Ivideon Embedded SDK Handbook

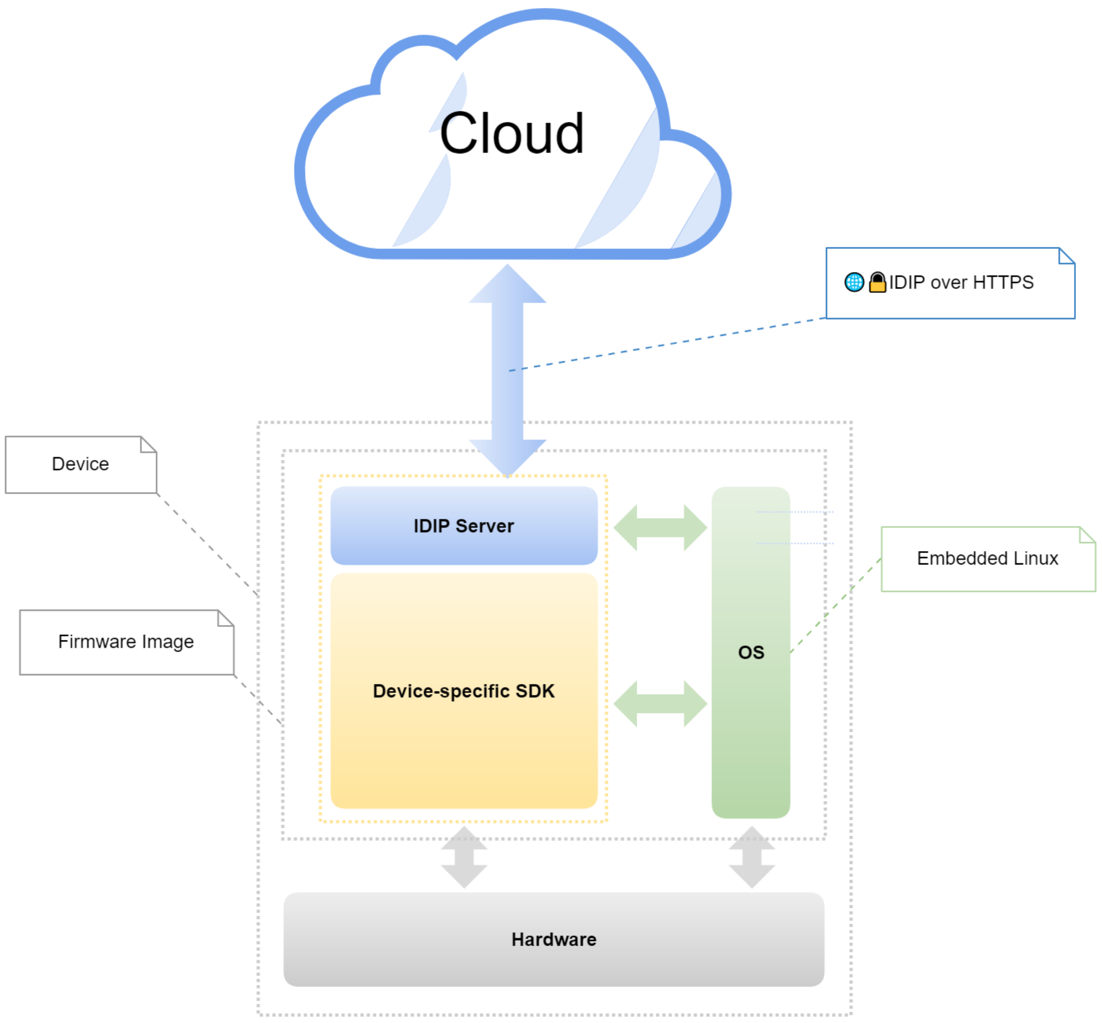
Contents

1 Ivideon Embedded SDK

The Ivideon Embedded Software Development Kit (IVESDK) is a set of libraries and tools that enables device intergration into Ivideon services.

The main goal of the IVESDK is to create an abstraction layer over the camera/DVR hardware, that can interact with the Ivideon Cloud by using an Ivideon Data Integration Protocol (IDIP). In this guide we will refer to this layer as 'IDIP server' or 'application' and to the camera/DVR as 'device'. The IDIP server allows Ivideon software to utilize device resources through IDIP calls and can be implemented as a standalone embedded application or be integrated to the device firmware.

1.1 Overview Diagram



In IVESDK 2.0 we introduce an agentless scheme. The scheme removes Ivideon Agent from the device firmware and allow IDIP server to communicate with the Cloud directly.

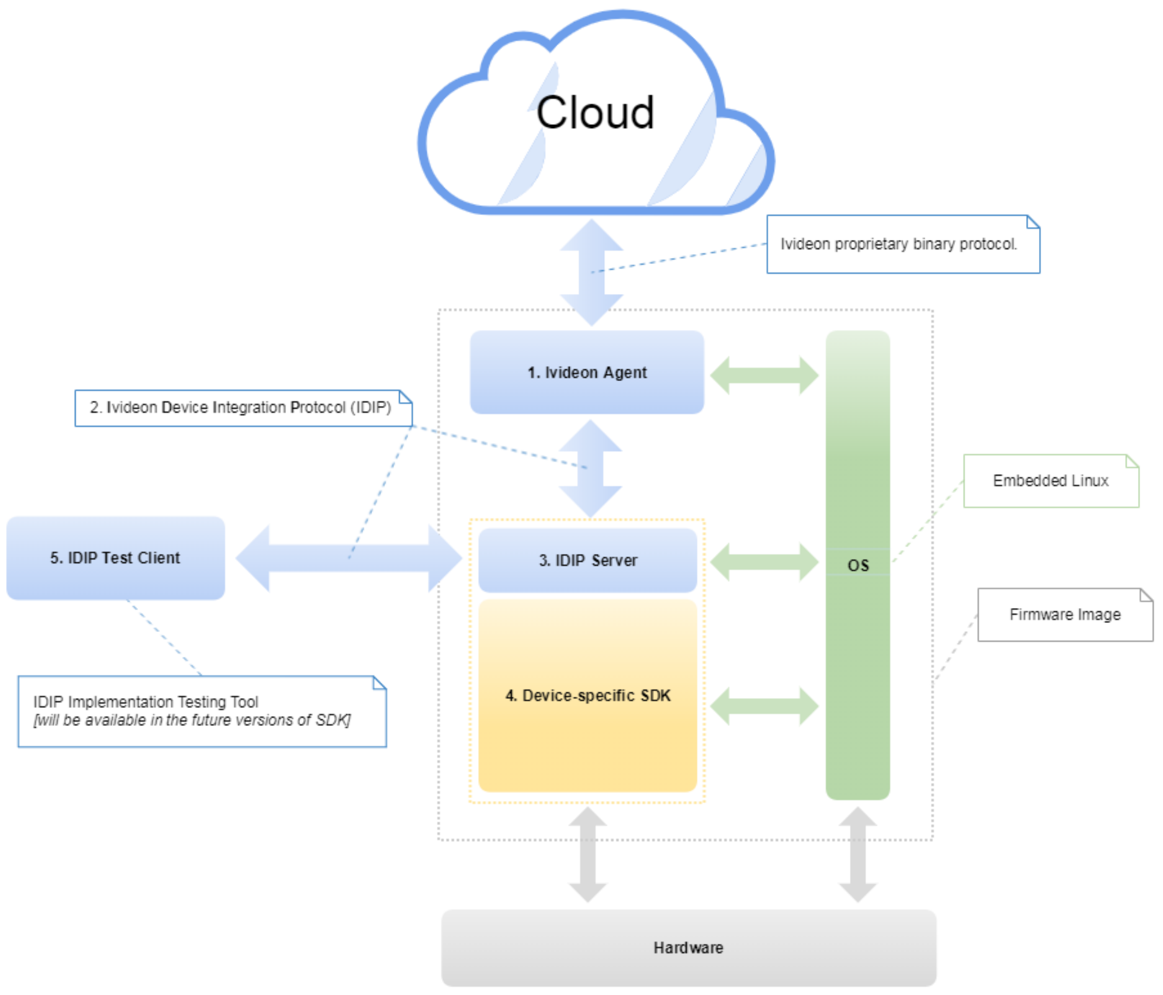
1. Ivideon Cloud is a main Ivideon service infrastructure. It implements accounting, data store and other buisness logic.
2. Ivideon Device Integration Protocol (IDIPv2, IDIP) – a message-oriented protocol, that simplifies communication between the Cloud and the device firmware. For security reasoons the IDIPv2 protocol uses TLS-encrypted connections (HTTPS).

3. IDIP server is a compact C library that implements IDIP protocol and provides an easy-to-use interface (functions and data structures) to protocol capabilities. It is distributed by Ivideon in source code along with demo examples and documentation. IDIP server is intended to be used by developers to integrate device-specific features into Ivideon ecosystem: video and audio live streaming, event triggering, edge video archive playback, etc…
4. Low-level device-specific SDK that is usually provided by a manufacturer or SoC developer.
5. IDIP Test Client is a desktop application with GUI that connects to IDIP Server and tests each particular feature of IDIP protocol implemented by IDIP server. It can used to debug and test their IDIP server implementations, by making sure that all needed features of IDIP protocol are implemented correctly.

The most obvious benefits of agentless scheme are:

- Reduced FLASH and RAM capacity requirements → ability to integrate cheaper devices.
- New IDIP 2.0 protocol allows only one client connection → zero network configuration, no port forwarding.
- No need to build Ivideon Agent for each integration request → faster and more scalable integration process.
- It’s much easier to debug and maintain Cloud components rather than embedded ones → faster debugging and bug fixing.
- In some cases it’s possible (and reasonable) to implement part of the logic on the Cloud side → faster feature development without FW upgrade.
- It will be relatively easy to migrate from classic integration (Agent + IDIP server) to the agentless scheme with minimal changes in IDIP servers.

Also you can choose the “agentfull” integration technique. It requires advanced steps to inegrate an Ivideon Agent on the camera, advanced configuration steps and so on. In most cases you prefer to choose the agentless scheme.


Cloud
Ivideon proprietary binary protocol.
1. Ivideon Agent
2. Ivideon Device Integration Protocol (IDIP)
Embedded Linux
5. IDIP Test Client
3. IDIP Server
OS
Firmware Image
4. Device-specific SDK
IDIP Implementation Testing Tool
[will be available in the future versions of SDK]
Hardware

1. Ivideon Agent is a service that transfers data and control commands between the Cloud and IDIP server.
2. Ivideon Device Integration Protocol (IDIP) – a message-oriented protocol, that simplifies communication between the Agent and the device firmware.
3. IDIP server is a compact C library that implements IDIP protocol and provides an easy-to-use interface (functions and data structures) to protocol capabilities. It is distributed by Ivideon in source code along with demo examples and documentation. IDIP server is intended to be used by developers to integrate device-specific features into Ivideon ecosystem: video and audio live streaming, event triggering, edge video archive playback, etc…
4. Low-level device-specific SDK that is usually provided by a manufacturer or SoC developer.
5. IDIP Test Client is a desktop application with GUI that connects to IDIP Server and tests each particular feature of IDIP protocol implemented by IDIP server. It can used to debug and test their IDIP server implementations, by making sure that all needed features of IDIP protocol are implemented correctly.

1.2 Device integration

A company that is going to integrate a device (e.g. IP camera) with Ivideon Cloud (or some Private Cloud derived from it):

1. Sends to Ivideon a toolchain for cross-compiling native apps for hardware architecture of the

target device (**Don't need for the agentless scheme**);
2. Receives from Ivideon Embedded SDK package containing the following components:

    a. The Ivideon Agent executable binary file (cross-compiled by the toolchain from the step 1, **Don't need for the agentless scheme**.);
    b. IDIP server library source code;
    c. Documentation and demo examples;
    d. An Ivideon engineer for operative support and troubleshooting;

3. Rceives a **Device Key** and **Cloud Endpoint URL** from Ivideon. The Device Key (or **DevKey**) is a string token that identifies a solution for the Ivideon Cloud;
4. Integrates all device capabilities into the Ivideon ecosystem using the library and device-specific SDK;
5. Embeds the Ivideon Agent and IDIP server into the FW image (**Don't need for the agentless scheme**);
6. Tests the solution.

1.3 Get an Ivideon Embedded SDK

To get the IVESDK you can download a tarball with a repository snapshot. Installation details are described in Installation chapter.

1.4 Requirements

All steps bellow assumes that they will run on a recent Debian-based Linux. We recommend Ubuntu 18.04, Linux Mint 19 or above. Following packages are required:

- `make`
- `cmake` (version 3.10 or above)
- `gcc` (version 4.7 or above, C99 support, 32-bit atomics native support or with libatomic)
- (optional) cross-compilation toolchain based on `gcc` or `llvm`.
- (optional) ffmpeg libraries (Version 3.4 or above) built for the target platform and header files. It requires to build our demo.
- (optional) libasound libraries and its header files build for the target platform. It requires for push-to-talk feature in our demo.

If GUI tools are preferrable, then you may choose to use `cmake-gui` tool instead `cmake`.

1.5 Target requirements

- 32 or 64 bit instruction set CPU.
- Linux OS.
- POSIX.1-2001 threads (or above).
- all requirements from libwebsockets and mbedtls libraries.
- 3-4 Mb of free memory for agentless integration, or 13-15 Mb for intergration with a Ivideon agent (videoserverd).
- 1 Mb/4 Mb of disk space (agentless/with Ivideon Agent).

1.6 Source tree

- `cmake` - Miscellaneous cmake stuff
- `doc` - documentation
- `examples` - libidip usage examples
  - `demo-server` - full-featured IDIP server demo
  - `skel` - minimalistic IDIP server skeleton
- `libidip` - library sources
- `thirdparty` - SDK dependencies
- `CMakeLists.txt` - SDK project file
- `version.txt` - version file

2 Installation

2.1 Dependencies

The IVESDK depends on these open-source libraries:

- `libwebsockets` https://libwebsockets.org/repo/libwebsockets, https://libwebsockets.org/
- `msgpack-c` https://github.com/msgpack/msgpack-c.git
- `mbedtls` https://tls.mbed.org/, https://github.com/ARMmbed/mbedtls

The directory `thirdparty` contains `libwebsockets`, `mbedtls` and `msgpack-c` sources. Just unpack tarball into any folder which is convenient to you, e.g.:

```
tar xvzf ivideon-embedded-sdk-vA.B.C.YYYYMMDD.hhhhh.tar.gz
```

where **A**, **B** and **C** are version numbers, **YYYYMMDD** is a archive build date, **hhhhh** is a commit hash number. Unpacked files are in `ivideon-embedded-sdk-vA.B.C.YYYYMMDD.hhhhh` directory.

The IVESDK `demo` also requires `ffmpeg` and `ALSA` development libraries:

```
sudo apt install libavcodec-dev libavdevice-dev libavfilter-dev \
    libavresample-dev libavutil-dev libswscale-dev libswresample-dev \
    libasound2-dev
```

2.2 Quick Installation on PC

1. Install required build tools and dependencies:

   ```
   # required packages
   sudo apt install cmake make gcc binutils
   # optional. If you want to build demo example
   sudo apt install libavcodec-dev libavdevice-dev \
       libavfilter-dev libavresample-dev libavutil-dev \
       libswscale-dev libswresample-dev libasound2-dev
   ```

2. Unpack the `ivideon-embedded-sdk-vA.B.C.YYYYMMDD.hhhhh.tar.gz` tarball. We assume that tarball is downloaded to user home directory.

```
tar xvzf $HOME/ivideon-embedded-sdk-vA.B.C.YYYYMMDD.hhhhh.tar.gz
# unpacked files are in $HOME/ivideon-embedded-sdk-vA.B.C.
    YYYYMMDD.hhhhh directory
```

3. (optional) If cross-compilation is required, install a target toolchain and create a toolchain file for it. We assume that toolchain file placed at `$HOME` directory and named `custom-toolchain.cmake`. For example, a Raspberry Pi toolchain file:

   ```
   # raspberry-pi-toolchain.cmake
   set(CMAKE_SYSTEM_NAME Linux)
   set(CMAKE_SYSTEM_PROCESSOR ARM)
   set(CMAKE_C_COMPILER /path/to/toolchain/bin/armv8-rpi3-linux-
       gnueabihf-gcc)
   set(CMAKE_CXX_COMPILER /path/to/toolchain/bin/armv8-rpi3-linux-
       gnueabihf-g++)
   ```

   where `/path/to/toolchain` must be replaced with the actual toolchain root directory.

4. Build and install SDK:

   ```
   cd $HOME/ivideon-embedded-sdk-vA.B.C.YYYYMMDD.hhhhh
   mkdir -p build
   cd build
   cmake -DCMAKE_INSTALL_PREFIX=$HOME/ivideon-embedded-sdk \
         -DCMAKE_TOOLCHAIN_FILE=$HOME/custom-toolchain.cmake \
         -DCMAKE_BUILD_TYPE=MinSizeRel ..
   make
   make install
   ```

   Note: omit the `CMAKE_TOOLCHAIN_FILE` argument if you compile with the host toolchain.

5. The Ivideon Embedded SDK is installed into `$HOME/ivideon-embedded-sdk` directory.

2.3 Advanced options

The IVESDK is a cmake-based project. It supports a set of additional build options. By default only `libidip` will be built and installed. `libidip` is the main part of the SDK that contains implementation of the Ivideon Device Integration Protocol (IDIP). To build additional parts of the IVESDK you could pass the following options to cmake (`cmake -D`):

- `IVESDK_BUILD_EXAMPLES` - Set `ON` to build examples (`idip-demo-server`). `OFF` by default.
- `IVESDK_INSTALL_THIRDPARTY` - Set `ON` to install third-party libraries and includes with idip library or set the value to `OFF` to disable this behavior. `ON` by default.
- `BUILD_SHARED_LIBS` - Set `ON` to build dynamically linked library (`libidip.so`) instead statically (`libidip.a`). `OFF` by default.
- `CMAKE_BUILD_TYPE` - Set a build type for all parts of IVESDK. `Debug` by default. Possible values are: `Debug`, `Release`, `MinSizeRel`.
- `CMAKE_INSTALL_PREFIX` - A base path where header files and binaries will be installed. By default is `/usr/local`.
- `CMAKE_TOOLCHAIN_FILE` - A path to a file with cmake syntax that describes advanced toolchain variables, paths to cross-compiler, etc. Refer to cmake site for detailed description and examples.

2.4 Cross-compilation

Get and install a target toolchain and create a toolchain file for cmake. Examples:

1. A toolchain file for the Raspberry Pi target might look like this:

```
# raspberry-pi.cmake
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR ARM)
set(CMAKE_C_COMPILER /path/to/toolchain/bin/armv8-rpi3-linux-
    gnueabihf-gcc)
set(CMAKE_CXX_COMPILER /path/to/toolchain/bin/armv8-rpi3-linux-
    gnueabihf-g++)
set(CMAKE_SYSROOT /path/to/toolchain/armv8-rpi3-linux-gnueabihf/
    sysroot)
```

2. `i386` target on the `x86_64` host:

```
# i386.cmake
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR i686)
set(CMAKE_C_COMPILER gcc)
set(CMAKE_CXX_COMPILER g++)
set(CMAKE_C_FLAGS_INIT -m32)
set(CMAKE_CXX_FLAGS_INIT -m32)
set(CMAKE_EXE_LINKER_FLAGS_INIT -m32)
set(CMAKE_SHARED_LINKER_FLAGS_INIT -m32)
set(CMAKE_MODULE_LINKER_FLAGS_INIT -m32)
```

The name of the toolchain file is passed to `cmake` through a special cache variable `CMAKE_TOOLCHAIN_FILE`, like so:

```
cd /path/to/ivideon-embedded-sdk-vA.B.C.YYYYMMDD.hhhhh
mkdir -p build
cd build
cmake -DCMAKE_TOOLCHAIN_FILE=/path/to/raspberry-pi.cmake ..
```

3 API Documentation

API Documentation can be found in doc/libidip[1] directory.

4 Ivideon Embedded SDK User's Guide

4.1 Concepts

The main part of the Ivideon Embedded SDK is the `libidip` library, which allows to manage **IDIP server** instances. The following concepts are using in the libidip API:

- **IDIP server instance** is an entity that implies the IDIPv2 protocol and serves cloud-to-device and device-to-cloud interaction.

[1] libidip/idip_8h.html

- **A feature handler** is a callback type that implies device-dependent logic for a feature. Device integrators have to implement these handlers and register them on the IDIP server instance. These handlers can pass some input data from the cloud and can send data to the cloud.
- **A target name** is a string that corresponds to a camera or a server subsystem. Most handlers accept a target name in its arguments. `libidip` API provides a default target with an empty name (or `NULL`). If only the default target is used then handler must check the `target` argument. If handler is bound to some target this check is unnecessary since `libidip` checks it before the handler execution.
- **A profile name** is a string that means a video substream in general. Typical profile names are "0"/"1"/"2". They are matching to the main stream (with the higest quality), and two substreams ("medium" and "low").
- **A video source** is a helper subsystem which offers a simple way to stream the live video. The application code creates one or more instances of `idip_av_source_t` with `idip_server_av_source_register()` call and uses them to put live video and live audio data whenever it's ready. Note that one target can support several profile names and `idip_server_av_source_register()` must be called for each target-profile pair. See an `idip_av_source_t` API for details. You can change any source parameters at any time (excluding the target name and profile name).

Feature handlers have to be implemented by device developers. Each feature handler has a corresponding type (see `idip.h`) and handles a specific IDIP event.

Warning Handlers and the application main loop are executed in different thread contexts.

Each handler returns an `idip_status_code_t` value. If a handler implies data sending then `libidip` provides a `idip_stream_xxx_t` pointer as the first argument – an output stream. Most of the handlers have an associated `_xxx_` stream type and `_xxx_put_` function to put data into the stream during the event handling. For example, the `idip_stream_archive_handler_t` handler is called with an instance of an `idip_stream_archive_t` stream and following functions might be used to put data into the stream:

- `idip_stream_ar_put_video_info(idip_stream_archive_t* stream, ...)` - to put a video stream description into the output stream (send it to Ivideon software)
- `idip_stream_ar_put_audio_info(idip_stream_archive_t* stream, ...)` - to put an audio stream description into the output stream
- `idip_stream_ar_put_video(idip_stream_archive_t* stream, ...)` - to put video data into the output stream
- `idip_stream_ar_put_audio(idip_stream_archive_t* stream, ...)` - to put audio data into the output stream

For other handlers and related stream functions same naming rule applies.

Pay attention: do NOT use a pointer to an `idip_stream_xxx_t` instance outside the handler - it might be changed between two consecutive calls!

All `idip_stream_xxx_put_yyy()` functions are blocking and accept a timeout as an argument (or the `IDIP_INFINITE_TIMEOUT` value for an infinite timeout).

libidip convention:
A handler must return immediately if the `idip_stream_xxx_put_yyy()` function returned a value that differs from `IDIP_OK`.

4.2 Handler overview

Before starting an `idip_server_t` instance you should register handlers. Use `idip_server_setup_xxx_feature()` functions to do this. Each handler should be registered for each target separately. Note, that you can specify a pointer to some user data for these functions. The pointer will be passed to the corresponding handler.

There is a short handler overview:

- `idip_stream_av_handler_t`. Send live video and audio data from the device. You don't have to implement this callback in most cases, it's low level and has a useful adaptor `idip_av_source_t` (See the Video streaming section).
- `idip_stream_snapshot_handler_t`. Send snapshots. It acts like a previous one, except it sends a JPEG image or a small sequence of video data.
- `idip_audio_playback_handler_t`. Play audio using an onboard speaker system. An implied usage is a push-to-talk feature. See notes about how it could be done.
- `idip_stream_archive_handler_t`. Similarly to the `idip_stream_av_handler_t`, except video and audio data is read from an onboard archive storage. Although we don't force the implementation, it is recommended to consider the storage specifications.
- `idip_stream_archive_list_handler_t`. Get the list of archives.
- `idip_stream_archive_days_handler_t`. Get the list of first-of-day records.
- `idip_stream_config_get_handler_t`. Get device dynamic configuration values.
- `idip_stream_config_update_handler_t`. Update dynamic configuration values. See the configuration API description below.
- `idip_stream_persistent_data_get_handler_t`. Get data from a device persistent storage. **The handler should be always implemented.**
- `idip_stream_persistent_data_set_handler_t`. Store data to the device persistent storage. **The handler should be always implemented.**
- `idip_wireless_scan_handler_t`. Locate wireless networks (if an onboard wireless interface is present).
- `idip_wireless_setup_handler_t`. Connect to a specified wireless network.
- `idip_ptz_handler_t`. Perform a PTZ action.
- `idip_stream_system_notify_handler_t`. Notify the device about system events (like connection to the Ivideon cloud) using device indicators (onboard leds, screens, etc.).
- `idip_stream_update_version_handler_t`. Notify the device about an available software update. It accepts a new software version string and a firmware fetching URL. The implementation should download the firmware (by the URL) and apply it.
- `idip_stream_ota_handler_t`. Allows the device to fetch new firmware. It is an alternative way to get new firmware. The feature isn't fully implemented yet.
- `idip_stream_passwd_handler_t`. This handler allows to change password for a system account on a camera/DVR from the cloud.

- `idip_stream_system_get_users_handler_t`. The handler should return a list of system users on a camera/DVR to the cloud.
- `idip_system_get_interfaces_handler_t`. The handler allows the Cloud Agent to know about network interfaces on the device. **The handler should be always implemented.**
- `idip_system_token_handler_t`. This handler type allows to pass to libidip a **cloud attaching token** from QR code parser. The libidip will call the handler when it need for a token.

All handlers following the following conventions:

- They return `IDIP_OK` when the callback has success done or the handler was interrupted.
- They return `IDIP_INVALID_ARGUMENTS` when function arguments are invalid.
- They return `IDIP_GENERIC_ERROR` when the function implementation internal logic error occured. In other words, a callback returns the value in some other cases.
- They are detecting an interrupt request on `idip_stream_xxx_put()` methods. When the method didn't return `IDIP_OK` value, the callback should immediately returns.

Each handler accepts an `udata` pointer that points to a some user data. This pointer is saving on handler registration and then the `libidip` will pass the pointer to the handler call. For example, we define here a simple structure `my_user_data`. It contains a pointer to a path string which contains a path to file to play. Production code should contain a data structure like this to pass some data through `libidip` to all callback implementations.

```
struct my_user_data_t {
    char* video_file;
} my_user_data = {
    .video_file = NULL
};

idip_status_code_t
my_xxx_handler_impl(idip_stream_xxx_t* stream,
                    const char* target,
                    void* udata)
{
    struct my_user_data_t* my_user_data = (struct my_user_data_t*)udata;
    // using my_user_data
    // ...
}
```

You can define different data structures which will be passed to each handler type. Each `udata` pointer will be saved by `idip_server_setup_xxx()` function and then will be passed to handler call.

4.3 Create your own IDIP server

Here we're placing a step-by-step guide to fast implement your own IDIP server.

4.3.1 Start from a template

Get the `examples/skel` project as a template. It's a cmake-based project and implements a lot of stubs for handlers and a boilerplate code. Follow comments in the `examples/skel/src/main.c` file to modify the code. The project intializes the `libidip` library, creates an instance of `idip_server_t` object, then creates an instance of video source, registers callback stubs, runs a main loop, then does required finalization steps.

Note. If you want to use cmake then modify the `skel/CMakeLists.txt` as shown in the Build the server chapter.

Let's look at `skel` implementation to explain interaction with the `libidip`. `libidip` must be initialized before any library function call. Finalization step is also required. In our sample we call the `idip_init()` function in first lines of `main()` and the `idip_cleanup()` at the end of the `main()` function.

```
int main(int argc, char** argv)
{
    // handle the command line arguments
    // ...

    // Initialize library
    if( IDIP_OK != idip_init() ) {
        exit(EXIT_FAILURE);
    }

    // ...
    // ... application code
    // ...

    // Free resources
    idip_cleanup();

    return 0;
}
```

Inside the `main` we're creating an instance of IDIP server, registering handlers, video sources and then starting the instance of IDIP server.

```
// 1. Create an idip server instance
idip_server_t* server = idip_server_new(&server_config);
if( !server ) {
    // log_error(...);
    idip_cleanup();
    exit(EXIT_FAILURE);
}

// 2. Set up server features. The suffix _xxx_ is a generalized
// template of registration methods name
rc = idip_server_setup_xxx_handler(server, xxx_handler_implementation);
if( 0 != rc ) {
    // Error handling
    goto fail;
}
// ... and so on.

// 3. Set up live streaming sources
rc = idip_server_av_source_register(server, &av_source_params_main, &
    av_source_main);
if( 0 != rc ) {
    // Error handling
    goto fail;
}
```

```c
// do register other sources

// 4. Start the server
if( IDIP_OK != idip_server_run(server) ) {
    // Error handling
    goto fail;
}

// 5. Start live streaming.
// ...
// 6. The main loop
// ..
```

After the server creation we can interact with it in the main loop. Then after the main loop is done, the steps to stop the IDIP server follow.

```c
// 7. Release sources. Ensure these sources are accessing only
// from the server instance before stopping them.
idip_server_av_source_unregister(server, &av_source_main);
// release another sources

// Notify the server instance about stopping.
// Function blocks main thread until the server instance is stopped.
idip_server_stop(server);
```

To initiate an IDIP server shutdown procedure the application has to call an `idip_server_stop()` function after exiting from the main loop. To avoid deadlocks calling `idip_server_stop()` from a handler is strongly NOT recommended, since it’s designed to be called from the from main thread.

We met the skeleton application and steps for creating, running and stopping the IDIP server.

4.3.2 Change server configuration

The instance of the IDIP server requires a set of parameters before it starts. They grouped in a `server_config` structure. Define and initialize it like so:

```c
// Global server's configuration.
static idip_server_conf_t server_config = {
  .host_or_path = "127.0.0.1",            // you can use "*" here to allow
                                          // listen on all interfaces
  .port = 55500,                          // Standart IDIP service port
  .keep_alive_ms = 10000ul,
  .vendor = "Ivideon",                    // TODO: Put a real data here.
  .model = "Ivideon IDIP Demo Camera",    // TODO: Put a real data here.
  .serial_number = "0000000001",          // TODO: Put a real data here.
  .firmware_version = "a.b.c.d_xxxx_xx_xx_zzzzzzzz",

  .disable_ipv6 = 0, // Set to 1 in order to disable IPv6

  // Agentless support.
  // Set both fields to NULL when the agentless scheme isn't using.

  // Ivideon Cloud Agent URL (an example).
  .cloud_enpoint_url = "https://idip-v1.extcam.com/idip/uplink" ,
  // Ivideon Device Key string (request it on Ivideon).
  .dev_key = "idip_example-123abc",
};
```

- host_or_path specifies an IP address to listen to. In our case server will listen to localhost. To listen to all available interfaces set it to "0.0.0.0" or "*".
- port specifies a TCP port to listen to.
- keep_alive_ms is a keep-alive interval. In our example we use long (not a real-world) interval of 10 seconds. After that period server will close client connections if no queries are received.
- vendor specifies a vendor name.
- model string specifies a model name string. It is a "name-on-a-box" string in most cases.
- serial_number specifies a string with camera/DVR serial number. The number must be printed on a camera or DVR case. This string may be geberated by a rule agreed upon Ivideon team. In most cases the serial number could be generated by manufacturer. **Note:** the serial number can be used for a camera personalization (attaching) procedure.
- firmware_version specifies a manufacturer firmware version string.
- disable_ipv6 is a flag that disables using IPv6.
- cloud_enpoint_url an URL like https://idip-v1.extcam.com/idip/uplink that's using for agent-less integration. Both http:// and https:// shemes are supporting. When present an idip_server_t instance will act as WebSocket client and will try to connect to the given url.
- dev_key a vendor identifier string by Ivideon for agentless integration scheme.

Note. You get values of dev_key and cloud_enpoint_url from Ivideon Team.

Note. An IDIP server instance is always supports the scheme with the Ivideon Agent on the camera. Then you should always set up host_or_path and port variables. For agentless scheme set .port to 0 and .host_or_path to /tmp/idip.sock.

Note. The IDIP server supports two connection methods for agentfull scheme:

- TCP sockets. To use this one set up both host_or_path and port fields to corresponding values.
- Unix Domain Sockets. To use this one specify host_or_path to path to unix domain socket file (for example "/tmp/idip.sock") and set the port field to zero value.

4.3.3 Set Up Logging

Setup logging method if needed. The libidip supports internal logging API. By default it puts messages on stdout. In some cases, you will want to put logs to a different destination. The libidip supports an environment variable IDIP_LOG_URI. The value of the variable is url-like. For example, set IDIP_LOG_URI="udp://0.0.0.0:12345" to out log messages to a UDP server, or set IDIP_LOG_URI="file:///tmp/idip.log" to put log messages to the /tmp/idip.log file. To get back to put log messages to stdout clear the value of the variable or set it to IDIP_LOG_URI="stdout".

Note. A strong recomendation is to have an ability to change the environment variable IDIP_LOG_URI in your firmware.

Note. When `systemd` is using to start the idip server, default logging method will be prefer. The `journald` daemon and `journalctl` tool can help you to collect logs from the idip server.

The logging subsystem exports a logging API. Logging functions produces log messages of different levels (debug, info, notice, warning, error and fatal):

- `void idip_log_fatal(const char* fmt, ...)`
- `void idip_log_error(const char* fmt, ...)`
- `void idip_log_warning(const char* fmt, ...)`
- `void idip_log_notice(const char* fmt, ...)`
- `void idip_log_info(const char* fmt, ...)`
- `void idip_log_debug(const char* fmt, ...)`

All of these logging functions have printf-like syntax for format string. The result message should not exceed 1 Kb. You can use these functions after the `idip_init()` call. When the application done call the `idip_done()`.

4.3.4 Implement mandatory feature handlers and regster them

Minimal idip server implementation should provide three mandatory handlers:

- `idip_system_get_interfaces_handler_t` - to get interfaces state.
- `idip_stream_persistent_data_get_handler_t` - to get persistent data storef from the cloud.
- `idip_stream_persistent_data_set_handler_t` - to save some data from the cloud.

The `skel` project provides stubs for them:

- The `get_interfaces_handler()` returns information about network interfaces on the device. The stub returns hardcoded example data. Your implementation should get actual state for all interfaces and put information with the `idip_stream_system_get_interfaces_put_iface_data()` call. On Linux-based systemp the implementaion can use NETLINK subsystem to get state of interfaces. A good examples of NETLINK using you can find in iproute2 sources[2].
- The `persistent_data_get_impl()` function is a half-stub function. It uses a default implementation from `libidip` that read a file from device file system.
- The `persistent_data_set_impl()` function is also half-stub and also uses a file to store data from the cloud.

DO NOT use the `idip_persistent_data_get_default_implementation_helper()` and `idip_persistent_data_set_default_implementation_helper()` on read-only filesystems.

Refer the System handlers chapter to see handler implementation details.

Use the following calls to calls to register your handlers on the IDIP server instance.

[2] https://git.kernel.org/pub/scm/network/iproute2/iproute2.git

- idip_server_setup_get_interfaces_feature()
- idip_server_setup_persistent_storage_feature()

The sample code is in the skel project also.

4.3.5 Set up video sources

The skel project declares one live video source for target “camera0” and profile “0”. If device supports substreams, create additional vidoe sources and register them with the same target name and different profile_name values. In the skel the initialization structure declares a source whth H.264-encoded video stream and no audio:

```
idip_av_source_params_t av_source_params = (idip_av_source_params_t)
{
    .target_name = "camera0",
    .profile_name = "0",
    .v_info = (video_stream_info_t)
    {
        .codec = VIDEO_CODEC_H264,
        .width = 0,
        .height = 0
    },
    .a_info = (audio_stream_info_t)
    {
        .codec_type = AUDIO_CODEC_NONE,
        .sample_format = AUDIO_SAMPLE_FORMAT_FLOATP,
        .channels = 1,
        .sample_rate = 0
    },
    .limits = IDIP_AV_LIMITS_INIT(2000, 64, 5),
    .event_handler = NULL,
    .event_handler_data = NULL,
};
```

If your device supports audio streaming common, initialize a_info with correspoinding values. For example:

```
.a_info = (audio_stream_info_t)
{
    .codec_type = AUDIO_CODEC_AAC,
    .sample_format = AUDIO_SAMPLE_FORMAT_FLOATP,
    .channels = 1,
    .sample_rate = 44100
}
```

The limits value declares bufering limits for the video stream: average video bitrate (in kbps), average audio bitrate (kbps) and buffer amount in seconds.

Hint. The examples/demo-server shows advanced example where many video streams are creating.

An important fields in av_source_params structure are .event_handler and .event_handler_data . The handler has the following signature:

```
void
```

```
av_source_event_handler_impl(idip_av_source_event_t event,
                             const idip_av_source_t* source,
                             const void* event_data,
                             void* udata)
```

and can be called by `libidip` on events from the video source. The most important event is the `IDIP_AV_EVENT_KEYFRAME_REQUEST`. It's a hint to an encoder to put a key frame to thevideo source ASAP. Other event types can be used for power consumption purposes. On the `IDIP_AV_EVENT_FIRST_CLIENT_ENTER` event you can enable an encoder, then on the `IDIP_AV_EVENT_LAST_CLIENT_LEAVE` you can disable the encoder for power consumption.

Note: See Video streaming chapter for details about video sources.

Create and register required video sources with the `idip_server_av_source_register()` call. Then you able to put encoded video and audio data to the source in a separathe thread.

Note. The registered `idip_av_source_t` should be unregistered **before** the `idip_server_stop()` call.

4.3.6 Implement advanced feature handlers and register them

Follow the next chapters to implement advanced hadlers:

- Video streaming
- System notification
- Firmware upgrade
- Security
- Personalization
- Persistent storage
- Attachment token and QR-code support
- Archive integration
- Cloud configuration and events
- Sending events

4.3.7 Implement cloud configuration parameters

The chapter Cloud configuration and events describes all aspects of Cloud Configuration values.

Note: refer `examples/demo-server/src/demo.c` to see an example of Cloud Configuration values.

4.3.8 Send events from the main loop

While main loop is working, the server instance connects to the cloud, accepts commands and interacts with the Cloud Agent. This means that handlers and the main loop might be executed in parallel.

```
while(!done) {

    // ...
    //  poll events
    // ...

    if(motion_detected) {
        if( IDIP_OK != idip_server_put_event(server, "camera0", "motion",
                                             now_ms(),
                                                IDIP_EVENT_ARG_NONE,
                                                NULL) )
        {
            break;
        }
    }
    else if(other_event_type_detected) {
        // idip_server_put_event(server, ...);
    }
    else {
       // do nothing
    }
}
```

Warning. Here we define the main loop as a loop that check some `done` flag and finalizes when it's **`true`**. It's enough for demonstration purposes but might cause data races in some cases.

The purpose of a running main loop is to poll events and send them to IDIP clients. The event detection code itself is application-specific. Each event has a name ("motion" in our example) and must be associated with a detection timestamp. We show here a generic pattern of the `idip_server_put_event()` function usage. If it returns a value different from `IDIP_OK`, the main loop must exit. As it is shown in the example, the application has to implement some kind of `now_ms()` to be able to get a timestamp. All timestamps must hold milliseconds from the `Epoch` (00:00:00 Thursday, 1 January 1970, UTC). We recommend to use the `clock_gettime()` function with `CLOCK_REALTIME` argument (requires `-lrt` linker flag) and convert its output to milliseconds.

You can find a working example in the `examples/demo-server/src/demo.c` file.

4.3.9 Build the server

To compile your own IDIP server you have to append to your project:

1. A path where you have installed libidip headers.
2. The libidip.a library and its dependencies (msgpack, libwebsockets, mbedx509, mbedcrypto, pthread).

A very simple example. Replace the value of `IVESDK_DIR` to a path where you have the Ivideon Embedded SDK installed. Then you able to build your IDIP server with cmake. Do following steps:

```
export IVESDK_DIR=$HOME/.local/ivesdk
cd /path/to/skel-copy
gcc -I$(pwd) -I$IVESDK_DIR/include -o skel.o -c src/main.c
gcc skel.o -o skel -L$IVESDK_DIR/lib \
    -Wl,-rpath,$IVESDK_DIR/lib -lidip -lwebsockets -lmsgpackc \
    -lmbedtls -lmbedx509 -lmbedcrypto -lpthread -lm -lrt
```

Build of your ouwn IDIP server has been done. The result is in the `build/skel`.

4.4 Onboarding test

Build Embedded SDK and the modified `skel` project before you start. You need to specify a real MAC-address and a real serial number in the modified `skel` project. These MAC-address and S/N should be unique, hence please make sure that no other device in your project will use the same data.

After that make sure that these pre-conditions are observed:

- If you already have a test account, please log into it. If you have not registered an account yet, please create it via the Web portal or mobile apps (for iOS or Android) and then log in.
- Make sure that the persistent data file doesn't exist or it's empty.

Attach the the `skel` application to your account by following these steps:

1. Open the Cameras tab in your test account and press the "Connect device" button.
2. Follow the onscreen instructions: select the option Camera and proceed with its connection via S/N or MAC-address - choose the connection flow which is more convenient for you to follow.
3. When the onscreen instructions ask you to power on the device, start your IDIP server application (`skel`).
4. After that IDIP server will establish connection to the cloud and attaches to your test account.
5. You will be asked to give a name to the attached server and select a pricing for it.
6. Activate a pricing plan for IDIP Server instance to watch demo video in your account.

5 System handlers

The `libidip` supports several system-wide handlers. Unlike streaming handlers (live streamer, archive, etc.) these handlers aren't bound to specific targets. They are system-wide. Therefore, no target name is required at handler registration.

5.1 System notification

The first one is `idip_stream_system_notify_handler_t`. It allows informing the device about system and cloud-related events.

```
idip_status_code_t
idip_stream_system_notify_handler_t(
        const idip_system_notify_query_args_t* args,
        void* udata);
```

The handler can be registerred via the method:

```
int
idip_server_setup_system_notify_handler(
        idip_server_t* server,
        idip_stream_system_notify_handler_t handler,
        void* udata);
```

Registered handler is calling for following events:

Event Name	Sender	Description
init.unbound	SDK	The `libidip` didn't detects saved parsistet data during initialization
init.bound	SDK	The `libidip` found saved parsistent data during initialization
bind.ok	Cloud	The device has been successfully attached to the Cloud (by any reason).
bind.failed	Cloud	The device has failed to attach to the Cloud (by any reason).
bind.token.wait	SDK	The device is waiting for Token and/or WiFi settings (e.g. via QR code)
bind.token.accepted	SDK	An information about Token and/or WiFi settings has been successfully received and parsed.
bind.token.wifi.ok	SDK	The WiFi has been configured successfully.
bind.token.wifi.failed	SDK	The WiFi has not been configured due to an error.
bind.token.ok	Cloud	The device has been successfully attached to the Cloud by token.
bind.token.failed	Cloud, SDK	The device cannot be attached to the cloud using provided Token and WiFi settings.
bind.removed	Cloud	The device has been detached from the Cloud.
connect	SDK	Connection to the cloud is in progress.
online	Cloud	Camera is connected and is ready to interoperate with the cloud
offline	Cloud, SDK	The device is bound, but is not connected to the Cloud or connection to the cloud was lost.
upgrade	Cloud	Software upgrade process is started. The camera is offline, all network interfaces may be down.

The application can subscribe to notifications via `idip_server_setup_system_notify_handler()` function (see the `idip.h` file) and receive events. These event names are passed to the handler as strings. The device can use this notification to show its status and to implement some recording features (e.g. start recording to local storage when cloud connection is lost), security, etc.

Events named “bind.token.*” will occur during the attaching process with QR-code. Other events will comming during a basic attach procedure and at normal work.

5.2 Firmware upgrade

The second system handler type `idip_stream_update_version_handler_t` allows to implement the **OTA feature**. Register the handler implementation with `idip_server_setup_update_handler()` call. The `libidip` calls the handler when the cloud initiates a firmware upgrade procedure. The `libidip` passes a firmware fetching URL (http or https) to the handler. It's a main scenario. The implementations should download the firmware image from the URL and start the firmware upgrade procedure. The `libidip` hasn't any requirements for the firmware download location in the case. The firmware might be saved in RAM, `/tmp` or onboard or on external flash or somewhere else. The location of the new firmware depends on the preferrable firmware upgrade techique.

Warning. With the "agentfull" scheme you have to specify `idip.firmwareUrl` parameter to a `videoserverd.conf` file. It should be like so:

```
{
    "idip": {
        "url": "idip://127.0.0.0:55500",
        "firmwareUrl": "https://cdn.server.net/downloads/firmware-{|
           version|}.zip"
    }
}
```

The `idip.firmwareUrl` is a URL template where the `{|version|}` expression is a template parameter that would be replaced by the cloud with an actual version string value.

5.3 Security

The next system handlers allows to modify system account password and get a list of system accounts to the `videoserver`. They are:

- `idip_stream_passwd_handler_t` and
- `idip_stream_system_get_users_handler_t`.

The `libidip` calls the first handler to set up a new password for a system account. The handler may be called with empty (NULL) user name. The implementation should change current system account password. If a user name was passed to the handler it shoud change the password for a given user name. `idip_stream_system_get_users_handler_t` should return a list of system users which passwords can be modified by the `videoserver`. If your device implements the password change feature then you should implement and register these handlers by `idip_server_setup_account_management_handlers()` call.

Note. In most cases, a camera is accessible from factory or after factory reset with default credentials.

When the camera is attached to the cloud we recommend changing the password to a randomly generated string. Then the camera will accessible only from a cloud account. Only the cloud user will able to update a password. The cloud do not generate and send the random password by security reasons. The device should generate the password by itself. When device was attached to the cloud, the videoserver sends a "bind.ok" message to the system notify handler in libidip. Then you can catch the string in your handler implementation and generate a new random password. A proof-of-concept code is there.

```c
int main()
{
    // ...
    // Register a system_passwd handler in the server instance.
    // The handler will be called when a user changes password
    // from its account.
    rc = idip_server_setup_account_management_handlers(server,
                                   system_passwd, NULL, NULL, NULL);
    if( 0 != rc ) {
        // handle the error
    }

    // register a system_notify_impl handler on the server instance
    // The handler will be called for some events including the "bind.ok"
       message.
    rc = idip_server_setup_system_notify_handler(server,
                                                 system_notify_impl, NULL)
                                                     ;
    if( 0 != rc ) {
        // handle the error
    }
    // ...
}

// system password change handler
idip_status_code_t
system_passwd(const idip_user_credentials_t* cred, void* udata)
{
    // The cred->user can be NULL or points to an empy string.
    // Then the handler should set a password for a default system user or
    // change an password for a given user name instead.
    if( cred->user ) {
        if( 0 != strcmp(cred->user, admin_name) ) {
            // error handling
            printf("User not found: %s\n", cred->user);
            return IDIP_GENERIC_ERROR;
        }
    }

    // check if the system password was already set by the user
    // direclty (using passwd command or some else)
    if( TheSystemPasswordWasSetDirectly() ) { // not the libidip API
      // we can't setup a new password in the case.
      // log_error(...)
      return IDIP_GENERIC_ERROR;
    }

    // the cred->password contains a new password
    // Set the new password for all services of the camera:
    // (These methods are not libidip API)
    SetSystemPasswd( cred->password ); // for local access (throught tty/
       ssh/etc)
    SetONVIFPasswd( cred->password ); // for onvif access
    SetRTSPPasswd( cred->password );  // for rtsp access
    SetWEBPasswd( cred->password );   // for web ui access
    // ... and so on
    // SaveNewPassword( cred->passwd ); // if it's required
    printf("Password for '%s' was changed to '%s'\n", admin_name,
        admin_password);
    return IDIP_OK;
}

// The system.notify handler implementation
idip_status_code_t
system_notify_impl(const idip_system_notify_query_args_t* args, void*
    udata)
{
    // catch a "bind.ok" event.
    // The camera was just attached to the cloud
    if( 0 == strcmp(args->name, "bind.ok") ) {
      // Generate random password. Save a flag that password wasn't
      // changed by user directly.
      char* password = GenerateAStrongRandomPassword(); // is not libidip
          API
      // Set the new password for all services of the camera:
      // (These methods are not libidip API)
      SetSystemPasswd( password ); // for local access (throught tty/ssh/
          etc)
      SetONVIFPasswd( password ); // for onvif access
      SetRTSPPasswd( password );  // for rtsp access
      SetWEBPasswd( password );   // for web ui access
      // ... and so on
      // SaveNewPassword( password ); // if it's required
      free(password);
    }
    // other events handling
    else if( 0 == strcmp(args->name, ... )) {
      // ...
    }
    return IDIP_OK;
}
```

5.4 Persistent storage

The device should be able to store some data into energy independent storage - the persistent storage. The cloud will store there cloud account data aand some other data attaching to the cloud (a personalization) needs to store. **The feature is required.** Here we describe the persistent storage API.

A manufacturer should implement two handlers:

- the first one of type `idip_stream_persistent_data_get_handler_t`,
- the second one of type `idip_stream_persistent_data_set_handler_t`.

Each handler can pass its own data pointer from the `libidip`. You have to register both handlers and specify its data pointers by a `idip_server_setup_persistent_storage_feature()` call.

The `libidip` calls the `idip_stream_persistent_data_set_handler_t` during attaching procedure to store account details on the device or to clear current account data. Do not interpret the data passed to the handler, just store them as is.

The `idip_stream_persistent_data_get_handler_t` handler may be called some times after the videoserverd connected. The handler implementation should read a data blob stored before if its exists and put them via idip_stream_persistence_data_put() function to the cloud. Wnen no data exist the handler should pass an empty (zero length) buffer to the idip_stream_persistence_data_put().

Note: The implementation should set up both handlers via idip_server_setup_persistent_storage_feature() call. We strongly recommend register persistent storage handlers before the idip_server_run() call. Otherwise, attaching issues might occur.

Note: the libidip queues these handlers, but they will be executed in a separate thread..

The libidip provides two helpers for a fast implementation of persistent storage handlers. They are:

- idip_persistent_data_get_default_implementation_helper() and
- idip_persistent_data_set_default_implementation_helper().

We use them in our examples. But **DO NOT** use these helpers in production code! They are using regular files to get and store data, and might not fit your device store policy (e.g. you have squashfs).

5.5 Attachment token and QR-code support

Devices with a WiFi interface able to connect to a wireless network by using a simple attachment procedure. A user run Ivideon Client Application on its own Android or iOS smatrphone, begin to append a new device then the mobile app shows a QR code which contains an **Attachment token** (synonym is the **System token**). The **Attacment token** is a JSON-array with a special format. The token contains user's wireless network credentials (SSID and password) and a identifier from the Cloud. Then a user show this QR-code to a camera then the camera will connect to user wireless network and then will attach to the cloud.

Note. Passing a Attachment Token by a QR-code isn't only one way. An alternative might be a sound codes or some else.

When the libidip requires the Attachment Token it calls a feature handler idip_system_token_handler_t with the follow prototype:

```
idip_status_code_t
idip_system_token_handler_t(idip_stream_system_token_t* stream, void*
    user_data);
```

Register your handler with the next call:

```
int idip_server_setup_token_handler(idip_server_t* server,
                                    idip_system_token_handler_t
                                        token_handler,
                                    void* token_handler_udata);
```

The implementation should get a QR-code from encoder, parse this QR. The parsing result is the Attachment Token in JSON format. The libidip provides a method idip_parse_cloud_qr_code_token_json() for parsing token to a view that compatible with the libidip. Look at the example of using the helper.

```
idip_parsed_token_t token = { 0 };
```

```
int rc = idip_parse_cloud_qr_code_token_json(json, json_len, &token);
if( 0 == rc ) {
    // Put token to the libidip
    rc = idip_stream_system_token_put_token(stream, &token);
}
idip_parsed_token_destroy(&token); // release token resources
```

After Attachment Token was got it should passed to the `libidip` by the `idip_stream_system_token_put_token()` call.

> **Note.** Passing token will produce some system events.

6 Personalization procedure

Personalization or attaching is a procedure that bounds a camera to a user account. The `videoserver` implies the feature. It supports different kinds of personalization:

- by a device mac address
- by a device serial number
- by **QR-code***

In this chapter we focus on personalization details common with the IDIP protocol and the IVESDK/`libidip`. Attaching by mac address and by serial number is simplier, the QR-code attaching is more complex procedure and requires more handling.

To support the feature the device:

- Should implement persistent storage handlers. The personalization procedure stores a bit of data on the device. Then the device should have a data storage with read-write access and should allow to store about 1-2 KiB binary data. The storage should be energy independent and keep the data when external power is off. The `libidip` assumes that it can store a data blob which then will be requested by `idip_stream_persistent_data_get_handler_t` handler.
- Should implement a `idip_stream_system_notify_handler_t` handler. This handler allows to catch notifications that would be sent during personalization. See details in System notification chapter.
- Set all the required fields in the `idip_server_conf` object (see `idip_server.h`).

Additionaly, for **QR-code supporting** should be implemented:

- Snapshot feature (see `idip_server_setup_snapshot_feature()` function in the `idip.h` file). The handler should support the special profile with name **"qr"**. The response data for such a request should contain either an image in the jpeg format (codec `VIDEO_CODEC_MJPEG`) or a data from the parsed QR-code in the text format (codec `VIDEO_CODEC_TEXT`).
- Setup of the WiFi feature (see `idip_server_setup_wireless_feature()` function in the `idip.h` file).

> **Note:** attaching by QR-code produces advanced events (`"bind.token.*"`). Refer to the table in a System notifications section.

To set up personalization throught IDIP you can use IDIP-autoconfiguration feature from `videoserver` or you can explicitly append a `"personalization"` section to the `videoserverd.config` file. The first example is showing how to use IDIP-based automatic configuration. Do append the `"idip"` section to videoserverd.conf configuration file, and remove all items from the `"cameras"` section:

```
"idip": {
    "url": "idip://127.0.0.1:55500"
},
"cameras": [],
...
```

Warning. Advanced steps to modify the `videoserverd.config` file are applicable for “agent-full” scheme only.

With such configuration the `videoserver` will ask the idip-server (on 127.0.0.1:55500) about its features: target names, profile names, implemented methods, and other features such as personalization settings. Then it appends all live streams info, archives info, etc., and makes a dynamic configuration for cameras, archives and other features.

Alternatively you can explicittly append only the `"personalization"` section (withoit the `"idip"` section) like so:

```
"personalization": {
    "macAddr": "0a1b2c3d4e5f",
    "processor": {
        "type": "idip",
        "url": "idip://127.0.0.1:55500"
    }
}
```

In the case `videoserver` will get and set account data throught IDIP protocol. But no automatic camera detection will be.

In very special cases you want to use idip-based automatic configuration and don’t want to use idip-based personalization, then you should define a different personalization type. For example, you may set up a basic LUA-based personalization:

```
"personalization": {
    "macAddr": "0a1b2c3d4e5f",
    "processor": {
        "type": "lua",
        "path": ":vs_attach.lua"
    }
}
```

Note: data from a persistent storage will not be used in this case.

6.1 Troubleshuting

Check the following:

- Does a camera removed from any account?
- A mac address or a serial number from `demo.c` or `skel.c` isn’t using for the device?

- Does the `videoserverd.conf` lacks the `"account"` section?

Reset settings to factory defaults:

- Restore contents of default `videoserverd.conf` file.
- Do sure the videoserverd.conf file uses the correct scheme. The `"idip"` section is present and points to a correct url. The `"personalization"` section is present (if needed) and contains `processor.type` value (`"idip"`).
- Clear persistent storage data. The `idip_stream_persistent_data_get_handler_t` handler should return empty data.
- Reset system password to default.

7 Video streaming

Live video streaming is the main feature of a camera or DVR. We support two different methods to implement the feature. The preferred way is to use `idip_av_source_t` type and related API. The second - is to implement an `idip_av_handler_t` handler.

7.1 Live streaming source

An recommended way to implement the live streaming feature is to use an `idip_av_source_t` interface. In most cases capture API works as a notification about captured (or encoded) frame data. In case of `idip_stream_av_handler_t` you have to deal with data polling, buffering and synchronization problems, which are already solved in `idip_av_source_t` as a data sink interface, that accepts packets and allows reconfiguration on bitrate or resolution changes (e.g. from the `idip_stream_config_update_handler_t`).

To use the `idip_av_source_t` you should initialize an `idip_av_source_params_t` structure with target and profile names, video and audio descriptions (`v_info` and `a_info` fields, correspondingly), buffering parameters that are calculated from the GOP properties. You can use camera default settings or previously saved setting values. The `idip_av_source_params_t` structure requires to fill buffering parameters for the source. These parameters could be calculated from GOP.

```
idip_av_source_params_t av_source_params = (idip_av_source_params_t)
{
    .target_name = "camera0",
    .profile_name = "0",
    .v_info = (video_stream_info_t)
    {
        .codec = VIDEO_CODEC_H264,
        .width = 0,
        .height = 0
    },
    .a_info = (audio_stream_info_t)
    {
        // Use AUDIO_CODEC_NONE if a sound is disabled
        .codec_type = AUDIO_CODEC_AAC,
        .sample_format = AUDIO_SAMPLE_FORMAT_FLOATP,
        .channels = 1,
        .sample_rate = 44100 // Set to 0 when audio stream is disabled
    },
    // video bitrate 2000 kbps, audio bitrate 64 kbps, max buffer length 5
       sec
    .limits = IDIP_AV_LIMITS_INIT(2000, 64, 5),
    .event_handler = NULL,
    .event_handler_data = NULL
};

// declare a pointer to a video source instance
idip_av_source_t* av_source = NULL;
```

An important fields in `av_source_params` structure are `.event_handler` and `.event_handler_data` . The handler has the following signature:

```
void
idip_av_source_event_handler_t(idip_av_source_event_t event,
                               const idip_av_source_t* source,
                               const void* event_data, void* udata);
```

and can be called by `libidip` on events from the video source. The most important event is the `IDIP_AV_EVENT_KEYFRAME_REQUEST`. It's a hint to an encoder to put a key frame to thevideo source ASAP. Other event types can be used for power consumption purposes. On the `IDIP_AV_EVENT_FIRST_CLIENT_ENTER` event you can enable an encoder, then on the `IDIP_AV_EVENT_LAST_CLIENT_LEAVE` you can disable the encoder for power consumption.

A video source could be registered by calling the `idip_server_av_source_register()` function. To push video and audio data use `idip_av_source_push()`, `idip_av_source_push_video()` and `idip_av_source_push_audio()` functions. **Pay attention** that the lifetime of the `idip_av_source_t` should be less than the lifetime of the `idip_server_t` instance. In other words, you should call `idip_server_av_source_unregister()` **before** stopping the server.

```
// somewhere inside main()
rc = idip_server_av_source_register(server, &av_source_params, &av_source)
    ;
if( 0 != rc ) {
    // log_error(...);
    goto fail;
}
```

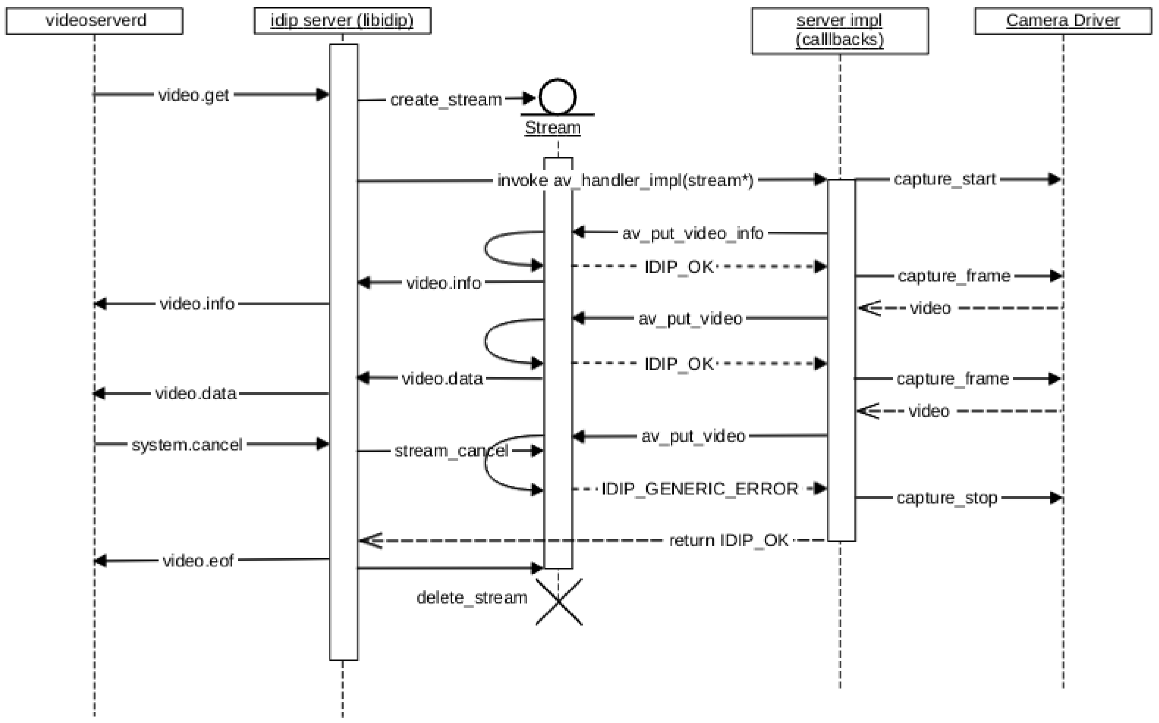
When idip server is running you can put encoded video and audio frames to the source. It's possible to do it from another thread or from the main loop. In the example below, we're getting a frame from a video codec and then putting the frame to the source in the context of `main()`.

```
int main()
{
    // ...
    // initialization steps

    // The main application loop
    while( !done ) {

        void* frame;
        size_t frame_size;
        bool is_key_frame;
        int64_t dts; // decoded time, milliseconds in UTC
        int64_t pts; // presentation time, milliseconds in UTC

        // get an encoded video frame from the codec
        video_codec_get_encoded_frame(codec, &frame, &frame_size, &
           is_key_frame, &dts, &pts)

        // Put the encoded frame to the source
        int rc = idip_av_source_push_video(av_source,
                                            pts,
                                            frame,
                                            frame_size,
                                            is_key_frame);
        if( rc != IDIP_OK ) {
            printf("idip_av_source_push_video() returns %d\n", rc);
            // break; // if you're in thread
        }

        // get an encoded audio frame from the codec
        audio_codec_get_encoded_frame(codec, &frame, &frame_size, &dts, &
           pts)

        // Put the encoded frame to the source
        int rc = idip_av_source_push_audio(av_source,
                                            pts,
                                            frame,
                                            frame_size);
        if( rc != IDIP_OK ) {
            printf("idip_av_source_push_audio() returns %d\n", rc);
            // break; // if you're in thread
        }

        // some other things
        // ...
    }

    // ...
    // finalization steps
}
```

When a user requests some changes in codec settings you should reconfigure the codec and update the `idip_av_source_t` instance by the `idip_server_av_source_update()` call.

7.2 Live streaming handler

`idip_stream_av_handler_t` handler is a low-level way to implement a live streaming feature and requires more wrapping code. But it can help to understand how video and audio data could be transferred to the cloud. There is a sequence diagram where video data transfer is shown.



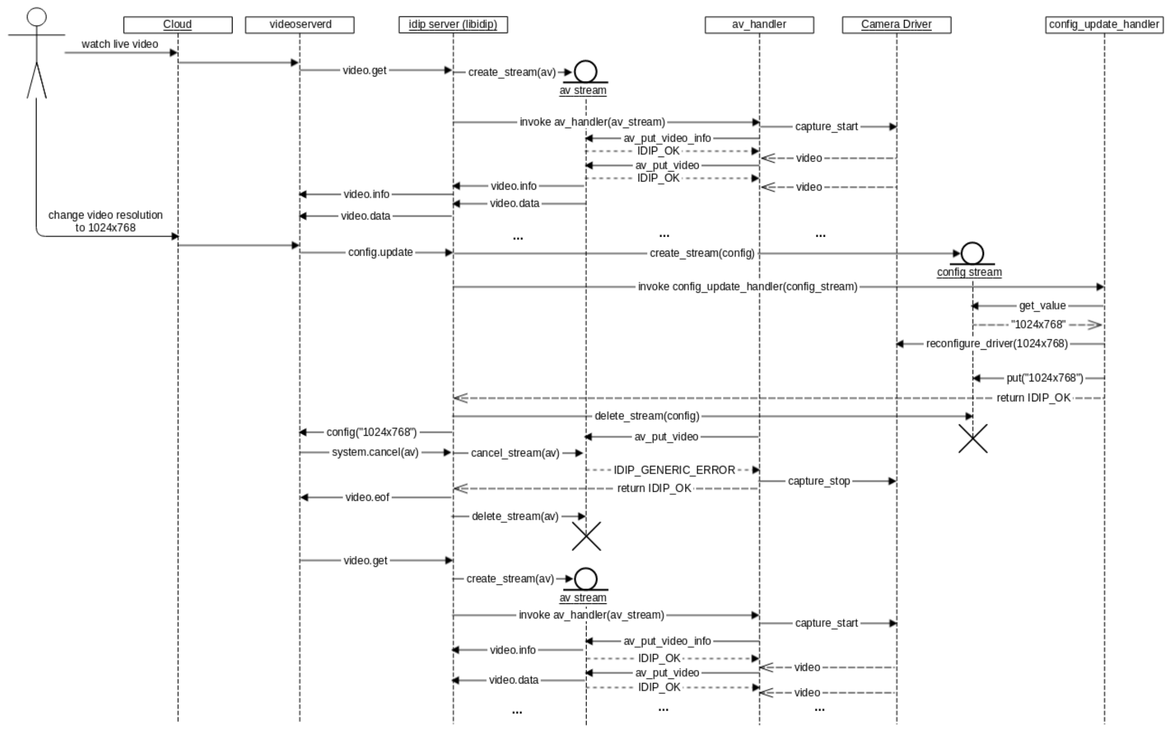
When the cloud requests live video it asks the videoserver app. Then videoserver sends **video.get** request by IDIP protocol. libidip creates an object that should be used as an output stream interface in the handler. Then libidip calls an idip_stream_av_handler_t implementation and passes the created stream to it. The implementation should start video (and audio) capturing and encoding. The handler should put to the output stream video/audio codec info before the video/audio data. All data put to this stream is sent to the videoserver, which in turn passes it to the cloud.

videoserver can cancel live video streaming (e.g. when a cloud configuration was changed). It sends a **system.cancel** request through IDIP protocol. The handler implementation should always checks return codes of each idip_stream_av_put_xxx() call. When a return code differs from IDIP_OK, the handler must return, or it will result in undefined behavior.

We recommend using the following profile naming rule: the best quality stream should be named as “0”, the substream - as “1”, the lowest quality stream - as “2”. For DVR with many sources, you should name these streams as N*65536 + K, where N is a camera number in DVR, K - is a profile number of N-th camera. The same rule is applied for the stream naming in the cloud configuration.

```
static idip_status_code_t
av_handler_impl(idip_stream_av_t* stream,
                const idip_stream_av_args_t* args,
                void* udata)
{
    // some user's context. It was saved from a
    // idip_server_setup_live_feature() call.
    struct user_data_t* context = (struct user_data_t*)udata;

    // User can check target.
    // Put some business logic here to switch between streams...
    if( 0 != strcmp(args->target, "camera0") ) {
        return IDIP_INVALID_ARGUMENTS;
    }
    // ... and profile names.
    if( 0 != strcmp(args->profile, "0") ) {
        // open the best quality stream
    }

    // set up "hardware" to given quality
    // and configure video buffering (if available)

    // Construct audio and video information
    video_stream_info_t v_info = {
        .codec_type = VIDEO_CODEC_H264
    };

    // send stream description

    // Each idip_stream_xxx() function returns 0 when information was
       accepted
    // by libidip, and negative value when libidip requires to stop (done)
        .
    //
    // NB! User MUST checks return values of these functions.
    if( IDIP_OK != idip_stream_av_put_video_info(stream, &v_info,
       IDIP_INFINITE_TIMEOUT) )
        goto end;

    // Playing streams up to IDIP server accepts data

    for(;;) {
        // Capture video frame buffers.
        if (hardware_get_h264_frame(packet) < 0)
            break;

        // User MUST checks return value to determine exit request
        //
        // All idip_stream_xxx_put_yyy() functions are synchronous. They
           do not
        // copy data to avoid performance troubles
        idip_status_code_t rc;
        rc = idip_stream_av_put_video_frame(stream, packet.is_key, packet.
           v_ts,
                packet.video_buf, packet.video_size, IDIP_INFINITE_TIMEOUT
                   );
        if( IDIP_OK != rc )
            break;

     }

end:
    // finalize hardware streams

    return IDIP_OK;
}
```

7.3 Live streaming reconfiguration



7.4 Archive feature

Here is a skeleton of the `idip_stream_archive_handler_t` implementation.

```
// Signature corresponds to the idip_stream_archive_handler_t type
idip_status_code_t
archive_handler_impl(idip_stream_archive_t* stream,
                     const char* target,
                     const idip_archive_play_params_t* params,
                     void* udata)
{
    // Analyze target name and playback parameters.
    // Search for the corresponding record and open it.
    // ...

    // Hold stream traits
    video_stream_info_t v_info;
    audio_stream_info_t a_info;

    // Setup hardware with requested settings.
    // ...
    // Fill v_info and a_info with the corresponding data
    // ...
    // Send v_info and v_info and check the return value.
    int rc = idip_stream_ar_put_video_info(stream, &v_info);
    if( IDIP OK != rc ) {
        // return from the handler immediately because the exit
        // condition is met.
        return IDIP_OK;
    }
    rc = idip_stream_ar_put_audio_info(stream, &a_info);
    if( IDIP OK != rc ) {
        return IDIP_OK;
    }

    // Start streaming
    for(;;) {
        // Read video and audio data.
        // ...

        // Send video and audio data.
        // We assume that is_key_frame and now_ms are supplied by
        // the application code.

        rc = idip_stream_ar_put_video(stream, is_key_frame, now_ms,
                v_buf, v_buf_size, IDIP_INFINITE_TIMEOUT)
        if( IDIP_OK != rc ) {
            return IDIP_OK;
        }
        rc = idip_stream_ar_put_audio_info(stream, now_ms, a_buf,
                a_buf_size, IDIP_INFINITE_TIMEOUT)
        if( IDIP_OK != rc ) {
            return IDIP_OK;
        }
    }

    // do clean up actions
    // ...
}
```

8 Archive integration

This chapter is about local video archive integration using the Ivideon Embedded SDK.

8.1 Archive abstraction

Ivideon Embedded SDK assumes that a video archive is a set of video records and each record has following attributes:

- start timestamp
- duration
- video and audio codec info
- optional meta-information

Ivideon Embedded SDK expects that data storage implementation provides:

- Absolute UTC timestamps in microseconds (or a conversion method).
- Looking up for records by a timestamp and a stream identifier.
- Looking up for the video record that is next to the current.
- Playing archive records forward with different speeds: 1x - required, 2x, 4x, 8x, 16x, 32x - desirable, 1/2x, 1/4x, … - optional.
- Playing archive records backward with different speeds (optional): -1x, -2x, -4x, etc.
- Playing video records from a given offset from the start.
- Playing video records frame by frame (optional).

8.2 UX and callbacks of libidip

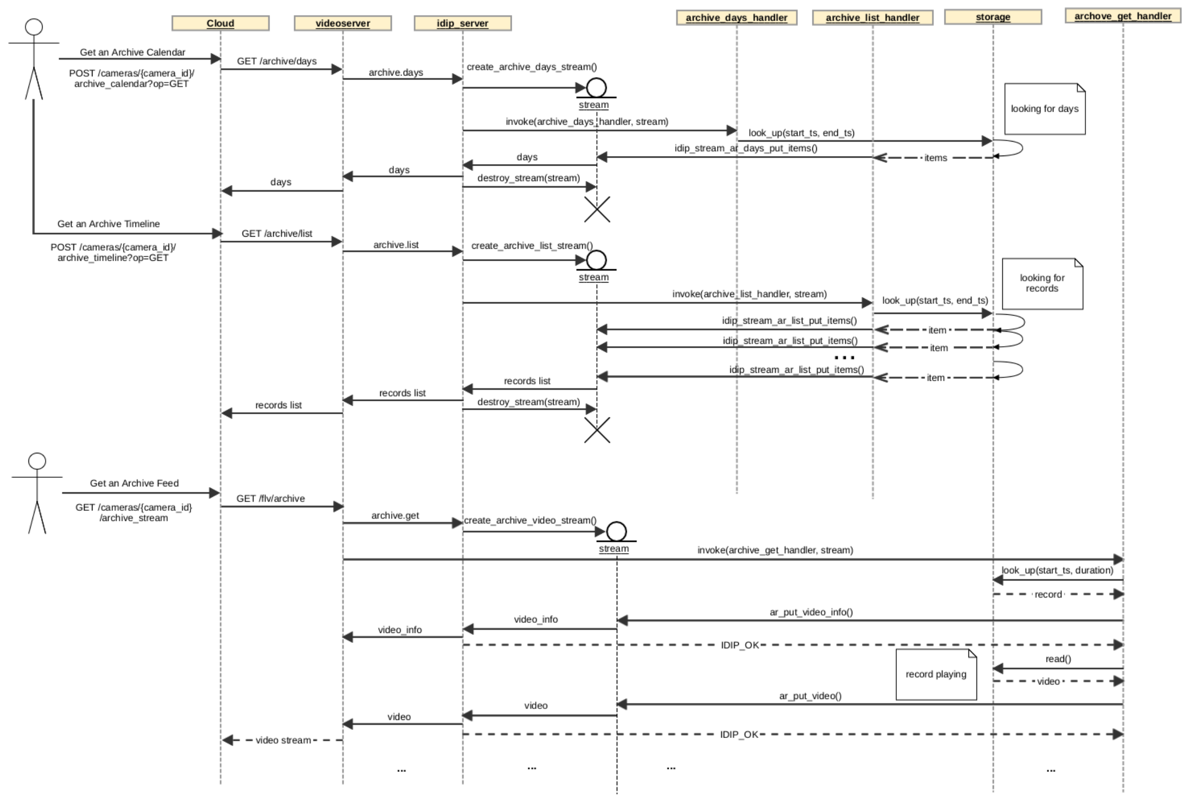
The main goal of `libidip` is to provide to customers access to video records. To support the feature WEB UI has a timeline to navigate records within a day and a calendar widget that gives an ability to find and select a day with records. They are require implementation of:

- `idip_stream_archive_days_handler_t()` - a handler for searching for days with records;
- `idip_stream_archive_list_handler_t()` - a handler for listing available video records for a specific day (timeline feature).

To be able to play the record a `idip_stream_archive_handler_t()` callback has to be implemented and support following playing modes:

- Continuos - play the records one after another;
- Single - play only the selected record;
- One shot - show only one key frame (same as snapshot).

All callbacks are called from different threads. Image below shows sequences for all UX cases.



8.3 Guide to callback implementation

In this part we assume that you have an archive implementation specific for your storage and hardware.

8.3.1 List of archive records

At first we show skeleton implementation of callback that is used for timeline. Its type is defined in `idip.h` as:

```
idip_status_code_t
(*idip_stream_archive_list_handler_t)(idip_stream_archive_list_t* stream,
                                      const char* target,
                                      const archive_interval_t* params,
                                      void* udata);
```

Let's name our own implementation as `archive_list_handler`:

```
idip_status_code_t
archive_list_handler(idip_stream_archive_list_t* stream,
                     const char* target,
                     const archive_interval_t* params,
                     void* udata)
{
    //
}
```

The function accepts 4 arguments:

- `stream` is an output stream instance for search results sending, its lifetime limited to the call-back's scope. **Don't try to save the pointer between different calls of the callback.**
- `target` is a name of an archive subsystem: different cameras in DVR or streams in a camcorder are named differently.
- `params` is a pointer to search arguments. It contains a time interval that will be used to search records within.
- `udata` is a pointer to user data, its value is set during the initialization of `idip_server_impl_t` instance.

The code below shows main strategy to search videos in archive.

```
idip_status_code_t
archive_list_handler(idip_stream_archive_list_t* stream,
    const char* target, const archive_interval_t* params,
    void* udata)
{
    // select target. For example, we switching
    // between main stream and sub stream
    if( 0 == strcmp(target, "main_stream") ) {
        // using archive for main stream
        // ...
    }
    else if( 0 == strcmp(target, "sub_stream") ) {
        // using archive for sub stream
        // ...
    }
    else {
        // error. unknown target.
        return IDIP_INVALID_ARGUMENTS;
    }

    // Unefficient way. For algorithm understanding only
    for(int i = 0; i < count_of_archive_records; ++i) {
        // Filter records by start time.
        // We assume that archive_records is an interface to enumerate
           videos.
        if( archive_records[i].start >= params->start_ms &&
            archive_records[i].start <= params->end_ms )
        {
            archive_record_t item = {
                .start_ms = archive_records[i].start,
                .duration_ms = archive_records[i].duration
            };
            // Put record to output stream
            int rc = idip_stream_ar_list_put_items(stream, &item, 1,
               IDIP_INFINITE_TIMEOUT);
            if( IDIP_OK != rc ) {
                break; // cancellation detected
            }
        }
    }

    return IDIP_OK;
}
```

Note. `idip_stream_ar_list_put_items()` accepts an array so batch sending is also possible.

In the given example we assume that information about stored videos (duration and start time) can be accessed by an index. It's not the optimal way, your implementation should work better than linear search. But data storage specific is not an area where libidip could works.

Last step is to register the `archive_list_handler` callback in the `idip_server_impl_t` instance. See `demo.c` for details.

8.3.2 List of per-day archive records

To implement the calendar feature `libidip` declares a special callback type:

```
idip_status_code_t
(*idip_stream_archive_days_handler_t)(idip_stream_archive_days_t* stream,
                                     const char* target,
                                     const archive_interval_t* params,
                                     void* udata);
```

As you may see, the prototype of `idip_stream_archive_days_handler_t()` is the same as for the `idip_stream_archive_list_handler_t()` callback. If a day contains several records, then we put only the first day record. The picture explains the algorithm.

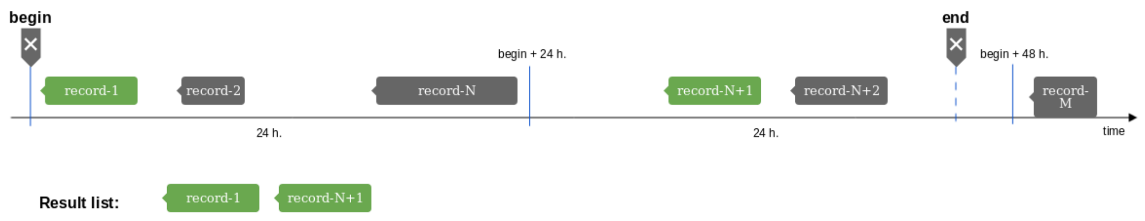


Figure 1: -

The `idip_stream_archive_days_handler_t` returns list of records where each record is the first record of a 24-hour `[start + 0, start + 24h)` interval, next record is a first record from a `[ start + 24h, start + 48h )` interval and so on.

To put results into output stream use `idip_stream_ar_days_put_items()` call. It works similarly to `idip_stream_ar_list_put_items()` call.

Requirements for this callback are the same as for `idip_stream_archive_list_handler_t`. **Don't try to save the `stream` pointer between different calls of the callback!**

8.3.3 Playing records

When user requests an archive record playback, the `idip_stream_archive_handler_t` is called. This function acts like handler which plays live video. It's called once per user request. You must implement a loop where video frames are put to the output `stream`. And you must return immediately when one of `idip_xxx_put()` functions returns a value that differs from `IDIP_OK`. The only difference between the `idip_av_xxx_put()` and the `idip_ar_xxx_put()` families is that the `idip_ar_xxx_put()` functions require additional timestamp - the absolute frame capture time (UTC, in microseconds).

9 Cloud configuration and events

Cloud configuration is a set of values (settings). Each setting has a certain format and type and this document uses JSON as a pseudo-language for describing them. All settings have getters and setters. A getter returns the current value and some bound meta information, a setter should initiate an update-and-apply procedure.

Some settings are related to events that IDIP implementation emits. `libidip` doesn't generate events automatically, but we provide some helper functions to send them. They are wrappers around the generic event sending function named `idip_server_send_event()`. Developers also can use this function directly to send events and a set of `idip_custom_arg_xxx()` to create custom event arguments and pass them to the `idip_server_send_event()`. The structure of each event will be described below, in JSON notation.

9.1 Configuration value common pattern

Each type of value has three fields:

- **name** - a string, that contains logical name. In most cases it is a sequence of words that delimited with a slash ('/') sign. Slash is used as logical delimiter.
- **value** - current value, which type is described in **meta.type** field. See below.
- **meta** - advanced data. It contains one or two fields:
  - **type** - it is a string with name of type. This field is always present.
  - **range** - an array of valid values or **min**, **max** and **step** of a range. In other words, content of array is depends on **meta.type** value.

9.2 Value types

Here is a list of configuration value types with short descriptions. Each type name defines setting value type and setting range type.

Type	Range	Value type	Description
bool	none	true or false	Boolean type
int	[min, max, step]	64-bit integer	range field is always present. It must contains an minimal, maximal values of setting and step value. In most cases `step` is `1`.
choice	[opt1, opt2, …]	string	Range field contains a set of valid strings.
choicei	[n1, n2, …]	64-bit integers	Range field contains a set of valid integers.
string	[min_len, max_len]	string	**Range can be absent.**
position	[width, height]	[x, y] - array of two integer numbers	Range contains 2 numbers that describe width and height of something (an image, for example). x and y are object **bottom left** corner coordinates relative to the **bottom left** image corner.
grid	[width, height]	string	Range holds grid dimensions. Value is a string of `width x height` length which holds grid cell states (`1` for enabled and `0` for disabled) from the **top left** to the **bottom right** corner.
polyline	[[x_min, x_max], [y_min, y_max], max_points]	List of pairs of integers: [[x0, y0], [x1, y1], …, [xN, yN]]	Set of points (2d-coordinates) that are vertexes of non-crossing lines (a polyline). The `max_points` value is a limit of points that idip implementation can support.
polygon	[[x_min, x_max],	List of pairs of	Set of points (2d-coordinates) that are vertexes

Type	Range	Value type	Description
	[y_min, y_max], max_points]	integers: [[x0, y0], [x1, y1], …, [xN, yN]]	of non-crossing edges of polygon. It assumes that first and last points of list are bound to the same edge.
rect	[[x_min, x_max], [y_min, y_max]]	Pair of 2d-coordinates that are top-left and bottom-right vertexes of rectangle: [[x0, y0], [x1, y1]]	A rectangle, that described by 2 diagonal vertexes (e.g. top left and bottom right or top right and bottom left).

A value can be described in JSON as follows:

```
// bool type
{
    "name": "setting/name",
    "value": true,    // or false
    "meta": {
        "type": "bool" // type name
        // The "range" field is absent.
    }
}

// int type
{
    "name": "setting/name",
    "value": 42, // or something else. int64_t
    "meta": {
        "type": "int",  // type name
        "range": [ // Optional. All values are int64_t
            min,   // minimal value (include it)
            max,   // maximum value (include it)
            step   // a distance between neighboring values.
                   // 1 by default
        ]
    }
}

// "string" type
{
    "name": "setting/name",
    "value": "I'm a string", // the value
    "meta": {
        "type": "string", // type name
        "range": [ // Optional. All values are uint32_t
            min_length, // minimal string length (in characters)
                        // 0 by default
            max_length  // maximal string length (in characters)
```

```
                            // 0x7FFFFFFF by default
        ]
    }
}

// choice type
{
    "name": "setting/name",
    "value": "1024x768", // string. One from range
    "meta": {
        "type": "choice", // type name
        // The range is a set of strings which are a possible values
        "range": [
            "1920x1080",
            "1024x768",
            "640x480"
            //, ...
        ]
    }
}

// choicei type (choice of integers)
{
    "name": "setting/name",
    "value": 256, // int64_t
    "meta": {
        "type": "choicei", // type name
        // Range is a set of numbers which can be accepted as a value
        "range": [
            8, 16, 32, 64, 128, 256, 1024, 2048
        ]
    }
}

// grid type
{
    "name": "setting/name",
    // State of grid "switches".
    // Here is a grid example with width = 4 and height = 4.
    // String of lenght 4x3 characters describes cell states as follows:
    //
    //      "0000"
    //      "0110"
    //      "0110"
    //      "0000"
    //
    "value": "0000011001100000",
    "meta": {
        "type": "grid", // type name
        "range": [
            width,  // width of the grid in cells.
            height  // height of the grid in cells.
        ]
    }
}

// position type
{
    "name": "setting/name",
    "value": [ x, y ], // Position from the view bottom left corner, [
       int64_t, int64_t ]
    "meta": {
        "type": "position", // type name
        "range": [
            width,   // width of view in pixels.
                     // value.x >= 0 && value.x < width
            height   // height of view in pixels.
                     // value.y >= 0 && value.y < height
        ]
    }
}

// polyline type
{
    "name": "setting/name",
    // Points on the view [ [ int64_t, int64_t ], ... ]
    "value": [ [x0, y0], [x1, y1], ..., [xK, yK] ],
    "meta": {
        "type": "polyline", // type name
        "range": [
            [x_min, x_max], // X-axis limits
            [y_min, y_max], // Y-axis limits
            max_points      // maximum points in "value" field
        ]
    }
}

// polygon type
{
    "name": "setting/name",
    // Points on the view [ [ int64_t, int64_t ], ... ]
    "value": [ [x0, y0], [x1, y1], ..., [xK, yK] ],
    "meta": {
        "type": "polygon", // type name
        "range": [
            [x_min, x_max], // X-axis limits
            [y_min, y_max], // Y-axis limits
            max_points      // maximum points in "value" field
        ]
    }
}

// rect type
{
    "name": "setting/name",
    // 2 points on the view [ [ int64_t, int64_t ], [ int64_t, int64_t ] ]
    // that are vertexes of a rect.
    "value": [ [x0, y0], [x1, y1] ],
    "meta": {
        "type": "rect", // type name
        "range": [
            [x_min, x_max], // X-axis limits
            [y_min, y_max]  // Y-axis limits
        ]
    }
}
```

9.2.1 Type narrowing

There is an asymmetry between reading and writing cloud configuration values (settings) - a type narrowing. An implementation of the IDIP server must send configuration values with meta-information - a type name and a range. The IDIP server informs the Cloud about each setting type and value limits. Cloud sends only a value without type name and range because they were declared by the IDIP server and therefore cannot be changed. Users of `libidip` should take it into account in `idip_stream_config_update_handler_t` implementations. Inside the handler, you must use correct `idip_cfg_param_get_xxx()` call that matches the value type.

For example, we support a setting named *osd/datetime/format*:

```
{
    // name
    "name": "osd/datetime/format",
    // Type is string. The value is one of strings from "range"
    "value": "YYYY/MM/DD",
    // The meta object (type description)
    "meta": {
        // Type is "choice"
        "type": "choice",
        // "range" is an array of strings
        "range": ["YYYY/MM/DD", "MM-DD-YYYY", "YYYY MM DD"]
    }
}
```

The server sends all the fields: **name**, **value**, **meta.type**, **meta.range**. When user changes the setting, cloud software sends only the updated **value**. In the example above the new **value** type would be a plain string. To get the new value inside the `idip_stream_config_update_handler_t` implementation you should call the `idip_cfg_param_get_string()` function.

We also provide an API which helps to implement the update handler. There are `idip_cfg_meta_xxx()` methods, `IDIP_CFG_META_INIT_XXX` macros, `idip_cfg_xxx_hook_t` callback types, and the `idip_cfg_meta_t` type. They provide an easy way to implement both `idip_stream_config_update_handler_t` and `idip_stream_config_get_handler_t` handlers. We will put references to correspond `IDIP_CFG_META_INIT_XXX` macros during the setting description.

9.3 General settings

Settings that described below will be always present. We describe each setting here in JSON notation. First line mean name of setting.

The *general/online_mode* setting enables or disables all streams from camera/DVR leaving device connected to the Ivideon Cloud. It's works like "mute" signal in amplifiers.

The helper macro is `IDIP_CFG_META_INIT_GENERAL_ONLINE_MODE`.

```
{
    "name": "general/online_mode",
    "value": true, // or false
    "meta": {
        "type": "bool"
    }
}
```

general/timezone. Get or set the current time zone offset.

The helper macro is `IDIP_CFG_META_INIT_GENERAL_TIMEZONE`.

```
{
    "name": "general/timezone",
    "value": "00:00",
    "meta": {
        "type": "choice",
        "range": [
            "-12:00", "-11:00", "-10:00", "-9:00", "00:00", "+01:00"
            //, ...
        ]
    }
}
```

microphone/enabled. Microphone state. Devices that don't have a microphone must return **`false`**. The setting enables audio data sending and sound detection. It's like a "mute" feature in amplifiers.

The helper macro is `IDIP_CFG_META_INIT_GENERAL_MICROPHONE_ENABLED`.

```
{
    "name": "microphone/enabled",
    "value": true, // or false
    "meta": {
        "type": "bool"
    }
}
```

The *led/enabled* setting controls onboard LEDs.

The helper macro is `IDIP_CFG_META_INIT_LED_ENABLED`.

```
{
    "name": "led/enabled",
    "value": true, // or false
    "meta": {
        "type": "bool"
    }
}
```

The *night_vision/mode* setting allows to turn on or off onboard IR lights.

The helper macro is `IDIP_CFG_META_INIT_NIGHT_VISION_MODE`.

```
{
    "name": "night_vision/mode",
    "value": "on",
    "meta": {
        "type": "choice",
        "range": [ "on", "off", "auto" ]
    }
}
```

The *general/power_freq* setting allows to select a frequency of electric power that used for lighting.

The helper macro is `IDIP_CFG_META_INIT_GENERAL_POWER_FREQ`.

```
{
    "name": "general/power_freq",
    "value": "50 Hz",
    "meta": {
        "type": "choice",
        "range": [ "50 Hz", "60 Hz" ]
    }
}
```

9.4 Video and audio

Most of video and audio settings depend on each other. Pay attention that change of some setting will change the value of dependent settings.

The *streams/N/resolution* setting controls image resolution of the `N`-th stream. Here and below `0` is the main video stream, `1` and `2` are substreams of lower and lowest quality, respectively.

The helper macro is `IDIP_CFG_META_INIT_STREAMS_N_RESOLUTION`.

```
{
    "name": "streams/N/resolution",
    "value": "1920x1080",
    "meta": {
        "type": "choice",
        // Camera exports its supported resolutions
        "range": [ "640x480", "1280x720", "1920x1080" ]
    }
}
```

The *streams/N/bitrate* setting controls the `N`-th stream bitrate.

The helper macro is `IDIP_CFG_META_INIT_STREAMS_N_BITRATE`.

```
{
    "name": "streams/N/bitrate",
    "value": 2048,
    "meta": {
        "type": "choicei",
        // Camera should exports actual list of supported
        // bitrates (in kilobytes per second).
        "range": [ 128, 256, 512, 1024, 2048 ]
    }
}
```

The *streams/N/fps* setting controls the `N`-th stream FPS. Range is for example here.

The helper macro is `IDIP_CFG_META_INIT_STREAMS_N_FPS`.

```
{
    "name": "streams/N/fps",
    "value": 30,
    "meta": {
        "type": "choicei",
        // Camera should exports actual list of supported
        // frame rates.
        "range": [ 1, 5, 10, 15, 20, 24, 25, 30, 60 ]
    }
}
```

The *streams/N/codec* setting allows to set up a type of video encoder for the `N`-th stream. **range** should contain the list of codecs that is supported both by the `libidip` and the camera.

The helper macro is `IDIP_CFG_META_INIT_STREAMS_N_VCODEC`.

```
{
    "name": "streams/N/codec",
    "value": "H264",
    "meta": {
        "type": "choice",
        "range": [ "mjpeg", "H264", "H265", "MPEG4" ]
    }
}
```

The *streams/N/acodec* setting allows to set up a type of audio encoder for the `N`-th stream. **range** should contain the list of codecs that is supported both by the `libidip` and the camera.

The helper macro is `IDIP_CFG_META_INIT_STREAMS_N_ACODEC`.

```
{
    "name": "streams/N/acodec",
    "value": "AAC",
    "meta": {
        "type": "choice",
        "range": [ "PCMA", "PCMU", "AAC", "MP3" ]
    }
}
```

9.5 Image settings

The setting *image/upside_down* allows to set up a image rotation for 180 degrrees. It's equals to flip and mirrot transformations.

The helper macro is `IDIP_CFG_META_INIT_IMAGE_UPSIDE_DOWN`.

```
{
    "name": "image/upside_down",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

The setting *image/rotate* applies 90-degrees clockwise image rotation to the image.

The helper macro is `IDIP_CFG_META_INIT_IMAGE_ROTATE`.

```
{
    "name": "image/rotate",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

NOTE: combination of *image/upside_down* and *image/rotate* settings gives all possible image transformations.

The setting *image/brightness* allows to set up live video image brightness in percents.

The helper macro is `IDIP_CFG_META_INIT_IMAGE_BRIGHTNESS`.

```
{
    "name": "image/brightness",
    "value": 50,
    "meta": {
        "type": "int",
        "range": [0, 100]
    }
}
```

The setting *image/contrast* allows to set up live video image contrast in percents.

The helper macro is `IDIP_CFG_META_INIT_IMAGE_CONTRAST`.

```
{
    "name": "image/contrast",
    "value": 50,
    "meta": {
        "type": "int",
        "range": [0, 100]
    }
}
```

The setting *image/saturation* allows to set up live video image saturation in percents.

The helper macro is `IDIP_CFG_META_INIT_IMAGE_SATURATION`.

```
{
    "name": "image/saturation",
    "value": 50,
    "meta": {
        "type": "int",
        "range": [0, 100]
    }
}
```

The setting *image/hue* allows to set up live video image hue in percents.

The helper macro is `IDIP_CFG_META_INIT_IMAGE_HUE`.

```
{
    "name": "image/hue",
    "value": 50,
    "meta": {
        "type": "int",
        "range": [0, 100]
    }
}
```

The setting *image/wdr* enables or disables WideDynamicRange feauture of image contrast compensation.

The helper macro is `IDIP_CFG_META_INIT_IMAGE_WDR`.

```
{
    "name": "image/wdr",
    "value": false, // or true
    "meta": {
        "type": "bool"
    }
}
```

The last image setting, *image/wdr_level*, allows to setup image contrast compensation level. It applies when the *image/wdr* is enabled.

The helper macro is `IDIP_CFG_META_INIT_IMAGE_WDR_LEVEL`.

```
{
    "name": "image/wrd_level",
    "value": 50,
    "meta": {
        "type": "int",
        "range": [0, 100]
    }
}
```

9.6 OSD

On-Screen Display (OSD) is an overlay where text messages are shown over a video stream. This section describes OSD-related settings.

osd/transparent. This setting is optional. It controls OSD transparency.

The helper macro is `IDIP_CFG_META_INIT_OSD_TRANSPARENT`.

```
{
    "name": "osd/transparent",
    "value": true,
    "meta": {
        "type": "bool"
    }
}
```

osd/flashing. It is also optional. If set, OSD must be flashing.

The helper macro is `IDIP_CFG_META_INIT_OSD_FLASHING`.

```
{
    "name": "osd/flashing",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

The *osd/datetime/enabled* setting controls the date and time displaying.

The helper macro is `IDIP_CFG_META_INIT_OSD_DATETIME_ENABLED`.

```
{
    "name": "osd/datetime/enabled",
    "value": true,
    "meta": {
        "type": "bool"
    }
}
```

The next setting controls the week number and the week name displaying.

The helper macro is `IDIP_CFG_META_INIT_OSD_DATETIME_DISPLAY_WEEK`.

```
{
    "name": "osd/datetime/display_week",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

osd/datetime/24h. If set, then on-screen time is displayed in 24-hours format. Otherwise, 12-hour format is used. The setting is optional.

The helper macro is `IDIP_CFG_META_INIT_OSD_DATETIME_24H`.

```
{
    "name": "osd/datetime/24h",
    "value": true,
    "meta": {
        "type": "bool"
    }
}
```

The *osd/datetime/format* setting controls the date format. Optional.

The helper macro is `IDIP_CFG_META_INIT_OSD_DATETIME_FORMAT`.

```
{
    "name": "osd/datetime/format",
    "value": "MM-DD-YYYY",
    "meta": {
        "type": "choice",
        "range": [ "YYYY/MM/DD", "MM-DD-YYYY", "YYYY MM DD" ]
    }
}
```

The next setting allows to set up the datetime frame position.

The helper macro is `IDIP_CFG_META_INIT_OSD_DATETIME_POS`.

```
{
    "name": "osd/datetime/pos",
    "value": [ 50, 100 ], // [ x, y ]
    "meta": {
        "type": "position",
        "range": [ 1024, 768 ] // [ width, height ]
    }
}
```

The *osd/name/enabled* setting enables or disables the additional string displaying. This line is usually used to show camera name.

The helper macro is `IDIP_CFG_META_INIT_OSD_NAME_ENABLED`.

```
{
    "name": "osd/name/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

To set up a text message to *name* string use the *osd/name/text* setting.

The helper macro is `IDIP_CFG_META_INIT_OSD_NAME_TEXT`.

```
{
    "name": "osd/name/text",
    "value": "Camera 01",
    "meta": {
        "type": "string"
        // range is optional
    }
}
```

Text position on the OSD is set up using the *osd/name/pos* setting.

The helper macro is `IDIP_CFG_META_INIT_OSD_NAME_POS`.

```
{
    "name": "osd/name/pos",
    "value": [ 50, 700 ], // [ x, y ]
    "meta": {
        "type": "position",
        "range": [ 1024, 768 ] // [ width, height ]
    }
}
```

Device can support up to 5 (`N = 0..4`) additional strings for the OSD. Next two settings are used to set up their text and position. Empty strings ("") aren't displayed.

The helper macro is `IDIP_CFG_META_INIT_OSD_CUSTOM_N_TEXT`.

```
{
    "name": "osd/custom/N/text",
    "value": "Camera 01",
    "meta": {
        "type": "string"
        // range is optional
    }
}
```

The helper macro for *osd/custom/N/pos* is `IDIP_CFG_META_INIT_OSD_CUSTOM_N_POS`.

```
{
    "name": "osd/custom/N/pos",
    "value": [ 700, 50 ], // [ x, y ]
    "meta": {
        "type": "position",
        "range": [ 1024, 768 ] // [ width, height ]
    }
}
```

9.7 Detectors and events

This section contains settings for detectors. Not of all settings can be supported by device. Settings are grouped by type and purpose. Each group sets a count limit `N`. For example, a device supports maximum 4 visual areas for tampering detection. By default, they are disabled. To refer to all of tamper detection settings we use notation *tampering_detection/N/*, where `0 <= N <= 3`.

To describe visual areas we use polygons and rectangles. Devices that don't support custom polygons **must** send `rect` as a type.

9.7.1 Common detectors

9.7.1.1 Sound detection settings

To enable or disable sound detector set the *sound_detection/enabled* value. The setting depends on the *microphone/enabled* setting: if the microphone was muted by *microphone/enabled* then the setting is ignored.

The helper macro is `IDIP_CFG_META_INIT_SOUND_DETECTION_ENABLED`.

```
{
    "name": "sound_detection/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

The *sound_detection/sensitivity* value sets up a sound detector sensitivity level in percent. Range is always `[0, 100]`.

Helper macros:

```
IDIP_CFG_META_INIT_SOUND_DETECTION_SENSITIVITY        // for fine tuning
IDIP_CFG_META_INIT_SOUND_DETECTION_SENSITIVITY_20EACH // for tuning with
   20% step
```

Description:

```
{
    "name": "sound_detection/sensitivity",
    "value": 50,
    "meta": {
        "type": "int",
        "range": [0, 100]
    }
}
```

9.7.1.2 Sound detection events

When *microphone/enabled* and *sound_detection/enabled* are true, implementation can send a sound detection events, which have no fields except the *name* set to `sound`:

```
{
    "name": "sound"
}
```

To send the event you may use the next lines of code:

```
int rc = idip_server_send_event(server,
            "MyTarget",// target name is an example
            "sound",   // event name
            now_ms,    // event time stamp (milliseconds UTC)
            NULL);     // there is no data bound to the event
if( rc != IDIP_OK ) {
    // Handle the error
}
```

Alternatively we provide a wrapper named `idip_server_send_sound_event()`. It does the same but has fewer arguments:

```
int rc = idip_server_send_sound_event(server,
            "MyTarget",// target name is an example
            now_ms);   // event time stamp (milliseconds UTC)
if( rc != IDIP_OK ) {
    // Handle the error
}
```

9.7.1.3 PIR detection settings The *pir_detection/enabled* setting controls the onboard PIR-detector separately from motion detectors.

The helper macro is `IDIP_CFG_META_INIT_PIR_DETECTION_ENABLED`.

```
{
    "name": "pir_detection/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

9.7.1.4 PIR detection events PIR detector should produce simple events with the *name* field set to `pir`.

```
{
    "name": "pir"
}
```

You may use the `idip_server_send_pir_event()` call to send the event.

9.7.1.5 Motion detection settings Camera implementation can support one or more motion detectors. `N` is a numeric identifier of motion detector, `K` total number of motion detectors, so `0 <= N < K`.

The *motion_detection/enabled* setting enables or disables motion detector algorithm.

The helper macro is `IDIP_CFG_META_INIT_MOTION_DETECTION_ENABLED`.

```
{
    "name": "motion_detector/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

The *motion_detector/N/sensitivity* setting can be set in three forms.

The helper macro is `IDIP_CFG_META_INIT_MOTION_DETECTION_SENSITIVITY`.

```
// The first (main) form.
{
    "name": "motion_detector/N/sensitivity",
    "value": 40,
    "meta": {
        "type": "int",
        "range": [0, 100, 20]
    }
}
// The second form. It is logically equivalent to the first one.
{
    "name": "motion_detector/N/sensitivity",
    "value": 40,
    "meta": {
        "type": "choicei",
        "range": [0, 20, 40, 60, 80, 100]
    }
}
// The third form. It allows user to setup sensitivity level using related
    words.
{
    "name": "motion_detector/N/sensitivity",
    "value": "medium",
    "meta": {
        "type": "choice",
        "range": [ "very low", "low", "medium", "hight", "very high" ]
    }
}
```

The *motion_detector/N/threshold* setting controls the `N`-th motion detector threshold value.

The helper macro is `IDIP_CFG_META_INIT_MOTION_DETECTION_THRESHOLD`.

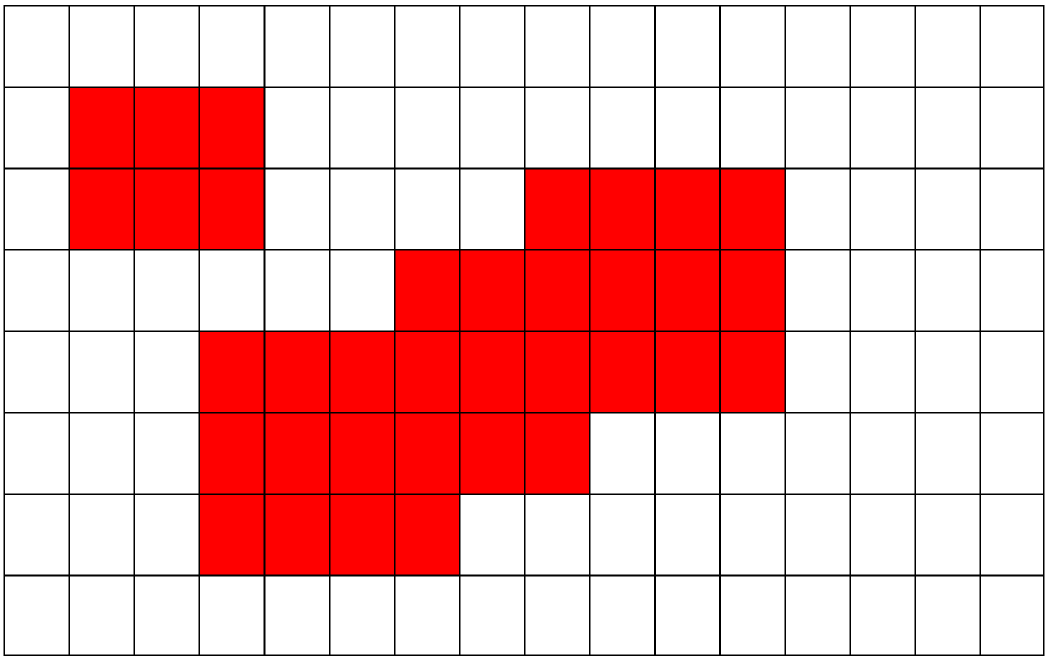
```
{
    "name": "motion_detector/N/threshold",
    "value": 40,
    "meta": {
        "type": "int",
        "range": [0, 100, 20]
    }
}
```

The last setting of the group is *motion_detector/N/zones*. It allows to setup motion detection zones. A field of view is divided into rectangular cells (e.g. `16x8`) and detection in each cell is enabled separately via the `grid`-type value that looks like a string with ‘`0`’ and ‘`1`’ characters and acts like a ‘mask’. The example below contains an example of the `grid`-type setting. The `N`-th zone could contain non-intersecting areas.

The helper macro is `IDIP_CFG_META_INIT_MOTION_DETECTION_ZONES`.

```
{
    "name": "motion_detector/N/zones",
    // Because the type is a "grid"
    // the value type is a string that contains
    // width*height (16x8) characters of { 0, 1 }.
    // We divide the example string to parts to
    // show how zone mask works.
    // Real string should not contain anything other than 0 or 1
        characters.
    //         0123456789ABCDEF
    "value": "0000000000000000   // 0
              0111000000000000   // 1
              0111000011110000   // 2
              0000001111110000   // 3
              0001111111110000   // 4
              0001111110000000   // 5
              0001111000000000   // 6
              0000000000000000"  // 7
    "meta": {
        "type": "grid",
        // Number of ceils for X and Y axis
        "range": [16, 8]
    }
}
```

The mask on the image contains 2 non-intersecting areas. Device can support a few number of masks.



9.7.1.6 Motion detection events When motion detector is enabled `motion` events can be generated by the device. A zone number is optional.

```
{
    "name": "motion",
    "data": {
        "zone": 3 // zone number
    }
}
```

There are 2 methods that allows you to send the event:

- `idip_server_send_motion_event()` and
- `idip_server_send_motion_zone_event()`.

9.7.2 Another events

Here is a set of event types that are also recognized by the Ivideon Cloud.

Alarm event. Device has to send this event type in general alarm cases. The event has no arguments. To send these events you may use the `idip_server_send_alarm_event()` call.

```
{
   "name": "alarm"
}
```

Call event on a call button (e.g., a door bell) pressing. The event has no arguments. A call named `idip_server_send_call_event()` may help you to send the event.

```
{
   "name": "call"
}
```

9.7.3 Privacy Mask

Privacy mask is an area on the camera view which can contain private information, e.g. bank card pin code. Therefore, the area is removed from view by filling with a solid color.

9.7.3.1 Privacy Mask Settings

The *privacy_mask/N/enabled* setting enables or disables the `N`-th privacy mask.

The helper macro is `IDIP_CFG_META_INIT_PRIVACY_MASK_N_ENABLED`.

```
{
    "name": "privacy_mask/N/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

The *privacy_mask/N/mask* setting allows to set up the area to fill. If the device does not support polygons, **type** must be `rect`.

Helper macros:

```
IDIP_CFG_META_INIT_PRIVACY_MASK_N_MASK      // for polygon
IDIP_CFG_META_INIT_PRIVACY_MASK_N_MASK_RECT // for rect
```

Description:

```
{
    "name": "privacy_mask/N/mask",
    // Current polygon contains 4 segments
    "value": [ [100, 100], [600, 100], [600, 400], [100, 400] ],
    "meta": {
        "type": "polygon",
        "range": [
            [0, 7680], // [ min_x, max_x ]
            [0, 4320], // [ min_y, max_y ]
            16         // the polygon can contain maximum of 16 points
        ]
    }
}
```

9.7.4 Line Detector

Line detector is defined by `N` polylines. When an object crosses the polyline in the specified direction, the device sends `vca.line_detection.crossed` event. Each of `N` polylines consists of `K` segments, where `K > 0`.

By default all of `N` detectors are in disabled state. Their segments are not visible.

9.7.4.1 Line Detector Settings

The *line_detection/enabled* setting enables or disables line crossing detection algorithm for all segmennts.

The helper macro is `IDIP_CFG_META_INIT_LINE_DETECTION_ENABLED`.

```
{
    "name": "line_detection/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

The *line_detection/N/enabled* setting enables or disables the `N`-th line detector.

The helper macro is `IDIP_CFG_META_INIT_LINE_DETECTION_N_ENABLED`.

```
{
    "name": "line_detection/N/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

line_detection/N/segments. Get or set a line for the `N`-th detector.

The helper macro is `IDIP_CFG_META_INIT_LINE_DETECTION_N_SEGMENTS`.

```
{
    "name": "line_detection/N/segments",
    // Current line consist of 2 segments (with 3 points)
    "value": [ [10, 10], [500, 500], [500, 700] ],
    "meta": {
        "type": "polyline",
        "range": [
            [0, 7680], // [ min_x, max_x ]
            [0, 4320], // [ min_y, max_y ]
            16         // line can contain maximum of 16 points
        ]
    }
}
```

line_detection/N/direction. Get or set the crossing direction to detect for the `N`-th line detector. Optional setting.

The helper macro is `IDIP_CFG_META_INIT_LINE_DETECTION_N_DIRECTION`.

```
{
    "name": "line_detection/N/direction",
    "value": "any",
    "meta": {
        "type": "choice",
        "range": [ "right", "left", "any" ]
    }
}
```

where `right`, `left` and `any` correspond the “ONVIF™ Video Analytics Service Specification, Ver. 2.2, Annex A.1” definitions.

The *line_detection/N/sensitivity* setting sets up the `N`-th line detector sensitivity in percents.

The helper macro is `IDIP_CFG_META_INIT_LINE_DETECTION_N_SENSITIVITY`.

```
{
    "name": "line_detection/N/sensitivity",
    "value": 50,
    "meta": {
        "type": "int",
        "range": [ 0, 100, 1 ]
    }
}
```

9.7.4.2 Line Detector Events To send an event that describes a line crossing, use the `idip_server_send_vca_line_detection_crossed_event()` call.

```
{
    "name": "vca.line_detection.crossed",
    "data": {
        "line": 0, // a number of detector (0 <= line && line < N)
        "direction": "left" // optional
    }
}
```

9.7.5 Field Detector

The field detector determines if some object is inside a specified polygon.

9.7.5.1 Field Detector Settings The *field_detection/enabled* setting enables or disables field detection algorithm for all detection fields.

The helper macro is `IDIP_CFG_META_INIT_FIELD_DETECTION_ENABLED`.

```
{
    "name": "field_detection/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

The *field_detection/N/enabled* setting enables or disables the `N`-th detector.

The helper macro is `IDIP_CFG_META_INIT_FIELD_DETECTION_N_ENABLED`.

```
{
    "name": "field_detection/N/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

The *field_detection/N/field* setting gives an ability to set up a field detection area. If the device does not support polygons, the type must be `rect`.

Helper macros:

```
IDIP_CFG_META_INIT_FIELD_DETECTION_N_FIELD      // for polygon
IDIP_CFG_META_INIT_FIELD_DETECTION_N_FIELD_RECT // for rect
```

Description:

```
{
    "name": "field_detection/N/field",
    // Current polygon contains 4 segments
    "value": [ [100, 100], [600, 100], [600, 400], [100, 400] ],
    "meta": {
        "type": "polygon",
        "range": [
            [0, 7680], // [ min_x, max_x ]
            [0, 4320], // [ min_y, max_y ]
            16         // the polygon can contain maximum of 16 points
        ]
    }
}
```

The *field_detection/N/sensitivity* setting sets up detector sensitivity in percents. The helper macros are:

```
IDIP_CFG_META_INIT_FIELD_DETECTION_N_SENSITIVITY        // for fine tuning
IDIP_CFG_META_INIT_FIELD_DETECTION_N_SENSITIVITY_20EACH // for tuning with
    20% step
```

```
{
    "name": "field_detection/N/sensitivity",
    "value": 50,
    "meta": {
        "type": "int",
        "range": [ 0, 100, 1 ]
    }
}
```

The *field_detection/N/time_interval* setting allows to set up a delay between the event notifications in milliseconds.

The helper macro is `IDIP_CFG_META_INIT_FIELD_DETECTION_N_TIME_INTERVAL`.

```
{
    "name": "field_detection/N/time_interval",
    "value": 5000, // 5 sec
    "meta": {
        "type": "int",
        "range": [ 0, 100000000, 1 ]
    }
}
```

9.7.5.2 Field Detector Events If enabled, implementation should send the detection event periodically, while the given object is inside the polygon. You may use the `idip_server_send_vca_field_detection_ins` `()` call to send the event.

```
// [periodic]
{
    "name": "vca.field_detection.inside",
    "data": {
        "field": 2 // a number of field detector
    }
}
```

9.7.6 Loitering Detector

A loitering detector is defined by a simple non-intersecting polygon as an area of interest and threshold loitering interval.

The detector determines if some object in the scene inside the polygon longer than the given time threshold value.

9.7.6.1 Loitering Detector Settings The *loitering_detection/enabled* setting enables or disables loitering detection algorithm for all detection fields.

The helper macro is `IDIP_CFG_META_INIT_LOITERING_DETECTION_ENABLED`.

```
{
    "name": "loitering_detection/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

The *loitering_detection/N/enabled* setting enables or disables the `N`-th detector.

The helper macro is `IDIP_CFG_META_INIT_LOITERING_DETECTION_N_ENABLED`.

```
{
    "name": "loitering_detection/N/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

loitering_detection/N/field. Detection area. If the device does not support polygons, the points in value type must be `rect`. In the example is shown the case when a camera don't support polygons.

Helper macros:

```
IDIP_CFG_META_INIT_LOITERING_DETECTION_N_FIELD      // for polygon
IDIP_CFG_META_INIT_LOITERING_DETECTION_N_FIELD_RECT // for rect
```

Description:

```
{
    "name": "loitering_detection/N/field",
    // A rectangle by 2 points
    "value": [ [100, 100], [600, 400] ],
    "meta": {
        "type": "rect",
        "range": [
            [0, 7680], // [ min_x, max_x ]
            [0, 4320]  // [ min_y, max_y ]
        ]
    }
}
```

loitering_detection/N/time_threshold. Setup a minimum time interval which object should be present in the area to start detection. Time is in milliseconds.

The helper macro is `IDIP_CFG_META_INIT_LOITERING_DETECTION_N_TIME_THRESHOLD`.

```
{
    "name": "loitering_detection/N/time_threshold",
    "value": 5000,
    "meta": {
        "type": "int"
    }
}
```

loitering_detection/N/sensitivity. Detector sensitivity in percents. Optional.

Helper macros:

```
IDIP_CFG_META_INIT_LOITERING_DETECTION_N_SENSITIVITY        // for fine
   tuning
IDIP_CFG_META_INIT_LOITERING_DETECTION_N_SENSITIVITY_20EACH // for tuning
   with 20% step
```

Description:

```
{
    "name": "loitering_detection/N/sensitivity",
    "value": 50,
    "meta": {
        "type": "int",
        "range": [ 0, 100, 1 ]
    }
}
```

loitering_detection/N/is_periodic_notify. If an object is in the detection area for a long time then the detector sends loitering events periodically. *loitering_detection/N/is_periodic_notify* allows to enable or disable this behavior.

The helper macro is `IDIP_CFG_META_INIT_LOITERING_DETECTION_N_IS_PERIODIC_NOTIFY`
.

```
{
    "name": "loitering_detection/N/is_periodic_notify",
    "value": true,
    "meta": {
        "type": "bool"
    }
}
```

loitering_detection/N/time_interval. If the object does continue to be inside the area then the detector should periodically send loitering events. The setting defines an interval between messages (in milliseconds).

The helper macro is `IDIP_CFG_META_INIT_LOITERING_DETECTION_N_TIME_INTERVAL`.

```
{
    "name": "loitering_detection/N/time_interval",
    "value": 5000,
    "meta": {
        "type": "int"
    }
}
```

9.7.6.2 Loitering Detector Events

A loitering detector should send events with the *name* set to `vca.loitering_detection.loitering`. You may use the `idip_server_send_vca_loitering_detector_event()` call to send the event.

```
// [periodic]
{
    "name": "vca.loitering_detection.loitering",
    "data": {
        "field": 5 // int
    }
}
```

9.7.7 Crowd Detector

A crowd detector is defined by a simple non-intersecting polygon as a region of interest. The detector triggers a `vca.crowd_detection.crowded` event either when the number of objects inside the polygon exceeds the `num_of_objects_threshold`, or the polygon around these objects overlaps a specified area by more than `overlap_area_threshold` (in percents). If the device does not support polygons, the points in the polygon must represent a rectangle.

9.7.7.1 Crowd Detector Settings

The *crowd_detection/enabled* setting enables or disables crowd detection algorithm for all detection fields.

The helper macro is `IDIP_CFG_META_INIT_CROWD_DETECTION_ENABLED`.

```
{
    "name": "crowd_detection/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

The *crowd_detection/N/enabled* setting enables or disables the `N`-th crowd detector.

The helper macro is `IDIP_CFG_META_INIT_CROWD_DETECTION_N_ENABLED`.

```
{
    "name": "crowd_detection/N/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

crowd_detection/N/field. Detection area. If the device does not support polygons, the type must be `rect`.

Helper macros:

```
IDIP_CFG_META_INIT_CROWD_DETECTION_N_FIELD      // for polygon
IDIP_CFG_META_INIT_CROWD_DETECTION_N_FIELD_RECT // for rect
```

Desciprtion:

```
{
    "name": "crowd_detection/N/field",
    "value": [ [100, 100], [600, 100], [600, 400], [100, 400] ],
    "meta": {
        "type": "polygon",
        "range": [
            [0, 7680], // [ min_x, max_x ]
            [0, 4320], // [ min_y, max_y ]
            16         //
        ]
    }
}
```

The *crowd_detection/N/overlap_area_threshold* setting allows to set up `N`-th crowd detector sensitivity. The setting is optional.

The helper macro is `IDIP_CFG_META_INIT_CROWD_DETECTION_N_OVERLAP_AREA_THRESHOLD`.

```
// [optional]
{
    "name": "crowd_detection/N/overlap_area_threshold",
    "value": 50,
    "meta": {
        "type": "int",
        "range": [ 0, 100 ]
    }
}
```

crowd_detection/N/num_of_objects_threshold. The next setting allows to setup a triggering level for the `N`-th crowd detector.

The helper macro is `IDIP_CFG_META_INIT_CROWD_DETECTION_N_NUM_OF_OBJECTS_THRESHOLD`.

```
// [optional]
{
    "name": "crowd_detection/N/num_of_objects_threshold",
    "value": 10,
    "meta": {
        "type": "int"
    }
}
```

9.7.7.2 Crowd Detector Events

```
// [periodic]
{
    "name": "vca.crowd_detection.crowded",
    "data": {
        "field": 0, // number of detector
        "num_of_objects": 42 // optional
    }
}
```

9.7.8 Tampering Detection

Tampering detector recognizes any kind of tampering to the image sensor. The `MODE` parameter defines the tampering type: *global_scene_change*, *image_too_dark*, *image_to_bright*, *signal_loss* and *image_too_blurry*.

9.7.8.1 Tampering Detector Settings The *tampering_detection/enabled* setting enables or disables tampering detection algorithm for all detection fields.

The helper macro is `IDIP_CFG_META_INIT_TAMPERING_DETECTION_ENABLED`.

```
{
    "name": "tampering_detection/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

The setting is enables or disables N-th tampering detection area.

The helper macro is `IDIP_CFG_META_INIT_TAMPERING_DETECTION_N_ENABLED`.

```
{
    "name": "tampering_detection/N/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

The *tampering_detection/N/threshold* setting allows to set up tampering detection threshold for N-th detection area in percents.

The helper macro is `IDIP_CFG_META_INIT_TAMPERING_DETECTION_N_THRESHOLD`.

```
// [optional]
{
    "name": "tampering_detection/N/threshold",
    "value": 50,
    "meta": {
        "type": "int",
        "range": [ 0, 100 ]
    }
}
```

The *tampering_detection/N/field* setting allows to set up a tampering N-th detection area. If the device does not support polygons, the type must be `rect`.

Helper macros:

```
IDIP_CFG_META_INIT_TAMPERING_DETECTION_N_FIELD      // for polygon
IDIP_CFG_META_INIT_TAMPERING_DETECTION_N_FIELD_RECT // for rect
```

Description:

```
// [optional]
{
    "name": "tampering_detection/N/field",
    "value": [ [100, 100], [600, 100], [600, 400], [100, 400] ],
    "meta": {
        "type": "polygon",
        "range": [
            [0, 7680], // [ min_x, max_x ]
            [0, 4320], // [ min_y, max_y ]
            16         //
        ]
    }
}
```

tampering_detection/N/{MODE}/enabled. A group of settings that allows to enable or disable `{MODE}` kind of N-th tampering detection and setup the threshold and detection time interval for the `N`-th detector.

```
{
    "name": "tampering_detection/N/{MODE}/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}

{
    "name": "tampering_detection/N/{MODE}/threshold",
    "value": 50,
    "meta": {
        "type": "int",
        "range": [0, 100]
    }
}

{
    "name": "tampering_detection/N/{MODE}/duration",
    "value": 5000,
    "meta": {
        "type": "int"
    }
}
```

Helper macros:

```
IDIP_CFG_META_INIT_TAMPERING_DETECTION_N_GLOBAL_SCENE_CHANGE_ENABLED,
IDIP_CFG_META_INIT_TAMPERING_DETECTION_N_GLOBAL_SCENE_CHANGE_THRESHOLD,
IDIP_CFG_META_INIT_TAMPERING_DETECTION_N_GLOBAL_SCENE_CHANGE_DURATION,
IDIP_CFG_META_INIT_TAMPERING_DETECTION_N_IMAGE_TOO_DARK_ENABLED,
IDIP_CFG_META_INIT_TAMPERING_DETECTION_N_IMAGE_TOO_DARK_THRESHOLD,
IDIP_CFG_META_INIT_TAMPERING_DETECTION_N_IMAGE_TOO_DARK_DURATION,
IDIP_CFG_META_INIT_TAMPERING_DETECTION_N_IMAGE_TOO_BRIGHT_ENABLED,
IDIP_CFG_META_INIT_TAMPERING_DETECTION_N_IMAGE_TOO_BRIGHT_THRESHOLD,
IDIP_CFG_META_INIT_TAMPERING_DETECTION_N_IMAGE_TOO_BRIGHT_DURATION,
IDIP_CFG_META_INIT_TAMPERING_DETECTION_N_SIGNAL_LOSS_ENABLED,
IDIP_CFG_META_INIT_TAMPERING_DETECTION_N_SIGNAL_LOSS_THRESHOLD,
IDIP_CFG_META_INIT_TAMPERING_DETECTION_N_SIGNAL_LOSS_DURATION,
IDIP_CFG_META_INIT_TAMPERING_DETECTION_N_IMAGE_TOO_BLURRY_ENABLED,
IDIP_CFG_META_INIT_TAMPERING_DETECTION_N_IMAGE_TOO_BLURRY_THRESHOLD,
IDIP_CFG_META_INIT_TAMPERING_DETECTION_N_IMAGE_TOO_BLURRY_DURATION
```

9.7.8.2 Tampering Detector Events

```
{
    "name": "vca.tampering_detection.{MODE}"
}
```

You may send these event types using these calls:

- `idip_server_send_vca_tampering_detection_global_scene_change_event();`
- `idip_server_send_vca_tampering_detection_image_too_dark_event();`
- `idip_server_send_vca_tampering_detection_image_too_bright_event();`
- `idip_server_send_vca_tampering_detection_signal_loss_event()`.

9.7.9 Face Detection

Camera analyzes video stream and triggers `vca.face_detection.face` event when it detects human faces.

9.7.9.1 Face Detection Settings

The *face_detection/enabled* setting enables or disables face detection.

The helper macro is `IDIP_CFG_META_INIT_FACE_DETECTION_ENABLED`.

```
{
    "name": "face_detection/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

The *face_detection/field* setting allows to set up a face detection area. If the device does not support polygons, the type should be `rect`.

Helper macros:

```
IDIP_CFG_META_INIT_FACE_DETECTION_FIELD      // for polygon
IDIP_CFG_META_INIT_FACE_DETECTION_FIELD_RECT // for rect
```

Description:

```
{
    "name": "face_detection/field",
    "value": [ [100, 100], [600, 100], [600, 400], [100, 400] ],
    "meta": {
        "type": "polygon",
        "range": [
            [0, 7680], // [ min_x, max_x ]
            [0, 4320], // [ min_y, max_y ]
            16
        ]
    }
}
```

Additional related settings with self-explanatory names:

- *face_detection/min_face_size*:

  ```
  // [optional]
  {
      "name": "face_detection/min_face_size",
      "value": 10,
      "meta": {
          "type": "int",
          "range": [ 1, 100 ]
      }
  }
  ```

  The helper macro is `IDIP_CFG_META_INIT_FACE_DETECTION_MIN_FACE_SIZE`.

- *face_detection/max_face_size*:

  ```
  // [optional]
  {
      "name": "face_detection/max_face_size",
      "value": 1024,
      "meta": {
          "type": "int",
          "range": [ 1, 100 ]
      }
  }
  ```

  The helper macro is `IDIP_CFG_META_INIT_FACE_DETECTION_MAX_FACE_SIZE`.

- *face_detection/min_face_size_pixels*:

  ```
  // [optional]
  {
      "name": "face_detection/min_face_size_pixels",
      "value": 64,
      "meta": {
          "type": "int",
          "range": [ 16, 10000 ]
      }
  }
  ```

  The helper macro is `IDIP_CFG_META_INIT_FACE_DETECTION_MIN_FACE_SIZE_PIXELS`.

- *face_detection/max_face_size_pixels*:

```
// [optional]
{
    "name": "face_detection/max_face_size_pixels",
    "value": 1024,
    "meta": {
        "type": "int",
        "range": [ 16, 10000 ]
    }
}
```

The helper macro is `IDIP_CFG_META_INIT_FACE_DETECTION_MAX_FACE_SIZE_PIXELS`.

- *face_detection/sensitivity*:

```
// [optional]
{
    "name": "face_detection/sensitivity",
    "value": 50,
    "meta": {
        "type": "int",
        "range": [ 0, 100 ]
    }
}
```

The helper macro is `IDIP_CFG_META_INIT_FACE_DETECTION_SENSITIVITY`.

- *face_detection/blur_threshold*:

```
// [optional]
{
    "name": "face_detection/blur_threshold",
    "value": 50,
    "meta": {
        "type": "int",
        "range": [ 0, 100 ]
    }
}
```

The helper macro is `IDIP_CFG_META_INIT_FACE_DETECTION_BLUR_THRESHOLD`.

- *face_detection/contrast_threshold*:

```
// [optional]
{
    "name": "face_detection/contrast_threshold",
    "value": 50,
    "meta": {
        "type": "int",
        "range": [ 0, 100 ]
    }
}
```

The helper macro is `IDIP_CFG_META_INIT_FACE_DETECTION_CONTRAST_THRESHOLD`.

9.7.9.2 Face Detection Events

```
{
    "name": "vca.face_detection.face",
    "data": {
        "track_begin": 10, // int
        "track_end": 25,   // int
        "best_shot": {
            "ts": 1566381974000, // int, milliseconds in UTC
            "data": BLOB,   // face crop in JPEG format
            "width": 100,   // int. In pixels. Optional.
            "height": 100, // int. In pixels. Optional.
            "position": {   // optional
                "x": 300,   // int
                "y": 400,   // int
                "width": 100, // int
                "height": 100 // int
            }
        }
    }
}
```

`libidip` provides a helper method `idip_server_send_vca_face_detection_face_event()` to send face detection events. You may use it instead the generic `idip_server_send_event()` call.

9.7.10 ROI

Region-Of-Interest is an area on camera view which contains more important details than the rest of the view. For example, bottom area of street view contains informative part (car plate numbers) whereas top area doesn't (sky, trees, buildings). Areas that marked as ROI should be encoded with better quality than the others, so the bandwidth is used mostly for important parts of the image.

By default, all of `N` regions are disabled, all regions are empty.

9.7.10.1 ROI Settings The *roi/enabled* setting enables or disables all regions.

The helper macro is `IDIP_CFG_META_INIT_ROI_ENABLED`.

```
{
    "name": "roi/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

The *roi/N/enabled* setting enables or disables the `N`-th region of interest.

The helper macro is `IDIP_CFG_META_INIT_ROI_N_ENABLED`.

```
{
    "name": "roi/N/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

To set up a ROI area on the camera view the *roi/N/roi* setting is using. If the device does not support polygons, the value type must be `rect`.

Helper macros:

```
IDIP_CFG_META_INIT_ROI_N_FIELD      // for polygon
IDIP_CFG_META_INIT_ROI_N_FIELD_RECT // for rect
```

Descipritіon:

```
{
    "name": "roi/N/roi",
    // Current polygon contains 4 segments
    "value": [ [100, 100], [600, 100], [600, 400], [100, 400] ],
    "meta": {
        "type": "polygon",
        "range": [
            [0, 7680], // [ min_x, max_x ]
            [0, 4320], // [ min_y, max_y ]
            16         // the polygon can contain maximum of 16 points
        ]
    }
}
```

9.7.11 Object removal detection

The Object removal detection is a tracking feature that continuously tracks objects in its FoV and notifies the Cloud about their disappearance.

9.7.11.1 Object removal detection settings

The *removal_object_detection/enabled* setting enables or disables removal object detection algorithm for all detection fields.

The helper macro is `IDIP_CFG_META_INIT_REMOVAL_OBJECT_DETECTION_ENABLED`.

```
{
    "name": "removal_object_detection/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

The *removal_object_detection/N/enabled* setting allows to enable or disable object removal detection for the `N`-th detector.

The helper macro is `IDIP_CFG_META_INIT_REMOVAL_OBJECT_DETECTION_N_ENABLED`.

```
{
    "name": "removal_object_detection/N/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

The *removal_object_detection/N/sensitivity* setting allows to set up sensitivity of the `N`-th detector in percents.

The helper macro is `IDIP_CFG_META_INIT_REMOVAL_OBJECT_DETECTION_N_SENSITIVITY`.

```
// [optional]
{
    "name": "removal_object_detection/N/sensitivity",
    "value": 50,
    "meta": {
        "type": "int",
        "range": [ 0, 100, 1 ]
    }
}
```

The *removal_object_detection/N/id* setting allows to set up a name for the `N`-th removable object.

The helper macro is `IDIP_CFG_META_INIT_REMOVAL_OBJECT_DETECTION_N_ID`.

```
// [optional]
{
    "name": "removal_object_detection/N/id",
    "value": "bag",
    "meta": {
        "type": "string",
        "range": [ 0, 100 ]
    }
}
```

The *removal_object_detection/N/field* setting allows to enable or disable object removal detection for the `N`-th detector. If the device does not support polygons, the value type must be `rect`.

Helper macros:

```
IDIP_CFG_META_INIT_REMOVAL_OBJECT_DETECTION_N_FIELD      // for polygon
IDIP_CFG_META_INIT_REMOVAL_OBJECT_DETECTION_N_FIELD_RECT // for rect
```

Description:

```
{
    "name": "removal_object_detection/N/field",
    "value": [ [100, 100], [600, 100], [600, 400], [100, 400] ],
    "meta": {
        "type": "polygon",
        "range": [
            [0, 7680], // [ min_x, max_x ]
            [0, 4320], // [ min_y, max_y ]
            16
        ]
    }
}
```

9.7.11.2 Object removal detection events When the object removal detection algorithm detects that an object was removed it should produce an event `vca.removal_object_detection.object_removed` event.

```
{
    "name": "vca.removal_object_detection.object_removed",
    "data": {
        "field": 1, // number of detector
        "id": "bag" // object identifier string
    }
}
```

You may use the `idip_server_send_vca_removal_object_detection_object_removed_event()` function to send it.

9.7.12 Unclaimed object detection

Unclaimed object detection is a camera feature that continuously tracks objects in its FoV and notifies Cloud about new objects which are in FoV for more than 5/10/15/30 minutes.

The *unclaimed_object_detection/enabled* setting enables or disables field detection algorithm for all detection fields.

The helper macro is `IDIP_CFG_META_INIT_UNCLAIMED_OBJECT_DETECTION_ENABLED`.

```
{
    "name": "unclaimed_object_detection/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

The *unclaimed_object_detection/N/enabled* setting enables or disables the `N`-th detection unit. The helper macro is `IDIP_CFG_META_INIT_UNCLAIMED_OBJECT_DETECTION_N_ENABLED`.

```
{
    "name": "unclaimed_object_detection/N/enabled",
    "value": false,
    "meta": {
        "type": "bool"
    }
}
```

The *unclaimed_object_detection/N/sensitivity* setting allows to control the `N`-th detector sensitivity in percents. The helper macro is `IDIP_CFG_META_INIT_UNCLAIMED_OBJECT_DETECTION_N_SENSITIVITY`.

```
// [optional]
{
    "name": "unclaimed_object_detection/N/sensitivity",
    "value": 50,
    "meta": {
        "type": "int",
        "range": [ 0, 100, 1 ]
    }
}
```

The *unclaimed_object_detection/N/detection_interval* setting controls the time interval before the alert. The helper macro is `IDIP_CFG_META_INIT_UNCLAIMED_OBJECT_DETECTION_N_DETECTION_INTERVAL`.

```
// [optional]
{
    "name": "unclaimed_object_detection/N/detection_interval",
    "value": 50,
    "meta": {
        "type": "int",
        "range": [ 0, 100, 1 ]
    }
}
```

The *unclaimed_object_detection/N/field* setting allows to set up a detection area for `N`-th detection unit. If the device does not support polygons, the value type must be `rect`.

Helper macros:

```
IDIP_CFG_META_INIT_UNCLAIMED_OBJECT_DETECTION_N_FIELD      // for polygon
IDIP_CFG_META_INIT_UNCLAIMED_OBJECT_DETECTION_N_FIELD_RECT // for rect
```

Description:

```
{
    "name": "unclaimed_object_detection/N/field",
    "value": [ [100, 100], [600, 100], [600, 400], [100, 400] ],
    "meta": {
        "type": "polygon",
        "range": [
            [0, 7680], // [ min_x, max_x ]
            [0, 4320], // [ min_y, max_y ]
            4
        ]
    }
}
```

When the unclaimed object detection unit detects an unclaimed objects it sends:

```
{
    "name": "vca.unclaimed_object_detection.new_object",
    "data": {
        "field": 1,     // detection area number
        "class": "bag" // object identifier string
    }
}
```

You may use the `idip_server_vca_unclaimed_object_detection_new_object_event()` call to send the event.

9.7.13 Thermal events

There are a couple methods and helper data structures in the `ivesdk` to send thermal events. These events are describing a rectangle on a camera view where a body themperature was triggered to very high or very low.

The method `idip_server_send_vca_body_temp_high()` allows to send a shot for high temperature trigger event, the method `idip_server_send_vca_body_temp_low()` - for low themperature trigger event. Both methods are passing an abnormal themperature value in Celsius degrees (the highest or the lowest in the rectangle), a themperature threshold level (in Celsius degrees) and a “shot”. This “shot” is a data structure containt an image with an abnormal body themperature and one or more areas (rectangles) that points to the abnormal themperature region on the image.

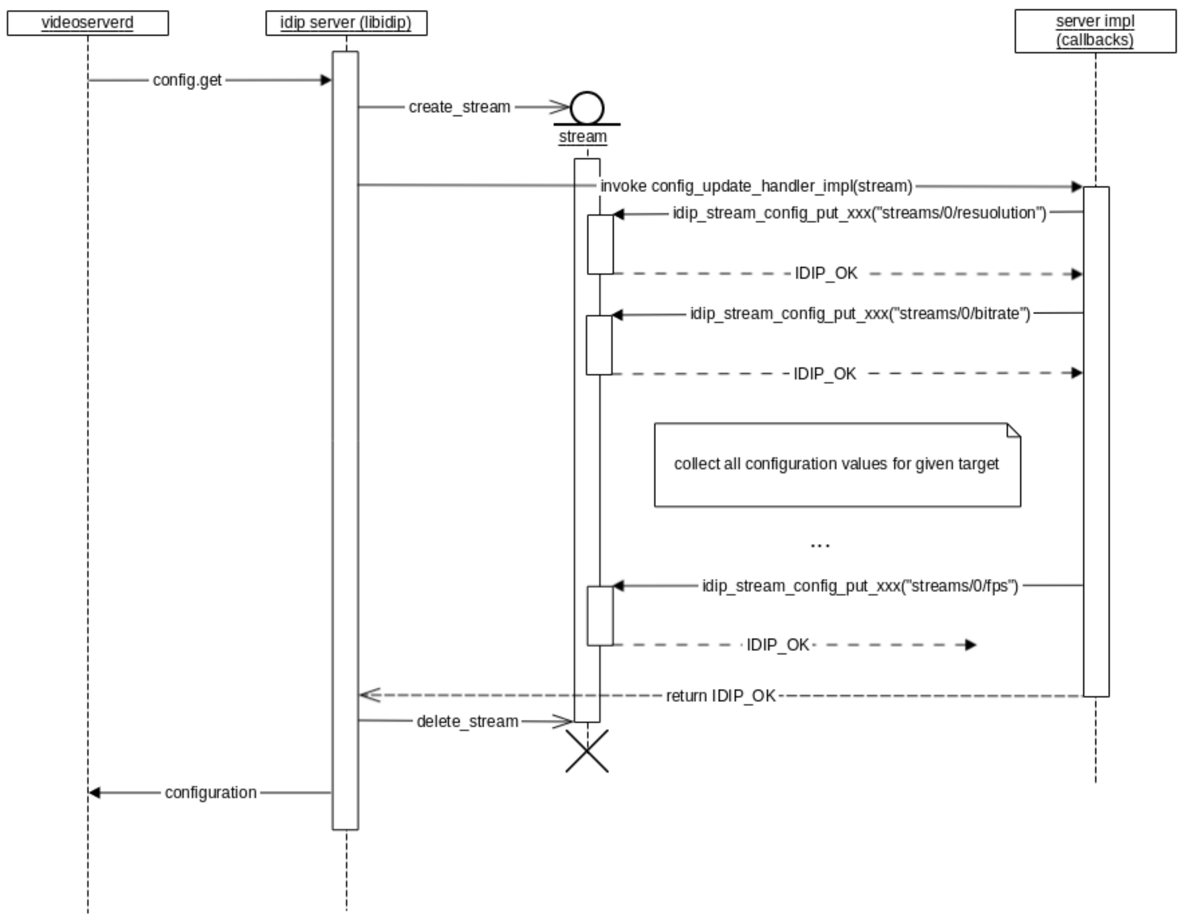
These API supports two ways to pass an image to the cloud. The first one method is to pass a path to an image in the camera local file system. The `libidip` will read the file and sent it to the cloud with an event. The second way assumes that the camera stores an image to some server in th Internet and pass an URL to the image to the `idip_ivs_vca_shot_t` data structure.

Code below show how to send a simple high themperature event with an image on camera local file system.

```c
// an area in the image with abnormal themperature
const idip_ivs_vca_rect_t r = { .x=50, .y=50, .width=100, .height = 100 };

// The "shot" describes an image, a rectangle on the image and the highest
     themperature on the rectangle.
const idip_ivs_vca_shot_t shot = {
    .image_url_or_path = "/tmp/tmp-CefdaR9Etr-shot.jpg",
    .temp_in_celsius = 37.5,
    .temp_treshold = NULL,
    .temp_threshold_size = 0,
    .rects = &r,
    .rects_size = 1
};

// sending the event
idip_server_send_vca_body_temp_high(server, "camera0", now_ms(), 38.0,
    37.5, &shot);
```

9.8 Configuration implementation notes

Device stores one or more sets of configuration values. Each set can be selected by a target name. All settings have a getter and a setter . Read operation requests **all** settings for the given target name. Developers have to send actual values for each setting for the given target name inside the `idip_stream_config_get_handler_t` callback. Each setting must be sent with its actual range and type. **DO NOT change data type of setting!** On the sequence diagram below is shown sequence of sending configuration values.


videoserverd
idip server (libidip)
server impl (callbacks)
config.get
create_stream
stream
invoke config_update_handler_impl(stream)
idip_stream_config_put_xxx("streams/0/resuolution")
IDIP_OK
idip_stream_config_put_xxx("streams/0/bitrate")
IDIP_OK
collect all configuration values for given target
...
idip_stream_config_put_xxx("streams/0/fps")
IDIP_OK
return IDIP_OK
delete_stream
configuration

`libidip` uses the `idip_stream_config_update_handler_t` callback type to notify the server about configuration changes. It calls it with an array of actual values to set.

Note. Update (write) operation is asymmetric to read operation because new values come to callback without its type and range. All names and thier types are listed above. `libidip` provides an API for reading new configuration values inside callback. There is a set of `idip_cfg_param_get_xxx()` methods. For example, to send value of the *streams/0/codec* setting developers can write the following code:

```
// It is part of possible implementation of callback
idip_status_code_t config_get_handler(
    idip_stream_config_t *stream,
    const char *target,
    void *udata)
{
    // ...

    // send actual value of "streams/0/codec" setting
    // The type of setting is choice.
    int rc = idip_stream_config_put_choice_str(stream,
        "streams/0/codec",    // setting name
        "H264",               // current value (it is string)
        { "MJPEG", "H264" }, // range:  all of possible values
        2                     // range size
    );

    // ...
}
```

In the example the value of the *streams/0/codec* setting with its range is sent. Updating procedure takes into account the type of setting value. As you can see, there are no type information inside updating procedure and we narrow the type to simple string (we use the `idip_cfg_param_get_string()` function to get a value):

```
idip_status_code_t config_update_handler(
    idip_stream_config_t *stream,
    const char *target,
    const idip_cfg_param_t **config,
    size_t config_items_count,
    void *udata)
{
    // ...

    // for each configuration parameter form query
    for(size_t i = 0; i < config_items_count; ++i) {
        // ...

        char buffer[100]; // buffer to store actual setting value
        size_t buffer_length = 0; // actual length
        // We assume that here is value of "streams/0/codec" at index i.
        // Type of value of "streams/0/codec" is string!
        int rc = idip_cfg_param_get_string(config[i],
            buffer, sizeof(buffer),
            &buffer_length);
        // For example, we can print new value.
        // Note: here is used the form of printf with field width.
        printf("streams/0/codec: %*s\n", buffer_length, buffer);

        // do apply new value
        // ...
    }

    // ...
}
```

`Videoserverd` sends only the value of the *streams/0/codec* setting without type and range information. Use the setting name to determine the type of value to read.

9.8.1 Configuration helper API

`libidip` provides some macros and functions that can help developers to implement:

- `idip_stream_config_get_handler_t` and
- `idip_stream_config_update_handler_t`.

All helpers defined in **idip/idip_cloud_config.h** file. With this API your configuration handlers would looks like this:

```
//
    -------------------------------------------------------------------------------

idip_status_code_t
config_get_handler_impl(idip_stream_config_t* stream,
                        const char* target,
                        void* udata)
{
    // Create meta information helper. We're doing it using global
       variables.
    // Production code can use udata argument to pass additional data here
       .
    idip_cfg_all_meta_t all_meta = {
        .meta = cfg_meta,
        .meta_count = ARRAY_SIZE(cfg_meta),
    };

    idip_mutex_lock(&cfg_mutex); // lock the meta array

    // Collect values from meta array and ut them to the stream
    idip_status_code_t result = idip_cfg_meta_put_values_to_stream(
       all_meta, stream, 0);

    idip_mutex_unlock(&cfg_mutex); // release the meta array

    return result;
}
//
   -----------------------------------------------------------------------------
```

and this:

```
//
   -----------------------------------------------------------------------------

idip_status_code_t
config_update_handler_impl(idip_stream_config_t* stream,
                           const char* target,
                           const idip_cfg_param_t** params,
                           size_t params_count,
                           void* udata)
{
    idip_cfg_all_meta_t all_meta = {
        .meta = cfg_meta,
        .meta_count = ARRAY_SIZE(cfg_meta),
    };
    idip_status_code_t result = IDIP_OK;

    idip_mutex_lock(&cfg_mutex);

    // Call helper. It will update all variables, which were described
    // during cfg_meta initialization. The helper will call registered
       hooks
    // (w_hook) BEFORE update of variables.
    result = idip_cfg_meta_handle_updates(all_meta, target, params,
       params_count, udata);

    // Put only modified values.
    result = idip_cfg_meta_put_values_to_stream(all_meta, stream, 1);

    idip_mutex_unlock(&cfg_mutex);
    return result;
}
//
   -----------------------------------------------------------------------------
```

They are really simple. The first one locks shared data and then puts **all** registered configuration values into the output stream. The idip_cfg_meta_put_values_to_stream() function uses a meta information array to select the appropriate idip_cfg_put_xxx() method. Then it unlocks the shared data and returns. The second handler locks shared data, then it tries to update some values which were bound to the meta information, and then it puts to the output stream only modified setting values and unlocks the shared data.

These two handlers use an array of idip_cfg_meta_t. It is a helper data type that allows to bind a variable in a program to its meta data. If you want to use the helper API in your handlers implementations you should create one (or more, one per target) array and initialize it. You have to initialize each idip_cfg_meta_t item with a specific memory layout that depends on the setting value type. libidip provides a family of IDIP_CFG_META_INIT_XXX macros which would help you to initialize meta information array items correctly. In the example above we assume that the meta information array is statically allocated. But you can create it on a heap and pass, for example, through the udata pointer to a handler.

Note. Use synchronized access to shared data inside the idip_stream_config_get_handler_t and the idip_stream_config_update_handler_t implementations since they are called from different thread contexts.

The image below explains which information is bound to which variable. Let's see an example. At first, we create some variables that would contain actual values. They are simple integers, string buffers, pointers to a constant string, idip_pos_t points. Then we create some additional static information – ranges.

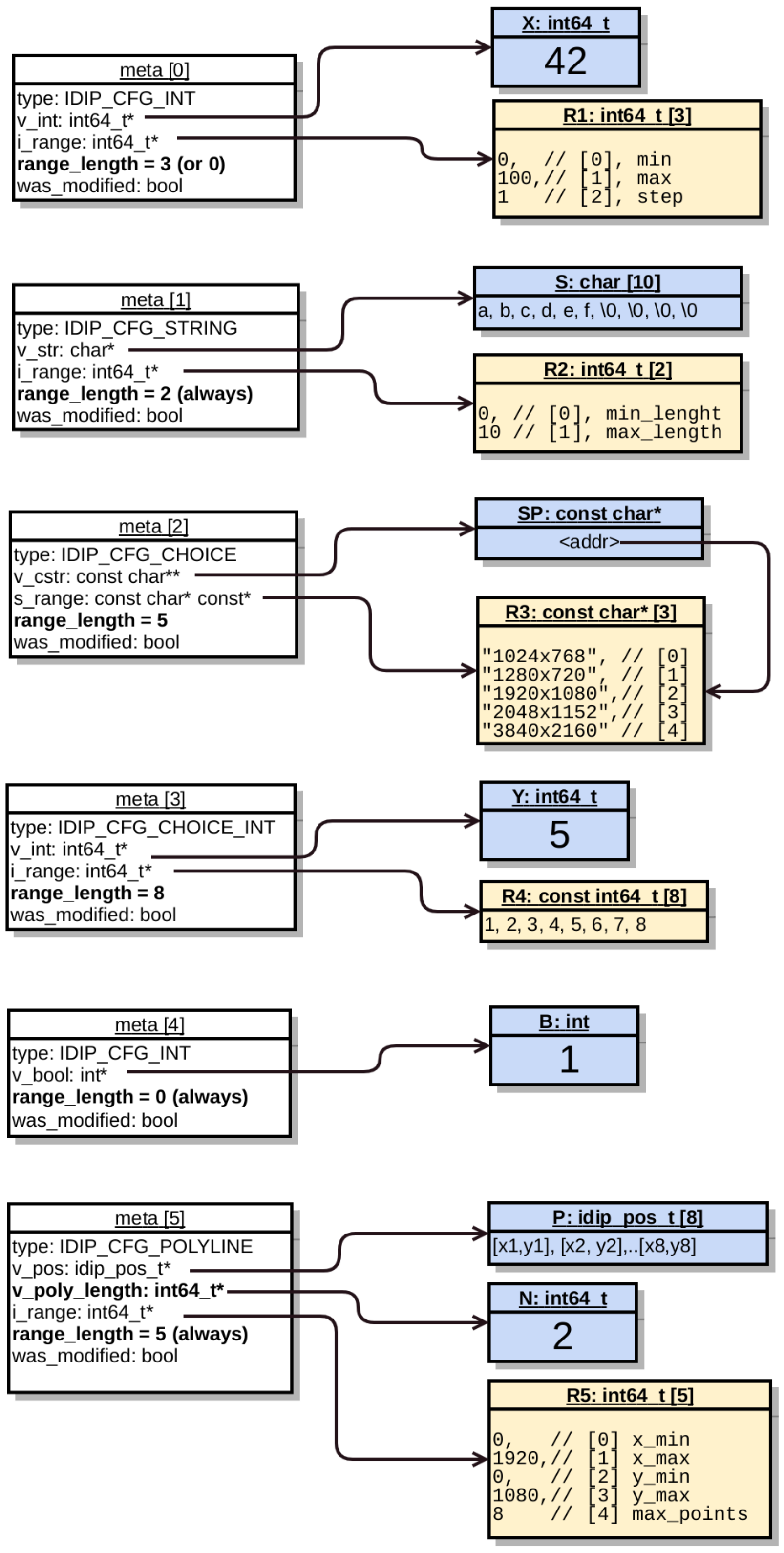
```
// include helper API
#include <idip/idip_cloud_config.h>

// ranges
static const int64_t R1[3] = { 0, 100, 1 };
static const int64_t R2[2] = { 0, 10 };
static const char* const R3[5] = { "1024x768","1280x720","1920x1080","2048
    x1152","3840x2160" };
static const int64_t R4[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
static const int64_t R5[5] = { 0, 1920, 0, 1080, 8 };

// variables which stores actual setting values
static int64_t X = 42;
static char S[10] = { 'a', 'b', 'c', 'd', 'e', 'f', '\0', '\0', '\0', '\0'
     };
static const char* SP = "1920x1080";
static int64_t Y = 5;
static int B = 1;
static idip_pos_t P[8]; // = { ... }. Allocate 8 points...
static int64_t N = 2;   // but actually we use only first two.
```

In the example we create an array cfg_meta of 6 items and initialize it with different setting types.

```
idip_cfg_meta_t cfg_meta[5] = {
    // [0]
    IDIP_CFG_META_INIT_SOUND_DETECTION_SENSITIVITY(&X, R1, NULL, NULL,
        NULL),
    // [1]
    IDIP_CFG_META_INIT_OSD_NAME_TEXT(S, R2, NULL, NULL, NULL),
    // [2]
    IDIP_CFG_META_INIT_STREAMS_N_RESOLUTION(0, &SP, R3, ARRAY_SIZE(R3),
        NULL, NULL, NULL),
    // [3]
    IDIP_CFG_META_INIT_STREAMS_N_FPS(0, &Y, R4, ARRAY_SIZE(R4), NULL, NULL
       , NULL),
    // [4]
    IDIP_CFG_META_INIT_LINE_DETECTION_N_ENABLED(0, &B, NULL, NULL, NULL),
    // [5]
    IDIP_CFG_META_INIT_LINE_DETECTION_N_SEGMENTS(0, P, &N, R5, NULL, NULL,
        NULL),
};
```

We initialize an integer setting with named *sound_detection/sensitivity* and bind it to the `X` variable. The second item in the array describes the string *osd/name/text* setting and binds it to the string buffer `S` and the range `R2`. Third array item describes choice-of-string setting type. It is bound to the `SP` pointer and the range for choice `R3`. And so on. The last setting (*field_detection/0/field*) is described by the last array item. We assume that the device supports line crossing detection with a polyline. The polyline can contain 7 segments (8 points) or less. In the initialized state it contains only 2 points (one line segment). The variable `N` contains `P`'s actual length.

The image below shows the memory layout for this example.



The `idip_cfg_meta_t.user_data` field allows you to bind some data to the setting. All `IDIP_CFG_META_XXX()` macros accept a pointer value as a last argument, which could be used in the update hook. Pay attention that this pointer is different from the `user_data` pointer passed to the hook argument during the call.

10 Sending events

This chapter shows how to use `libidip` event API to send events.

`libidip` has one generic method that sends events – `idip_server_send_event()` – and a set of wrappers with fewer parameters. All prototypes are in `idip/idip_custom_events.h` file.

Each event consists of:

- event name. Unique. Full list is in the Cloud configuration and events chapter.
- event creation time. The time when the event was detected.
- event target name. Optional. Specifies a subscriber name for which the event is intended.
- event arguments. Extended data bound to the event. Depends on the event name, some events have only the *name* field.

In JSON notation:

```
{
    "name": "vca.face_detection.face",
    "data": {
        "track_begin": 10, // int
        "track_end": 25,   // int
        "best_shot": {
            "ts": 1565688051000, // int, milliseconds in UTC
            "data": BLOB,  // face crop in JPEG format
            "width": 100,  // int. In pixels. Optional.
            "height": 100, // int. In pixels. Optional.
            "position": {  // optional
                "x": 300,  // int
                "y": 400,  // int
                "width": 100, // int
                "height": 100 // int
            }
        }
    }
}
```

Event arguments are created with `idip_custom_arg_xxx` functions. Created argument has the `idip_custom_arg_t` type and binds custom data to the event.

Note. Since the `idip_custom_arg_t` object stores only a pointer without data copying, the lifetime of the bound argument must exceed the `idip_custom_arg_t`'s one.

For illustration purposes we define a helper macro that shortenes error handling in the example. We don't recommend it in the actual code.

```
#define IS_OK( stmt ) \
if( CUSTOM_ARG_OK != (rc = (stmt)) ) { \
    printf( "Can't append value. Code %d at line %d\n", rc, __LINE__); \
    goto fail; \
}
```

Next, we create a method that sends face detection events.

```
int send_face_detection_event_example(idip_server_t* server)
{
    int rc = 0;
    idip_custom_arg_t* arg = idip_custom_arg_new();
    if( !arg ) {
        /* error handling */
        printf("Can't create custom event arg\n");
        return -1;
    }

    // Concrete values in the example are randomly generated data.

    // fake jpeg image data.
    // NOTE! Life time of bounded arguments MUST BE greater than
    // life time of idip_custom_arg_t instance!
    static const char face_jpeg[] = { /* 0x01, 0x02, ... */ };

    IS_OK( idip_custom_arg_append_int(arg, "track_begin", 42) );
    IS_OK( idip_custom_arg_append_int(arg, "track_end",   84) );
    // "Open" a sub object (dictionary)
    IS_OK( idip_custom_arg_begin_dict(arg, "best_shot") );
        // The field "ts" is appending here to "best_shot" object.
        IS_OK( idip_custom_arg_append_int64(arg, "ts", 1565688051000ll) );
        IS_OK( idip_custom_arg_append_int(arg, "width",  256) );
        IS_OK( idip_custom_arg_append_int(arg, "height", 256) );
        // Begin level 2 sub object. Position and crop image size inside
        // full image.
        IS_OK( idip_custom_arg_begin_dict(arg, "position") );
            // Fields "x", "y", "width" and "height"
            // are adding here to the "best_shot.position" object.
            IS_OK( idip_custom_arg_append_int(arg, "x", 732) );
            IS_OK( idip_custom_arg_append_int(arg, "y", 476) );
            IS_OK( idip_custom_arg_append_int(arg, "width",  256) );
            IS_OK( idip_custom_arg_append_int(arg, "height", 256) );
        // "Close" the "position" sub object.
        IS_OK( idip_custom_arg_end_dict(arg) );
        // Append the next one field to "best_shot" object.
        // Data bytes aren't copying here.
        IS_OK( idip_custom_arg_append_blob(arg, "data", face_jpeg,
                                           sizeof(face_jpeg)) );
    // Finalize the "best_shot" object.
    IS_OK( idip_custom_arg_end_dict(arg) );

    // All custom data fields were described.
    // Notify idip server instance about
    // "vca.face_detection.face" event
    rc = idip_server_send_event(
        server, // server instance
        "camera0", // target name
        "vca.face_detection.face", // event name
        now(), // event detection time
        arg); // event arguments
fail:
    idip_custom_arg_delete(arg);

    if( IDIP_OK != rc  )
        return -1;
    return 0;
}
```

11 Changelog

Version 2.0

- Agentless sheme support
- Network backend is using libwebsockets instead libev.
- IDIP protocol up to dated to v2.

Version 1.2

- Added temperature events API.
- Minor bug fixes.

Version 1.1

- Bug fixes and minor enhancements.
- Added lbbidip version information access API.

Version 1.0

- Minor bug fixes.
- Added version information.
- Dividing to binary and source releases.

Version 0.11

- Added image configuration settings.
- Consistency in event naming.
- Added local password changing API.
- Refactored network subsystem.
- Major bug fixes.

Version 0.10a

- Subscription to all events (with videoserver >= 3.5.0.4287)

Version 0.10

- OTA feature (firmware fetching URL)
- Typos fixed
- Documentation updates

Version 0.9

- Expanded event API. Added some usefull helpers.
- Added cloud configuration helper API.
- Advanced analytics events support with unclaimed object detection and object removal detection.
- Allow users to set different data pointers for handler implementations.
- Major bug fixes.
- Added skeleton application.

Version 0.8

- Live video streaming API.
- Explicit server feature initialization API.
- Server initialization structures were refactored.
- Advanced demo example that uses the new streaming API.
- Advanced server information.
- Minor bug fixes.

Version 0.7

- Analytics events support: Motion Detection, Face Detection, Line Crossing, Area Invasion/Intrusion Detection, Loiter Detection, Tamper Detection, Removal of Object, Unclaimed Object, Fire/Smoke Detection, Face Recognition, Sound (Panic/SOS/Crying/Breaking Glass/Heavy Object Falling) Detection.
- Documentation is proveded as User's Handbook

Version 0.6

- Event API supports custom event arguments
- Major bug fixes
- Advanced demo example, that can play archives

Version 0.5

- Configuration API.
- Archive API fixes.
- Enchanced demo example with configuration and archive callbacks.

Version 0.4

- Snapshot API.
- Firmware upgrade API
- PTZ control API.

- WiFi configuration API.
- H.265 codec support.
- Notification API.
- Arbitrary data save/load API.
- Events API now can work with events from various sources (targets).
- Fixed bugs and enhanced demo example.

Version 0.3

- New build system (based on CMake) with cross compilation support.
- All third-party dependencies are included in SDK package (msgpack-c and libev).
- Documentation: overview and quick start documents + detailed docs generated from sources using doxygen.
- Advanced demo example, that uses ffmpeg for streaming video from mp4 file.

Version 0.2

- Network core (based on libev) and IDIP protocol transport layer.
- Live video/audio streaming API.
- Video archive playback API.
- Push-to-talk API.
- Generic events API.
- Device configuration API.

Version 0.1

- First draft.