possible to prevent frame loss.

Decoder buffers obtained from `sdvr_alloc_av_buffer()` will be released when you call `sdvr_send_av_frame()`. If you don't need to send the buffer after it is allocated, you must release the buffer by calling this function.

**NOTE:** To release a raw video YUV frame, you must call `sdvr_release_yuv_buffer`.

**Parameters**

| Name | Definition |
|---|---|
| *frame_buffer* | The pointer to a buffer obtained using `sdvr_get_av_buffer()` or `sdvr_alloc_av_buffer()`. You should not call this function if frame_buffer was already used in `sdvr_send_av_frame()`. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.3.5 sdvr_release_yuv_buffer

```
sdvr_err_e sdvr_release_yuv_buffer (sdvr_yuv_buffer_t *frame_buffer);
```

This function is used to release a YUV frame buffer to the SDK. Buffers should be released as quickly as possible to prevent frame loss.

**Parameters**

| Name | Definition |
|---|---|
| *frame_buffer* | The pointer to a buffer obtained using `sdvr_get_yuv_buffer()`. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.3.6 sdvr_get_buffer_channel

```
sdvr_chan_handle_t sdvr_get_buffer_channel (void *frame_buffer);
```

This function returns the channel handle corresponding to the given frame buffer.

**Parameters**

| Name | Definition |
|---|---|
| *frame_buffer* | The encoded or raw video frame from which to get its corresponding channel. |

**Returns**

The channel handle corresponding to the given frame buffer. A value of `INVALID_CHAN_HANDLE` indicates that the given buffer is invalid.

# 2.4 Decoding API

## 2.4.1 sdvr_enable_decoder

```
sdvr_err_e sdvr_enable_decoder (sdvr_chan_handle_t handle, sx_bool enable);
```

This function enables decoding using a particular channel. This function must be called every time a new playback session is started and ended.

Each decoded video frame can either be displayed on the SMO by calling `sdvr_set_smo_grid`() or be sent to the DVR Application for display on the host monitor by calling `sdvr_stream_raw_video`().

We recommend that you call `sdvr_set_smo_grid`() and `sdvr_stream_raw_video`() before enabling the decoder so that no decoded video is lost.

Parameters

| Name | Definition |
|---|---|
| *handle* | A decoding channel handle. |
| *enable* | If true, then decoding is enabled; otherwise, disabled. |

Returns

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.4.2 sdvr_alloc_av_buffer

```
sdvr_err_e sdvr_alloc_av_buffer (sdvr_chan_handle_t handle,
                                 sdvr_av_buffer_t ** frame_buffer);
```

This function is used to get a buffer from the SDK. To decode video, the DVR Application first needs to call this function to get an empty buffer. The application then copies encoded video from the disk into this buffer. Then it can be sent to the board for decoding using `sdvr_send_av_frame`().

Each buffer can hold one encoded AV frame. For normal speed playback, the DVR Application needs to send 30 frames per second to the decoder.

The decoded data can be retrieved from the board using `sdvr_get_av_buffer`() using the channel number that was used for decoding.

Buffers allocated this way should be returned as quickly as possible to the SDK, so that the decoder is not starved of data.

The returned buffer has the board and channel numbers, and other information embedded in the buffer data structure. This information has to be treated as read-only. Only the *payload* field in the `sdvr_av_buffer_t` structure can be modified by the DVR Application.

**How to fill up the AV buffer?** The buffer returned by this function is of type `sdvr_av_buffer_t` and has a header and location for the frame buffer (payload). The frame buffer can be filled with the encoded data from disk. If you stored the header for the encoded buffer (see the description of `sdvr_get_av_buffer`()), then you can copy the header into the header for this buffer. You have to be careful, however, not to overwrite the reserved field of the buffer returned by this function, as the reserved field is used by the SDK when you call `sdvr_send_av_frame`().

**NOTE:** The board index, channel number, and channel types will be set to match the given channel handle upon successful return.

Parameters

| Name | Definition |
|---|---|
| *handle* | A decoding channel handle. |
| *frame_buffer* | A pointer to the buffer variable that is set when this function returns. |

Returns

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.4.3 sdvr_send_av_frame

```
sdvr_err_e sdvr_send_av_frame (sdvr_av_buffer_t *frame_buffer);
```

The buffer obtained using `sdvr_alloc_av_buffer()` is filled with encoded data by the DVR Application, and then the buffer is sent to the decoder for decoding using this function.

**NOTE:** Sending the buffer for decoding also implicitly returns the buffer to the SDK so that it can be used again in a future call to allocate a buffer.

**Parameters**

| Name | Definition |
|---|---|
| *frame_buffer* | Pointer to the buffer to be sent. The channel number, board number, and so on are already a part of the buffer data structure. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

# 2.5 Display and Sound API

## 2.5.1 sdvr_stream_raw_video

```
sdvr_err_e sdvr_stream_raw_video (sdvr_chan_handle_t handle,
                                  sdvr_video_res_decimation_e res_decimation,
                                  sx_uint8 frame_rate, sx_bool enable);
```

This function is used to enable streaming of raw video for a channel to the host. If the channel is an encoding channel, then the raw video is from the camera. If it is a decoding channel, the raw video is the decoded video.

This function can be called multiple times to enable and disable streaming.

Parameters

| Name | Description |
|---|---|
| *handle* | An encoding or decoding channel handle. |
| *res_decimation* | Specifies whether the video is streamed at the system-wide maximum resolution, 1/4, or 1/16 of that resolution. |
| *frame_rate* | Specifies the frame rate at which the raw video frames are being sent to the host application. In general this value should be 30. The valid range is 1–30. |
| *enable* | If true, streaming is enabled; otherwise, disabled. |

**Returns** `SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.5.2 sdvr_stream_raw_audio

```
sdvr_err_e sdvr_stream_raw_audio (sdvr_chan_handle_t handle, sx_bool enable);
```

This function is used to enable streaming of raw audio for a channel to the host. If the channel is an encoding channel, then the raw audio is from the microphone. If it is a decoding channel, the raw audio is the decoded audio.

The raw audio can only be enabled on a channel that is streaming raw video. Otherwise, this function is ignored. After the raw audio is enabled on a channel, encoded audio cannot be streamed.

This function can be called multiple times to enable and disable streaming.

**Parameters**

| Name | Description |
|---|---|
| *handle* | An encoding or decoding channel handle. |
| *enable* | If true, streaming is enabled; otherwise, disabled. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

# 2.6 On-Screen Display API

## 2.6.1 sdvr_set_osd_text

```
sdvr_err_e sdvr_set_osd_text (sdvr_chan_handle_t handle,
                              sdvr_osd_text_config_t * osd_text_config);
```

This function is used to set ASCII OSD text for OSD display. Additionally, you can specify whether to append a time stamp to the displayed text.

Calling this function multiple times causes the OSD information from the previous call to be overwritten by the information from the latest call.

This function only supports one OSD text item of maximum 10 characters long. `sdvr_osd_text_config_ex`() supports multiple OSD text items of up to 100 Unicode characters.

**Parameters**

| Name | Description |
|---|---|
| *handle* | An encoding channel handle. |
| *osd_text_config* | This data structure defines the text and its position for display over the video for the given channel, as well as whether a time stamp should be appended to the text. You can also specify whether OSD should be enabled for this channel. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.6.2 sdvr_get_osd_text

```
sdvr_err_e sdvr_get_osd_text (sdvr_chan_handle_t handle,
                              sdvr_osd_text_config_t * osd_text_config);
```

This function is used to retrieve the current OSD configuration and status.

Parameters

| Name | Description |
|---|---|
| *handle* | An encoding channel handle. |
| *osd_text_config* | A structure describing the position of the string and its contents, whether a time stamp is appended to the string, and the enable status of OSD on this channel |

**Returns** `SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.6.3 sdvr_enable_osd_text

```
sdvr_err_e sdvr_enable_osd_text (sdvr_chan_handle_t handle, sx_bool enable);
```

This function is used to enable the OSD display that was set by `sdvr_set_osd_text()`.

**Parameters**

| Name | Description |
|---|---|
| *handle* | An encoding channel handle. |
| *enable* | If true, OSD is enabled; otherwise, disabled. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.6.4 sdvr_osd_text_config_ex

```
sdvr_err_e sdvr_osd_text_config_ex (sdvr_chan_handle_t handle, sx_uint8 osd_id,
                                    sdvr_osd_config_ex_t * osd_text_config);
```

This function is used to configure the OSD text for an OSD item. The text associated with each OSD item can be up to 100 characters long but the actual display is limited to the size of the video and the starting location of the text. Additionally, you can specify whether to append a time stamp to the displayed text. Appending a time stamp reduces the number of characters that can be displayed for your OSD item.

After you configure an OSD item, you can show or hide its display state by calling `sdvr_osd_text_show`(). The original state of the OSD item is hidden.

**Parameters**

| Name | Description |
|---|---|
| *handle* | An encoding or decoding channel handle. |
| *osd_id* | The OSD item to be configured. Currently there can be two OSD items per each channel. ID zero corresponds to the first OSD, 1 to the second, and so on. The valid range is 0–1. |
| *osd_text_config* | This data structure defines the text and its position for display over the video for the given channel, as well as whether a timestamp should be appended to the text. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.6.5 sdvr_osd_text_show

```
sdvr_err_e sdvr_osd_text_show (sdvr_chan_handle_t handle, sx_uint8 osd_id,
                               sx_bool show);
```

This function is used to control the display state of an OSD item for a given channel. The OSD text is displayed as soon as encoding, raw video, or SMO streaming is enabled for the current channel handle. Changing the video streaming has no affect on the display state of an OSD item for the given channels.

This function must be called only after an OSD item is configured by calling `sdvr_osd_text_config_ex`() for the given channel handle.

**Parameters**

| Name | Description |
|---|---|
| *handle* | An encoding or decoding channel handle. |
| *osd_id* | The OSD item to show or hide. Currently each channel is limited to two OSD items. ID zero corresponds to the first OSD, 1 to the second, and so on. The valid range is 0–1. |
| *show* | If true, OSD item is displayed; otherwise, it is not displayed. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.6.6 sdvr_osd_set_font_table

```
sdvr_err_e sdvr_osd_set_font_table (sdvr_font_table_t *font_desc);
```

Use this function to specify a different OSD text display font table than the default ASCII font. Currently, Bitmap Distribution Format (BDF) is the only supported font format. BDF by Adobe is a file format for storing bitmap fonts. BDF is most commonly used font file within the Linux operation system.

The new font table is used for all DVR boards connected at the time `sdvr_osd_set_font_table()` is called. You can either choose to use all the characters within the font table or a subset of it. To use all the characters, set the `start_font_code` parameter to 0 and the `end_font_code` parameter to 65536. The specified YUV font color is used for all characters within the table.

**Parameters**

| Name | Description |
|---|---|
| *font_desc* | A pointer to font descriptor data structure defining the new font table to load. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

**Remarks**

You should only call this function once after you have connected to all the DVR boards.

## 2.6.7 sdvr_osd_use_font_table

```
sdvr_err_e sdvr_osd_use_font_table (sx_uint8 font_id);
```

After you down load font tables into the firmware by calling `sdvr_osd_set_font_table()`, you can select which font table to use. By default, the last font table downloaded is selected. If no font table is specified, the default English font table is used.

Font table IDs of less than 8 are reserved for pre-defined system fonts. Currently, there is only one system font ID `SDVR_FT_FONT_ENGLISH` defined.

**Parameters**

| Name | Description |
|---|---|
| *font_id* | The font table ID to be selected. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

**Remarks**

The given font table is used for all DVR boards connected at the time this function is called.

# 2.7 Spot Monitor Output API

## 2.7.1 sdvr_set_smo_grid

```
sdvr_err_e sdvr_set_smo_grid (sdvr_chan_handle_t handle,
                              sdvr_smo_grid_t * smo_grid);
```

This function is used to set the Spot Monitor Output (SMO) grid pattern.

You can divide the SMO into different grid patterns. Each tile of the SMO grid is defined by its left top most coordinate, in pixels, and by the decimation of its original video resolution.

The grid pattern on a spot monitor is flexible and can be defined in various patterns (such as 4x4, 3x3, or 8x8 grids) as it applies to the resolution of the connected monitor.

Each tile of the grid is described by an `sdvr_smo_tile_t` data structure.

Each tile can either be enabled or disabled. If you have more than one channel in one tile position, you must also specify a dwell time, that is, the amount of time each channel is displayed before switching to the next channel. If you only have one channel assigned to a tile, that channel is permanently displayed while it is enabled.

**NOTE:** You can include encoder and decoder channels in the SMO tile. For encoder channels, the live video is displayed. For decoder channels, the decoded (playback) video is displayed.

You can assign multiple channels to a tile, but a specific channel cannot be assigned to different tiles.

Assuming your SMO is an NTSC monitor of 720x480 resolution, and you want to specify 16 tiles arranged in a 4x4 grid pattern, then each tile is 180x120. The first left top most grid is (0,0) the next one is (180,120), and so on. The resolution decimation for each grid is 1/4 of the camera resolution.

**NOTE:** The top left coordinates of each tile must be an even number (i.e., coordinates (0,5) or (1,2) are invalid).

**Parameters**

| Name | Description |
|---|---|
| *handle* | An encode or decode video channel handle whose output is displayed on this grid. |
| *smo_grid* | A pointer structure defining one grid on an SMO. |

**Returns** `SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

**Remarks** The SMO grid is not displayed at all if it does not fit completely on the screen.

## 2.7.2 sdvr_get_smo_grid

```
sdvr_err_e sdvr_get_smo_grid (sdvr_chan_handle_t handle,
                              sdvr_smo_grid_t * smo_grid);
```

This function is used to get the current SMO grid configuration.

Parameters

| Name | Description |
|---|---|
| *handle* | An encode or decode video channel handle whose output is displayed on this grid. |
| *smo_grid* | A pointer structure defining one grid on an SMO. |

**Returns** `SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

# 2.8 RS485 Communication API

## 2.8.1 sdvr_init_uart

```
sdvr_err_e sdvr_init_uart (sx_uint32 board_index, sx_uint32 baud_rate,
                           sx_uint8 data_bits, sx_uint8 stop_bits,
                           sx_uint8 parity_enable, sx_uint8 parity_even);
```

This function is used to initialize the UART port on the S6 to talk to the RS-485.

**Parameters**

| Name | Description |
|---|---|
| *board_index* | The board number. This is a zero-based number; that is, the first board number is index zero, the second board number is index one, and so on. |
| *baud_rate* | The output baud rate. Must be between 50 and 115200. |
| *data_bits* | Number of data bits. Valid values are 5–8. |
| *stop_bits* | Number of stop bits. Valid values are 1 and 2. |
| *parity_enable* | If this field is set to zero, parity is disabled. If set to a non-zero value, parity is enabled. |
| *parity_even* | If this filed is set to zero, odd parity is used. If set to a non-zero value, even parity is used. This field is ignored if parity is disabled. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.8.2 sdvr_write_uart

```
sdvr_err_e sdvr_write_uart (sx_uint32 board_index, sx_uint8 count, sx_uint8 *data);
```

This function is used to write a number of 8-bit characters to the RS485 port.

**NOTE:** This function is not implemented on this release.

**Parameters**

| Name | Description |
|---|---|
| *board_index* | The board number. This is a zero-based number; that is, the first board number is index zero, the second board number is index one, and so on. |
| *count* | Number of bytes to transmit. |
| *data* | The data bytes to transmit. The number of valid bytes in the array is determined by the *count* field. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

## 2.8.3 sdvr_read_uart

```
sdvr_err_e sdvr_read_uart (sx_uint32 board_index, sx_uint8 *data_count_read,
                           sx_uint8 max_data_size, sx_uint8 *data);
```

This function is used to read up to the requested number of 8-bit characters from the RS485 port.

You must initialize the UART by calling `sdvr_init_uart`() and send commands to receive data before calling this function.

It is possible for the data not to be ready on the port at the time of calling this function. In this case, you need to call this function multiple times while the actual number of bytes received is zero or is less than the expected amount.

**Parameters**

| Name | Description |
|---|---|
| *board_index* | The board number. This is a zero-based number; that is, the first board number is index zero, the second board number is index one, and so on. |
| *data_count_read* | A pointer to a variable that holds the actual number of characters read upon successful return. This count is less than or equal to `max_data_size`. A value of zero means there is no data available at the port. |
| *max_data_size* | The maximum number of bytes requested to be read from the board.<br>**NOTE:** The actual returned bytes may be less than the number requested. |
| *data* | A character pointer long enough to hold the maximum number of bytes requested. |

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

# 2.9 Sensors and Relays API

## 2.9.1 sdvr_trigger_relay

```
sdvr_err_e sdvr_trigger_relay (sx_uint32 board_index, sx_uint32 relay_num,
                               sx_bool is_triggered);
```

This function is used to trigger relays on each SDVR board.

**Parameters**

| Name | Description |
|---|---|
| *board_index* | Index of the board whose relay you want to set. This is a zero-based number; that is, the first board number is index zero, the second board number is index one, and so on. |
| *relay_num* | The number of the relay—a board with M relays has the relays numbered sequentially from 0 to M-1 (see schematics for position of relay number 0, 1, and so on). You can determine the number of relays from the board configuration data structure. |
| *is_triggered* | If true, the relay is triggered. If false, the relay is reset. |

**Returns**

`SDVR_ERR_NONE` - On success.

## 2.9.2 sdvr_enable_sensor

```
sdvr_err_e sdvr_enable_sensor (sx_uint32 board_index, sx_uint32 sensor_num,
                               sx_bool enable);
```

This function is used to enable or disable a sensor on each SDVR board.

Parameters

| Name | Description |
|---|---|
| *board_index* | Index of the board whose sensor you want to set. This is a zero-based number; that is, the first board number is index zero, the second board number is index one, and so on. |
| *sensor_num* | The number of the sensor—a board with M sensors has the sensors numbered sequentially from 0 to M-1 (see schematics for position of sensors number 0, 1, etc.). You can determine the number of sensors from the board configuration data structure. |
| *enable* | If true, the sensor is enabled, and if triggered you are notified. If false, the sensor is disabled and no notification is sent to the DVR Application. Default status of the sensor is enabled. |

**Returns**

`SDVR_ERR_NONE` - On success.

`SDVR_ERR_BOARD_NOT_CONNECTED` - Error code if the board was never connected.

`SDVR_ERR_INVALID_ARG` - Error code if the given board index does not exist.

## 2.9.3 sdvr_config_sensors

```
sdvr_err_e sdvr_config_sensors (sx_uint32 board_index, sx_uint32 sensor_enable_map,
                                sx_uint32 edge_triggered_map);
```

This function is used to specify how each sensor should be triggered, as well as how to enable or disable each of the sensors on each SDVR board.

Sensors can be either edge-triggered or level-sensitive. You can specify the method to use to determine whether a sensor is triggered.

A notification message will be sent to the DVR Application for every sensor that is enabled and is triggered. The sensors that are disabled receive no notification messages.

**Parameters**

| Name | Description |
|---|---|
| *board_index* | Index of the board whose sensors you want to set. This is a zero-based number; that is, the first board number is index zero, the second board number is index one, and so on. |
| *sensor_enable_map* | A bit map corresponding to each sensor to indicate whether it is enabled. The least most significant (LMS) bit of this map corresponds to the first sensor, the next LMS bit to the second sensor, and so on. You can determine the number of sensors from the board configuration data structure. The value of 1 for a bit means that the sensor is enabled for the corresponding sensor, and 0 means it is disabled. Default status of the sensors is enabled. |
| *edge_triggered_map* | A bit map corresponding to each sensor to indicate whether it is edge triggered. The least most significant (LMS) bit of this map corresponds to the first sensor, the next LMS bit to the second sensor, and so on. You can determine the number of sensors from the board configuration data structure. The value of 1 for a bit means that the sensor is edge triggered for the corresponding sensor, and 0 means it is level sensitive. |

**Returns**

`SDVR_ERR_NONE` - On success.

`SDVR_ERR_BOARD_NOT_CONNECTED` - Error code if the board was never connected.

`SDVR_ERR_INVALID_ARG` - Error code if the given board index does not exist.

# 2.10 Recording to File API

## 2.10.1 sdvr_start_recording

```
sdvr_err_e sdvr_start_recording (sdvr_chan_handle_t handle,
                                 sdvr_sub_encoders_e sub_chan_enc,
                                 char * file_name);
```

This function saves the encoded audio and video frames for any of the subencoders of the specified encoder channel into the given file name. The generated file is in .mov format.

Calling this function does not affect the current enable state of the encoder. After calling this function, you must call `sdvr_enable_encoder`() to enable the subencoder of the given encoder channel.

Calling this function multiple times consecutively results in writing the audio and video frames into the new file without restarting the encoder. Hence, there will not be any lose of frames. This function ensures the new recorded file starts with an I-frame.

**NOTE:** Not implemented in this release.

Call `sdvr_stop_recording`() to stop saving encoded frames. Additionally, if you need to stop the encoder, you must call `sdvr_enable_encoder`().

**NOTE:** This function is not supported and maybe changed in the next release.

**Parameters**

| Name | Description |
|---|---|
| *handle* | The channel handle for which to record encoded audio and video frames. |
| *sub_chan_enc* | The encoder subchannel for which to save encoded audio and video frames. |
| *file_name* | The full file name path to record audio/video frames. |

**NOTE:** This function always truncates the given file before recording new audio and video frames.

**Returns**

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

**Remarks**

Even though this function is called to save the encoded frames, the DVR Application is still notified by the callback function as the audio and video frames arrive. You can choose to ignore these frames or call `sdvr_get_av_buffer`() to get the frames and perform additional processing.

## 2.10.2 sdvr_stop_recording

```
sdvr_err_e sdvr_stop_recording (sdvr_chan_handle_t handle,
                                sdvr_sub_encoders_e sub_chan_enc);
```

Call this function to stop saving the encoded audio and video frames for the subencoder of the specified encoder channel that is recording.

Calling this function does not affect the current enable state of the encoder. After calling this function, you must call `sdvr_enable_encoder`() to stop the subencoder of the given encoder channel.

**NOTE:** This function is not supported and maybe changed in the next release.

Parameters

| Name | Description |
|---|---|
| *handle* | The channel handle for which to stop recording encoded audio/video frames. |
| *sub_chan_enc* | The encoder subchannel to stop recording. |

Returns

`SDVR_ERR_NONE` - On success. Otherwise, see the error code list.

# Index

## A



## B



## C



## D

## E



## F

## T

## V

Stretch Inc.
1322 Orleans Drive
Sunnyvale, CA 94089
tel 408.543.2700 • fax 408. 747.5736
www.stretchinc.com