

System Integration Guide

ONVIF Network Video Transmitter Suite with Edge Analytics

Introduction

Synesis ONVIF Network Video Transmitter (NVT) Suite with Edge Analytics enables security solution providers to release intelligent video surveillance devices such as IP-cameras and video encoders compliant with the global standard ONVIF. The customer quickly creates its differentiating IP while reducing the time to market and avoiding the costs of ONVIF protocol stack maintenance. The suite provides an abstraction layer for platform dependent services and libraries.

Rich feature set based on industry standard

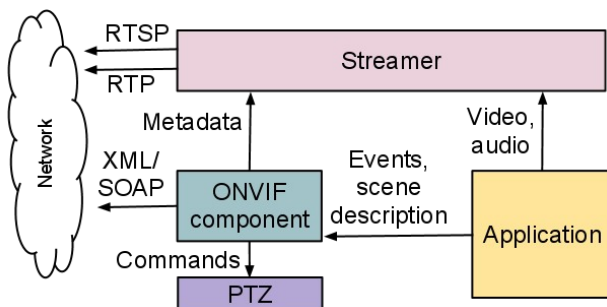
The following feature blocks are available in Synesis reference designs based on the ONVIF NVT suite:

- Network device discovery
- Live video and audio streaming
- Video capture configuration
- Video compression configuration
- Video analytics configuration
- Event and metadata configuration
- Rule management for alerts
- Secure firmware update (with optional software copy protection)
- Local storage / network storage recording
- PTZ camera control

The present implementation matches ONVIF version 1.02 while certain advanced features are designed in consideration of the ONVIF 2.0. The standard conformance is validated by the community test tools, third-party VMS integration and internal QA.

Intelligent video pipeline

The suite is integrated with a camera calibrator, professional video analytics, tampering detectors, various digital filters and a range of codes. These extensions are customizable.



ONVIF NVT software framework

The modular and flexible architecture of the middleware suite provides cost-effective software customization, third-party integration and maintenance. The suite consists of the following components as shown in the Figure:

- **ONVIF** component serves ONVIF requests from network video clients (NVC).
- **Streamer** transmits multimedia by RTSP/RTP protocol.
- **User application** provides access to a source of video and/or audio data. It also can contain implementation of data processing such as filtering, video analytics, event raising, etc.
- **PTZ** sends PTZ commands via serial port

ONVIF and streamer are portable components, user application is platform dependent.

The IP device can be integrated with legacy video management systems (without ONVIF support) by mapping ONVIF calls onto a custom protocol handler. In this way, alternative protocols such as PSIA can be supported.

For multiple channel configurations, the middleware is integrated with H.264 AVC Main Profile codec from Ittiam. Third-party codecs are supported via the TI's Codec Engine (xDAIS) interface. Video analytics and metadata streaming is available on all channels.

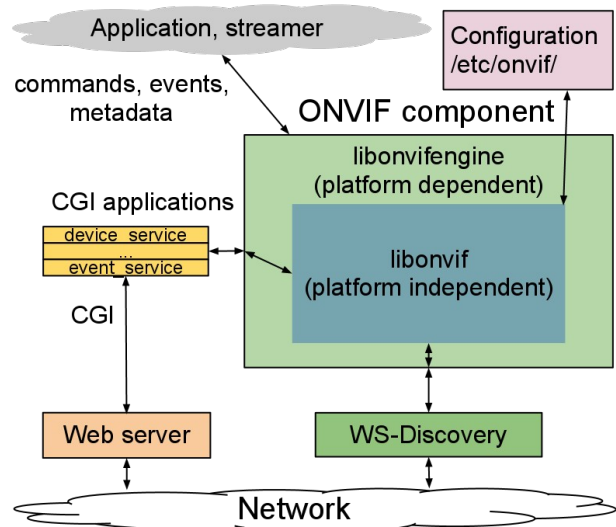


Table of Contents

System Integration Guide

ONVIF Network Video Transmitter Suite with Edge Analytics.....	1
Introduction.....	1
Rich feature set based on industry standard.....	1
Intelligent video pipeline.....	1
ONVIF NVT software framework	1
Components.....	3
Components API.....	3
ONVIF.....	3
Streamer.....	4
VideoEncoder.....	5
PTZ.....	5
Relay.....	7
System.....	7
SystemInfo.....	7
VideoAnalytics.....	8
ONVIF platform independent component.....	9
libonvif API.....	9
Middleware API.....	11
Third-party dependencies.....	16
MontaVista products.....	16
Texas Instruments products.....	16
Other products.....	16

Components

System is divided onto few components, which interact only via documented API. Every component may be replaced with custom implementation. Some of components are optional.

Interface	Functionality	Implementation	Optional	Included to ONVIF NVT Suite
ONVIF	Serve soap requests Store system settings Serialize and deliver events Serialize metadata	libonvif, libonvifengine	No	Yes
Streamer	Deliver audio, video and metadata streams via RTSP	libstreamer	No	Yes
Encoder	Check supported codecs and resolutions Initialize and start audio, video and metadata streaming Configure video sensor	Application	No	Yes
PTZ	Send PTZ commands to connected cameras	libptz	Yes	Yes
Relay	Trigger relays, mounted on the device	Application	Yes	Yes
System	Execute console command	Application	No	Yes
SystemInfo	Get CPU usage, memory load and other information	Application	Yes	Yes
VideoAnalytics	Configure, start and stop video analytics	Application	Yes	Linux sources DSP binaries

Components API

This section gives brief listing of the components API. Refer doxygen documentation for full information.

ONVIF

```
/**
 *
 */
bool virtual initialize(ISystem* system, IEncoder* encoder, IStreamerInterface* streamer) = 0;

/**
 * Creates a thread and launches message loop inside it.
 */
bool virtual start() = 0;

/**
 * Stops processing ONVIF requests, releases resources and terminates the thread.
 */
bool virtual stop() = 0;

/**
 * Serializes metadata. Returns pointer to buffer containing serialized representation of metadata and
 * stores its size in metadata_size parameter.
 * The returned pointer is valid until the next call.
 */
virtual const char* serialize_metadata(metadata_t* metadata, int* metadata_size) = 0;

/**
 * Delivers event raised outside of Onvif to interested subscribers.
 */
bool virtual deliver_event(event_t* event) = 0;

/**
```

```

    * Takes a reference to PTZ component
    */
bool virtual set_ptz_interface(IPTZInterface* interface) = 0;

/**
 * Takes a reference to Relay control component
 */
bool virtual set_relay_interface(IRelay* interface) = 0;

/**
 * Takes a reference to Video Analytics control component
 */
bool virtual set_video_analytics_interface(IVideoAnalytics* interface) = 0;

```

Streamer

```

/**
 * \brief Start stream session
 *
 * \param stream_id    stream identifier
 * \param stream_name  name of stream
 * \param transport    type of transport
 * \param multicast    is multicast enabled
 * \param video        video encoder params
 * \param audio        audio encoder params
 * \param metadata     metadata params
 *
 * \return status
 * \retval  0          On success
 * \retval -1          On failure
 */
virtual int start_stream (int stream_id, char *stream_name, transport_t transport, bool multicast,
                        video_params_t video, audio_params_t audio, meta_params_t metadata) = 0;

/**
 * \brief Stop stream session
 *
 * \param stream_id    stream identifier
 *
 * \return status
 * \retval  0          On success
 * \retval -1          On failure
 */
virtual int stop_stream (int stream_id) = 0;

public:
/**
 * \brief put frame in to stream
 *
 * \param stream_id    stream identifier
 * \param frame_type   type of frame
 * \param buff         pointer to data frame
 * \param size         size of data frame
 * \param timestamp    frame timestamp
 *
 * \return status
 * \retval  0          On success
 * \retval -1          On failure
 */
virtual int put_frame (int stream_id, frame_type_t frame_type, char *buff, unsigned int size, uint64_t timestamp) = 0;

public:
/**
 * \brief Add user to RTSP stream server
 *
 * \param username     user name
 * \param password     user password
 *
 * \return status
 * \retval  0          On success
 * \retval -1          On failure
 */
virtual int user_add (char const *username, char const *password) = 0;

/**
 * \brief Delete user from RTSP stream server
 *
 * \param username     user name
 *
 * \return status
 * \retval  0          On success
 * \retval -1          On failure
 */
virtual int user_del (char const *username) = 0;

/**
 * \brief Delete all users from RTSP stream server
 *
 * \return status
 * \retval  0          On success
 * \retval -1          On failure
 */
virtual int user_clear (void) = 0;

```

VideoEncoder

```
/**
 * Insert I-frame to all video stream(s) on a given channel
 *
 * \param ch Channel to insert I-frames on
 * \retval 0 OK
 * \retval -1 Failed
 */
virtual int set_sync_point(int ch) = 0;

/**
 * Set sensor settings on a given channel
 * \param ch Channel to set settings on.
 * \param settings SensorSettings structure.
 * \retval 0 OK
 * \retval -1 Failed
 */
virtual int set_sensor_settings(int ch, sensor_settings_t* sensor_settings) = 0;

/**
 * Returns a value indicating if the input/output video params compatible with the given channel and encoder
 * \param ch Channel to check.
 * \param videnc Type of video encoder.
 * \param inWidth Input (from a sensor or other video source) video width.
 * \param inHeight Input (from a sensor or other video source) video height.
 * \param outWidth Output (via RTSP stream) video width
 * \param outHeight Output (via RTSP stream) video height
 * \retval 1 Validated
 * \retval 0 Not validated
 */
virtual int validate_video_encoder(int ch, int videnc, int inWidth, int inHeight, int outWidth, int outHeight) = 0;

/**
 * Setup or change streaming parameters
 * \param stream_id Stream id to change or setup. If stream_id == -1 then allocate new stream else update existing
 * \param ch Channel to setup stream on.
 * \param video_source_settings Input video parameters.
 * \param video_encoder_settings Video encoder parameters.
 * \param audio_settings Audio parameters.
 * \param metadata_settings Metadata parameters.
 * \retval Stream id
 * \retval 0 Not validated
 */
virtual int setup_streaming(int stream_id, int ch,
                           video_source_settings_t* video_source_settings,
                           video_encoder_settings_t* video_encoder_settings,
                           audio_encoder_settings_t* audio_settings,
                           metadata_settings_t* metadata_settings) = 0;

/**
 * Get supported resolutions on a given channel
 * \param ch Channel to get resolutions from.
 * \param list Supported resolution array to fill.
 * \param size Pointer to interger to store size of list array
 * \retval 0 Operation was successfull
 * \retval -1 Operation could not be completed
 */
virtual int get_supported_resolutions(int ch, const resolution_t** list, int* size) = 0;

/**
 * Get supported streams number on each channel
 * \retval 0 Number of streams supported by each channel
 */
virtual int get_streams_per_channel_number() = 0;

/**
 * Get input standard on a channel
 * \param ch Channel to get standard from.
 * \retval Value indicating current standard
 */
virtual tvstandard_t get_current_tvstandard(int ch) = 0;
```

PTZ

```
/**
 * Set PTZ communication protocol
 *
 * \param protocol Protocol to set
 * \return Operation status
 * \retval 0 OK
 * \retval -1 Failed
 */
virtual int set_interface_protocol(int protocol) = 0;

/**
 * Set hardware device to communicate with PTZ devices
 *
 * \param dev Pointer to device name
 * \return Operation status
 * \retval 0 OK
 * \retval -1 Failed
 */
virtual int set_interface_device (const char* dev) = 0;
```

```

/**
 * Set communication speed
 *
 * \param speed Speed to set
 * \return Operation status
 * \retval 0 OK
 * \retval -1 Failed
 */
virtual int set_interface_speed (int speed) = 0;

/**
 * Move PTZ node to absolute position
 *
 * \param addr Address of PTZ node
 * \param position Vector with absolute coordinates
 * \param speed Vector with speed values
 * \param limits Vector with coordinate limits
 * \param converter Values converter to use
 * \return Operation status
 * \retval 0 OK
 * \retval -1 Failed
 */
virtual int move_absolute (int addr, vector_t* position, vector_t* speed = NULL, space_t* limits = NULL, int converter =
GENERIC) = 0;

/**
 * Move PTZ node to relative position
 *
 * \param addr Address of PTZ node
 * \param position Vector with relative coordinates
 * \param speed Vector with speed values
 * \param limits Vector with coordinate limits
 * \param converter Values converter to use
 * \return Operation status
 * \retval 0 OK
 * \retval -1 Failed
 */
virtual int move_relative (int addr, vector_t* position, vector_t* speed = NULL, space_t* limits = NULL, int converter =
GENERIC) = 0;

/**
 * Move PTZ node continuously
 *
 * \param addr Address of PTZ node
 * \param speed Vector with velocity values
 * \param timeout Time to stop movement after
 * \param limits Vector with coordinate limits
 * \param converter Values converter to use
 * \return Operation status
 * \retval 0 OK
 * \retval -1 Failed
 */
virtual int move_continuous (int addr, vector_t* speed, int timeout, space_t* limits = NULL, int converter = GENERIC) = 0;

/**
 * Stop PTZ node movement
 *
 * \param addr Address of PTZ node
 * \param params Vector with stop parameters
 * \return Operation status
 * \retval 0 OK
 * \retval -1 Failed
 */
virtual int stop (int addr, vector_t* params) = 0;

/**
 * Get PTZ node status
 *
 * \param addr Address of PTZ node
 * \param status Pointer to status structure to be filled
 * \param converter Values converter to use
 * \return Operation status
 * \retval 0 OK
 * \retval -1 Failed
 */
virtual int get_status (int addr, status_t* status, int converter = GENERIC) = 0;

/**
 * Store PTZ node position as preset
 *
 * \param addr Address of PTZ node
 * \param preset Preset index to store position at
 * \param converter Values converter to use
 * \return Operation status
 * \retval 0 OK
 * \retval -1 Failed
 */
virtual int set_preset (int addr, int preset, vector_t* position, int converter = GENERIC) = 0;

/**
 * Move to PTZ node preset
 *
 * \param addr Address of PTZ node
 * \param preset Preset index to move
 * \param speed Vector with speed values
 * \param converter Values converter to use
 */

```

```

* \return Operation status
* \retval 0 OK
* \retval -1 Failed
*/
virtual int goto_preset (int addr, int preset, vector_t* speed = NULL, int converter = GENERIC) = 0;

/**
* Send auxiliary command to PTZ node
*
* \param addr Address of PTZ node
* \param cmd Pointer command string
* \param response PTZ node response
* \param size Response size
* \return Operation status
* \retval 0 OK
* \retval -1 Failed
*/
virtual int send_auxiliary_command(int addr, const char* cmd, char* response, int size) = 0;

```

Relay

```

/**
* Setup relay operation mode
* \param index Index of a relay to setup
* \param mode Relay mode to set
* \param idle_state Idle mode to set
* \param timeout Timeout value to use when realy is in a monostable mode
* \retval 0 Operation was successfull
* \retval -1 Operation could not be completed
*/
virtual int set_relay_settings(int index, relay_mode_t mode, relay_idle_state_t idle_state, int timeout) = 0;

/**
* Change relay state
* \param index Index of a relay to setup
* \param state State of the relay to switch to
* \retval 0 Operation was successfull
* \retval -1 Operation could not be completed
*/
virtual int set_relay_state(int index, relay_state_t state) = 0;

```

System

```

/**
* Reboot the system
*/
virtual void system_reboot() = 0;

/**
* Execute command in system shell
* \param cmd Pointer to string with command
* \retval 0 Operation was successfull
* \retval -1 Operation could not be completed
*/
virtual int exec_command(const char *cmd) = 0;

```

SystemInfo

```

/**
* Save dsp usage in system info
* \param dsp_usage Value to save
* \retval 0 Operation was successfull
* \retval -1 Operation could not be completed
*/
virtual int store_dsp_usage(unsigned int dsp_usage) = 0;

/**
* Get CPU usage
* \retval CPU usage value
*/
virtual unsigned int get_cpu_usage() const = 0;

/**
* Get dsp usage
* \retval DSP usage value
*/
virtual unsigned int get_dsp_usage() const = 0;

/**
* Get memory usage
* \retval memory usage value
*/
virtual unsigned int get_mem_usage() const = 0;

/**
* Get network usage
* \retval network usage value
*/

```

```

*/
virtual unsigned int get_net_usage() const = 0;

/**
 * Get free memory
 * \retval free memory
 */
virtual unsigned int get_freeram() const = 0;

/**
 * Get temperature from board sensor
 * \retval temperature value
 */
virtual int get_system_temperature() const = 0;

/**
 * Get IP address from a given interface
 * \param iface Pointer to string with interface name
 * \param buff Pointer to buffer to store ip address in string format
 * \param size Available size of buffer to store ip address in string format
 * \retval 0 Operation was successfull
 * \retval -1 Operation could not be completed
 */
virtual int get_ipaddress(const char *iface, char *buff, int size) const = 0;

```

VideoAnalytics

```

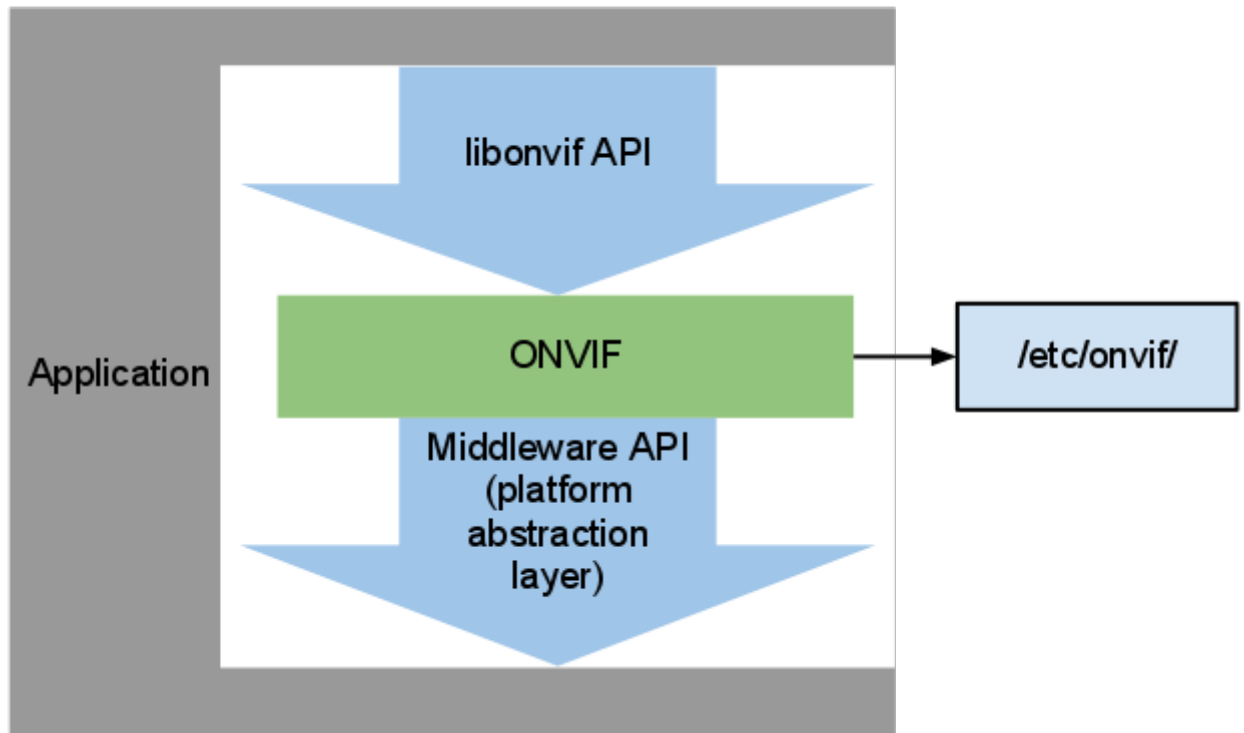
/**
 * Enable video analytics
 *
 * The implementation shall apply the video analytics configuration to the corresponding channel.
 * The implementation shall validate an incoming XML and return corresponding error code in the
 * error case.
 *
 * \param ch          Number of the channel, for which the configuration is being applied
 * \param xml         XML representation of the configuration
 * \param xml_size    Size of the XML representation
 *
 * \return IVIDEO_ANALYTICS_OK in the case of success, otherwise error code of the ivideo_analytics_error_code_t type
 */
virtual int enable_video_analytics(int ch, const char* xml, int xml_size) = 0;
/**
 * Disable video analytics
 *
 * The function shall disable video analytics on the corresponding channel.
 *
 * \param ch Number of the channel, for which the configuration is being applied
 *
 * \return IVIDEO_ANALYTICS_OK in the case of success, otherwise error code of the ivideo_analytics_error_code_t type
 */
virtual int disable_video_analytics(int ch) = 0;

```

ONVIF platform independent component

ONVIF platform independent component (libonvif) may be used not only as part of Network Video Transmitter Suite, but may be embedded to existing software suites, which do not support ONVIF interface currently.

It's API is divided onto two parts: libonvif API and middleware API



libonvif API

Interface exported by libonvif to user application. It contains functions for:

- Start and stop ONVIF interface
- Process HTTP requests from client
- Deliver events to ONVIF component
- Read serialized metadata (for sending it via RTSP)

API functions (see Doxygen documentation for full information):

```
/**
 * Initializes the ONVIF Framework. This function must be called prior to any other function
 * from the ONVIF Framework API.
 *
 * \param api A pointer to a structure of Middleware type.
 * \param activeConfiguration Token of active configuration. Will be used only when configuration
 * are loaded from default file.
 * Leave empty or NULL, if you don't use alternative device configurations
 * \return \b true if initialization was successful, \b false if not.
 */
bool InitializeOnvif(const MiddlewareApi* api, const char* activeConfiguration);

/**
 * Deinitializes the ONVIF Framework. No functions from the ONVIF Framework API can be called after
 * call of this function.
 *
 * \return No return value.
 */
void FinalizeOnvif();

/**
 * Executes a message loop. Inside the message loop the Framework takes a message from the message queue
 * and dispatches it to message handlers. There always a default message handler installed by the Framework.
 * Applications can install it's own message handlers. The function exits when \b OM_QUIT message is received.
 *
 * \return The exit code.
 */
int ExecuteMessageLoop();

/**
```

```

* Performs a single iteration of the message loop. This call can be used in synchronous functions in order
* to let the framework to process other messages while waiting for specific condition.
*
* \return The \b Type of the processed message.
*/
int ProcessSingleMessage();

/**
* Posts a message to the message queue.
*
* \param message A message that is to be added to the message queue.
* \return Operation status
* \retval 0 OK
* \retval -1 Event queue is full
*/
int PostMessage(const Message* message);

/**
* Posts the OM_QUIT message to the message queue, thus making ExecuteMessageLoop() to return with exit code 0.
* \return No return value.
*/
void PostQuitMessage();

/**
* Register a message handler for receiving messages. If the given message handler is already installed,
* the function does nothing.
*
* \param handler A pointer to a function that is called every time when a new message is taken from the message queue.
* \return No return value.
*/
void RegisterMessageHandler(MessageHandler handler);

/**
* Unregisters the previously registered message handler. If the given message handler is not installed,
* the function does nothing.
*
* \param handler A pointer to a function that is called every time when a new message is taken from the message queue.
* \return No return value.
*/
void UnregisterMessageHandler(MessageHandler handler);

/**
* Schedules periodic invocation of the \b timerProc every \b elapsed milliseconds
* \param elapsed period of time, in milliseconds
* \return Timer ID
*/
int SetTimer(int elapsed, TimerProc timerProc, int param);

/**
* Releases resources occupied by \b timerId timer.
*/
void KillTimer(int timerId);

/**
* Asynchronously processes ONVIF request from NVC. Incoming request is to be read by
* \b asyncRequest.Read callback and written back by \b asyncRequest.Write.
* The \b asyncRequest object must not be released until \b asyncRequest.Close is called.
*/
void ProcessOnvifRequestAsync(AsyncRequest* asyncRequest);

/**
* Serializes metadata.
* Implementation may have several RTSP streams simultaneously. The param \b streamName
* allows to separate metadata flows between them. Output buffer \b outputBuf is valid until
* the next call of \b SerializeMetadata.
*
* \param streamName Stream name returned by GetStreamName request.
*/
int SerializeMetadata(const char* streamName, const char* metadata, const char** outputBuf);

/**
* Schedules the event for sending it to subscribers.
* Note that text from topic and params are copied to XML as is without reserved symbols processing.
* Library can't check all text due performance reasons, so, it must be done by the client code.
* \param utcTime Time stamp for the event
* \param topic Event topic
* \param property Property operation
* \param params Event parameters. NULL if there are no parameters for the event
*/
void ScheduleEvent(time_t utcTime, const char* topic, PropertyOperation property,
const struct EventParams* params);

/**
* Simplified version of ScheduleEvent. Allows to schedule only events, which contain simple items
* Use this function for events, which does not contain Element items and does not contain spaces in the
* name and value
* Note that text from topic and params are copied to XML as is without reserved symbols processing.
* Library can't check all text due performance reasons, so, it must be done by the client code.
* check all text for performance reasons, do it
* \param utcTime Time stamp for the event
* \param topic Event topic
* \param property Property operation
* \param source Source items. Presented as space separated list of item names and values.

```

```

*           Example: "CPULoad 26% MemoryLoad 87%"
* \param key Key items. Presented as space separated list of item names and values.
* \param Data Data items. Presented as space separated list of item names and values.
*/
void ScheduleSimpleEvent(time_t utcTime, const char* topic, PropertyOperation property,
                        const char* source, const char* key, const char* data);

/**
 * Notifies the ONVIF Framework that network settings were changed from outside the Framework,
 * i.e. that from now the device is running with new network settings and it's required to reinit
 * components dependent on this.
 * Such change of network settings can occur as a result of DHCP lease time expiration.
 */
int NetworkChangeNotify();

```

Middleware API

Since libonvif is platform independent library, it requires abstraction layer, used to interact with the platform. User's application (platform) exports set of callbacks to libonvif. API contains functions for:

- Configure and start media streams
- Configure video sensor
- Read and modify network settings
- Read and modify system date and time
- Reset settings to factory defaults
- Reboot system
- Write and read logs
- Configure and trigger relays (if available)
- Send commands to PTZ cameras (if available)
- Configure video analytics (if available)

API functions (see Doxygen documentation for full information):

```

struct MiddlewareApi
{
    /**
     * Implementation shall return information about the host.
     * \param type Indicates what information is requested, see \b HostInformationType for details.
     * \param buffer A pointer to a buffer where implementation shall write a string containing requested info.
     * \param size Size of the \b buffer.
     * \return Implementation must return \b 0 if call was successfull and \b -1 if was not.
     */
    int (*GetHostInformation)(HostInformationType type, char* buffer, int bufferSize);

    /**
     * Update network settings
     * \param type Indicates what information must be updated, see \b HostInformationType for details.
     * \param buffer A pointer to a buffer where new value of the parameter is stored
     * \param size Size of the \b buffer.
     * \return Implementation must return \b 0 if call was successfull and \b -1 if was not.
     */
    int (*SetHostInformation)(HostInformationType type, const char* buffer);

    /**
     * Implementation shall return list of DNS servers used on device.
     * \param dnsList A pointer to buffer where DNS server list shall be written. DNS server addresses
     * must be delimited by a space.
     * \param searchDomainList A pointer to buffer where search domain list shall be written. Search domains
     * must be delimited by a space.
     * \param source Indicates if DNS settings were set manually or obtained by DHCP.
     * \return Implementation must return \b 0 if call was successfull and \b -1 if was not.
     */
    int (*GetDns)(char* dnsList, char* searchDomainList, SourceType* source);

    /**
     * Implementation shall set DNS servers on device.
     *
     * NOTE: ONVIF library calls this function only after settings has been changed by NVC.
     * Middleware should save this settings, apply it after system has been
     * restarted, include to system backup file.
     *
     * \param dnsList A pointer to the DNS server list. DNS server addresses are delimited by a space.
     * \param searchDomainList A pointer to the Search Domain list. Search domains are delimited by a space.
     * \param source Indicates if DNS settings shall be set manually or obtained by DHCP.
     * \return Implementation must return \b 0 if call was successfull and \b -1 if was not.
     */
    int (*SetDns)(const char* dnsList, const char* searchDomainList, SourceType source);

    /**
     * Implementation shall return list of NTP servers used on device.
     * \param ntpList A pointer to buffer where NTP server list shall be written. NTP server names or
     * addresses must be delimited by a space.
     * \param source Indicates if NTP settings were set manually or obtained by DHCP.
     * \return Implementation must return \b 0 if call was successfull and \b -1 if was not.
     */
}

```

```

int (*GetNtp) (char* ntpList, SourceType* source);

/**
 * Implementation shall set NTP servers on device.
 * \param ntpList A pointer to the NTP server list. NTP server addresses or names are delimited by a space.
 * \param source Indicates if NTP settings shall be set manually or obtained by DHCP.
 * \return Implementation must return \b 0 if call was successfull and \b -1 if was not.
 */
int (*SetNtp) (const char* ntpList, SourceType source);

/**
 * Implementation must return network settings for the given network interface.
 *
 * \param interface The network interface name. For example, "eth0".
 * \param macAddress If not \b NULL this arguments point to a buffer where implementation
 * shall write MAC address.
 * \param ipAddress If not \b NULL this arguments points to a buffer where implementation
 * shall write IP v4 address in digits-and-dots notation.
 * \param netMaskPrefix If not \b NULL this arguments points to a buffer where implementation
 * shall write length of netmask prefix.
 * \param dhcpEnabled If not \b NULL this arguments points to a buffer where implementation
 * shall write a flag indicating if DHCP enabled or not.
 * \param zeroEnabled If not \b NULL this arguments points to a buffer where implementation
 * shall write a flag indicating if zero configuration enabled or not.
 * \return Implementation must return \b 0, if success \b -1 in case of any error.
 */
int (*GetNetworkInterfaces)(NetworkInterface** interfaces, int* count);

/**
 * Modifies configuration files. Actual network settings change are to be applied during
 * NetworkReset call. Such two-stage setup is used because we must send response to
 * the SetNetworkInterface ONVIF call prior to resetting the network interface.
 *
 * NOTE: ONVIF library calls this function only after settings has been changed by NVC.
 * Middleware should save this settings, apply it after system has been
 * restarted, include to system backup file.
 *
 * \param interface The network interface name. For example, "eth0".
 * \param ipAddress The IP v4 address in digits-and-dots notation.
 * \param netMaskPrefix Length of netmask prefix.
 * \param dhcpEnabled A flag indicating if DHCP enabled (\b 1) or not (\b 0).
 * This flag can equal \b -1. It means that parameter shall not be set.
 * \param zeroEnabled A flag indicating if zero configuration enabled (\b 1) or not (\b 0).
 * This flag can equal \b -1. It means that parameter shall not be set.
 * \return Implementation must return \b 0 if call was successfull and \b -1 if was not.
 */
int (*SetNetworkInterface)(NetworkInterface* interface);

/**
 * Configure ports for web server and media streamer
 * \param protocols Lists of ports for every protocol
 */
int (*SetNetworkProtocols) (NetworkProtocols* protocols);

/**
 * Implementation must return current date and time settings.
 * \param datetime A pointer to \b DateTime structure where settings shall be returned.
 * \return Implementation must return \b 0 if call was successfull and \b -1 if was not.
 */
int (*GetDateTime) (DateTime* datetime);

/**
 * Implementation shall apply new date and time settings.
 * \param datetime A pointer to \b DateTime structure with new settings.
 * \return Implementation must return \b 0 if call was successfull and \b -1 if was not.
 */
int (*SetDateTime) (DateTime* datetime);

/**
 * Implementation shall reboot the device.
 * \param delay A time in seconds for delaying before reboot.
 * \return Implementation must return \b 0 if call was successfull and \b -1 if was not.
 */
int (*Reboot) (int delay);

/**
 * Implementation shall manage discovery module.
 * \param action Actions: 0 - stop, 1 - start, 2 - restart.
 * \param params A pointer to a buffer with additional command line params.
 * \return Implementation must return \b 0 if call was successfull and \b -1 if was not.
 */
int (*Discovery) (int action, const char *params);

/**
 * Implementation must provide network reset in order to apply changes
 * made by SetNetworkInterface or by DHCP.
 * \return Implementation must return \b 0 if call was successfull and \b -1 if was not.
 */
int (*NetworkReset) ();

/**
 * Implementation shall reset settings to factory defaults according to ONVIF specification.
 * \param type Indaicates what kind of factory defaults is requested: \b soft or \b hard.

```

```

    * \return Implementation must return \b 0 if call was successfull and \b -1 if was not.
    */
int (*SetFactoryDefault) (FactoryDefaultType type);

/**
 * Implementation shall return information about device.
 * \param type Indicates what information is requested, see \b DeviceInformationType for details.
 * \param buffer A pointer to a buffer where implementation shall write a string containing requested info.
 * \param size Size of the \b buffer.
 * \return Implementation must return \b 0 if call was successfull and \b -1 if was not.
 */
int (*GetDeviceInformation) (int type, char* buffer, int size);

/** \brief Get a URI which can be used for upload firmware
 * The URI should be valid indefinitely.
 *
 * \param buffer Buffer for result string
 * \param size Size of result buffer
 * \return Operation status
 * \retval 0 OK
 * \retval -1 Failed
 */
int (*GetFirmwareUploadUri) (char *buffer, int size);

/**
 * Implementation must return a flag indicating if the \b videoEncoder is compatible with
 * the given \b videoSourceToken
 * \param videoSourceToken An ONVIF token of video source.
 * \param inWidth Input (from a sensor or other video source) video width.
 * \param inHeight Input (from a sensor or other video source) video height.
 * \param outWidth Output (via RTSP stream) video width
 * \param outHeight Output (via RTSP stream) video height
 * \param videoEncoder Type of video encoder.
 */
int (*ValidateVideoEncoder) (const char* videoSourceToken, int inWidth, int inHeight,
                             int outWidth, int outHeight, VideoEncodingType videoEncoder);

/**
 * \param videoSourceToken An ONVIF token of video source.
 * \param settings SensorSettings structure.
 */
int (*SetSensorSettings) (const char* videoSourceToken, SensorSettings* settings);

/**
 * Start stream and returns its name.
 * Name is the last part of URL. It will be used to generate URI in libonvif
 * If RTSP over HTTP is used, middleware must additionally specify HTTP port of the streamer
 * Examples:
 * 1) Link "rtsp://192.168.0.1:5554/streams/media0". Stream name is "streams/media0". *httpPort is ignored
 * 2) Link "http://192.168.0.1:8080/h264_playback". Stream name is "h264_playback". *httpPort is 8080
 * \param streamSetup Stream settings.
 * \param nameBuffer Return value. buffer of 2048 bytes length for resulting stream URI.
 * \return Operation status
 * \retval 0 OK
 * \retval -1 Failed
 */
int (*StartStream) (StreamSetup* streamSetup, char* nameBuffer);

/**
 * The implementation shall stop streaming for the given profile.
 */
int (*StopStream) (const char* profileToken);

/**
 * Get a URI which can be used for download JPEG snapshot
 * The URI should be valid indefinitely.
 * Usually, returned URI is not an URI of snapshot file, but a URI of CGI application or webserver,
 * which takes snapshot, when it is requested.
 *
 * \param videoSourceToken Token of video source to be used
 * \param bounds Bounds or area, visible on screenshot
 * \param uriBuffer Buffer for result string
 * \param bufferSize Size of result buffer
 * \return Operation status
 * \retval 0 OK
 * \retval -1 Failed
 */
int (*GetSnapshotUri) (const char* videoSourceToken, VideoBounds* bounds,
                       char *uriBuffer, int bufferSize);

/**
 * Insert I-frame to all video stream(s), wich have given source and encoding
 *
 * \param videoSourceToken Token of video source, used by video stream(s)
 * \param videoEncoding Video encoding of video stream(s)
 * \return Operation status
 * \retval 0 OK
 * \retval -1 Failed
 */
int (*SetSyncPoint) (const char* videoSourceToken, VideoEncodingType videoEncoding);

/**
 * Get guaranteed number of video encoder instances.
 *
 * See "GetGuaranteedNumberOfVideoEncoderInstances" description in ONVIF standard
 */

```

```

* \param videoSourceToken Token of video source, for which number of video encoders requested
* \param bounds Bounds region, for which encoders number requested
* \param totalNumber Total number of supported video encoders
* \param jpeg Number of JPEG encoder instances, -1 if not set
* \param h264 Number of H264 encoder instances, -1 if not set
* \param mpeg4 Number of MPEG4 encoder instances, -1 if not set
**/
int (*GetGuaranteedNumberOfVideoEncoderInstances) (const char* videoSourceToken, VideoBounds* bounds,
int* totalNumber, int* jpeg, int* h264, int* mpeg4);

/**
* Get PTZ node status
*
* \param node Address of PTZ node
* \param position Buffer to return PTZ node current position
* \param panTiltMoveStatus Buffer to return PTZ node PanTilt move status
* \param zoomMoveStatus Buffer to return PTZ node Zoom move status
* \param error Buffer to return error messages from PTZ node
* \param size Size of error buffer
* \return Operation status
* \retval 0 OK
* \retval -1 Failed
*/
int (*PTZGetStatus) (int node, Vector3D* position,
MoveStatus* panTiltMoveStatus, MoveStatus* zoomMoveStatus,
char* error, int size);

/**
* Send PTZ move command
*
* \details Some parameters (except node and moveType) can be required or optional (can be set as 0 or NULL) depending
on movement type.
* Here is a brief list of required parameters depending on movement type (moveType parameter):
* ABSOLUTE: position, speed, limits;
* RELATIVE: position, speed, limits;
* CONTINUOUS: speed, limits, timeoutMs;
* PRESET: preset, speed;
* STOP: speed;
*
* \param node Address of PTZ node
* \param moveType Movement type
* \param preset Preset number to move to. The value must be provided if moveType is PRESET.
* Special value 0 is used to move to Home (Zero) position
* \param position Position coordinates to move to.
* Must be provided if moveType is ABSOLUTE or RELATIVE (position coordinates represent absolute and relative
position accordingly)
* otherwise can be set as NULL.
* \param speed Speed values to use while moving. If moveType is STOP speed vector values has special meaning.
* Use x = 0 to stop Pan movement, y = 0 to stop Tilt movement, z = 0 to stop Zoom movement
* \param limits Space limits that must be followed while moving PTZ node.
* Must be provided if moveType is ABSOLUTE/RELATIVE/CONTINUOUS otherwise can be set as NULL.
* \param timeout Timeout in milliseconds to stop when performing continuous movement (moveType = CONTINUOUS))
* \return Operation status
* \retval 0 OK
* \retval -1 Failed
*/
int (*PTZMove) (int node, MoveType moveType, int preset,
Vector3D* position, Vector3D* speed,
Space3D* limits, int timeout);

/**
* Set PTZ node preset
*
* \param node Address of PTZ node
* \param preset Preset number to set/overwrite. Special value 0 is used to set Home (Zero) position
* \param position Buffer to return PTZ node saved preset position
* \return Operation status
* \retval 0 OK
* \retval -1 Failed
*/
int (*PTZSetPreset) (int node, int preset, Vector3D* position);

/**
* Send auxiliary command to PTZ node
*
* \param node Address of PTZ node
* \param auxCmd Auxiliary command to send
* \param response Buffer to return PTZ node response
* \param size Size of response buffer
* \return Operation status
* \retval 0 OK
* \retval -1 Failed
*/
int (*PTZSendAuxiliaryCommand) (int node, const char* auxCmd,
char* response, int size);

/**
* Implementation must provide printing of debug messages.
*
* \param level Defines the debug level of the message. Implementation may choose when to display messages
of the given debug level.
* \param message A debug message.
* \return No return value.
*/
void (*Log) (DebugLevel level, const char* message);

/**
* Get system logs for return it to the user.

```

```

* If no logs are available, function must return 0 and SystemLog.data must be set to NULL
*
* \param type Requested log type
* \param log Output parameter. Must be filled by middleware with valid values.
* \param releaseFunc Release function. Will be called by the library when it is possible to free system resources.
* \return \b 0 if call was successfull and \b -1 if was not.
*/
int (*GetSystemLog) (SystemLogType type, SystemLog* log, PostActionFuncType* releaseFunc);

/**
* Implementation shall performs backup of system settings, pack backup files together (if backup consists of
* several files) and copy the path of this file info a buffer pointed by backupFilePath.
* Implementation shall not delete this file - it is deleted by the ONVIF framework.
*
* \param backupFilePath A pointer to the buffer where implementation shall copy backup file path.
* The buffer size is 1024.
* \return \b 0 if call was successfull and \b -1 if was not.
*/
int (*Backup) (char* backupFilePath);

/**
* Implementation shall perform restore of system settings. After successful restore system will be restarted
* by invoking Reboot callback.
*
* \param restoreFile A pointer to binary data containing restore file.
* \param size Restore file size.
* \return \b 0 if call was successfull and \b -1 if was not.
*/
int (*Restore) (void* restoreFile, int size);

/**
* Implementation shall perform update of user settings.
*
* \param users A array of users, passwords add access levels.
* \param size size of array.
* \return \b 0 if call was successfull and \b -1 if was not.
*/
int (*UpdateUsers) (UserInfo *users, int size);

/**
* Set relay output settings
*
* \param index relay output index to set settings for
* \param mode operation mode
* \param idleState idle (inactive) state
* \param timeout release timeout for monostable relay in ms
* \retval 0 OK
* \retval -1 Failed
*/
int (*SetRelayOutputSettings)(int index, RelayMode mode, RelayIdleState idleState, int timeout);

/**
* Set relay output state
*
* \param index relay output index to set settings for
* \param state state to set
* \retval 0 OK
* \retval -1 Failed
*/
int (*SetRelayOutputState)(int index, RelayState state);

/**
* Enable video analytics
*
* The implementation shall apply the video analytics configuration to the corresponding video source.
* The implementation shall validate an incoming XML and return corresponding error code in the
* error case.
*
* \param videoSourceToken Token of video source, for which the configuration is being applied
* \param xml XML representation of the configuration
* \param xmlSize Size of the XML representation
*
* \return VIDEO_ANALYTICS_OK in the case of success, otherwise error code of the VideoAnalyticsErrorCode type
*/
int (*EnableVideoAnalytics)(const char* videoSourceToken, const char* xml, int xmlSize);

/**
* Disable video analytics
*
* The function shall disable video analytics on the corresponding video source.
*
* \param videoSourceToken Token of video source, for which the configuration is being applied
*
* \return VIDEO_ANALYTICS_OK in the case of success, otherwise error code of the VideoAnalyticsErrorCode type
*/
int (*DisableVideoAnalytics)(const char* videoSourceToken);
};

```

Third-party dependencies

MontaVista products

linux	2.6.18_pro500	GPL 2
-------	---------------	-------

Texas Instruments products

ubl	1.3.4	
bios	5.33.06	TI license
codec_engine	2.24.01	
dsplib	2.10	TI license
dsplink_linux	1.63	GPL 2
framework_components	2.24.01	
imglib	2.0.1	TI license
xdais	6.24	
xdctools	3.15.00.50	

Other products

alsa-lib	1.0.21a	LGPL 2.1
alsa-utils	1.0.22	GPL 2
busybox	1.16.2	GPL 2
gnupg	1.4.9	GPL 3
gsoap	2.7.13	GPL 2
libc	2.5.90	LGPL 2.1
liboconfig	1.4.3	LGPL 2.1
libpng	1.2.29	libpng/zlib
libxml2	2.6.32	MIT
linuxutils	2.24.02	GPL 2
live555	2010.03.16	LGPL 2.1
lzo2	2.03	GPL 2
mini_httpd	20080508	GPL2
ntp	4.2.6	NTP License
openssh	0.9.8i	OpenSSL and SSLeay
thttpd	2.25b	BSD
u-boot	1.3.4	GPL 2
udev	070	GPL 2
ws4d-gsoap	0.7	GPL 2
zlib	1.2.3	libpng/zlib