# PJSUA2 Documentation

***Release 1.0-alpha***

**Sauw Ming Liong, Benny Prijono**

February 27, 2014

Contents:

CHAPTER

ONE

# INTRODUCTION

This documentation is intended for developers looking to develop Session Initiation Protocol (SIP) based client application. Some knowledge on SIP is definitely required, and of course some programming experience. Prior knowledge of PJSUA C API is not needed, although it will probably help.

PJSIP libraries provide multi-level APIs to do SIP calls, presence, and instant messaging, as well as handling media and NAT traversal. PJSUA2 API is the highest API from PJSIP, on top of PJSUA-LIB API. PJSUA-LIB API itself is a library that unifies SIP, audio/video media, NAT traversal, and client media application best practices into a high level, integrated, and easy to use API. The next chapter will guide you on selecting which API level to use depending on your requirements.

This documentation can be viewed online, or alternatively you can download the PDF format for offline viewing.

## 1.1 Getting Started with PJSIP

Check PJSIP Datasheet to make sure that it has the features that you require.

To start using PJSIP, the Getting Started Guide contains instructions to acquire and build PJSIP on various platforms that we support.

## 1.2 PJSIP Info and Documentation

To get other relevant info and documentations about PJSIP, you can visit:

- PJSIP General Wiki is the home for all documentation
- PJSIP FAQ
- PJSIP Reference Manual - please see Reference Manual section

CHAPTER

TWO

# DEVELOPMENT GUIDELINES AND CONSIDERATIONS

## 2.1 Development Guidelines

### 2.1.1 Preparation

- **Essential:** Familiarise yourself with SIP. You don't need to be an expert, but SIP knowledge is essential.
- Check out our features in Datasheet. Other features may be provided by our community.
- All PJSIP documentation is indexed in our Trac site.

### 2.1.2 Development

- **Essential:** Follow the Getting Started instructions to build PJSIP for your platform.
- **Essential:** Interactive debugging capability is essential during development
- Start with default settings in <*pj/config_site_sample.h*>. The default settings should be good to get you started. You can always optimize later after things are running okay.

### 2.1.3 Coding Style

**Essential:** set your editor to use 8 characters tab size in order to see PJSIP source correctly.

Detailed below is the PJSIP coding style. You don't need to follow it unless you are submitting patches to PJSIP:

- Indentation uses tabs and spaces. Tab size is 8 characters, indentation 4.
- All public API in header file must be documented in Doxygen format.
- Apart from that, we mostly just use K & R style, which is the only correct style anyway.

### 2.1.4 Deployment

- **Essential:** Logging is essential when troubleshooting any problems. The application MUST be equipped with logging capability. Enable PJSIP log at level 5.

## 2.2 Platform Consideration

Platform selection is usually driven by business motives. The selection will affect all aspects of development, and here we will cover considerations for each platforms that we support.

### 2.2.1 Windows Desktop

Windows is supported from Windows 2000 up to the recent Windows 8 and beyond. All features are expected to work. 64bit support was added recently. Development is based on Visual Studio. Considerations for this platform include:

1. Because Visual Studio file format keeps changing on every release, we decided to support the lowest denominator, namely Visual Studio 2005. Unfortunately the project upgrade procedure fails on Visual Studio 2010, and we don't have any solution for that. VS 2008 and VS 2012 onwards should work.

### 2.2.2 MacOS X

All features are expected to work. Considerations include:

1. Development with XCode is currently not supported. This is **not** to say that you cannot use XCode, but PJSIP only provides basic Makefiles and if you want to use XCode you'd need to arrange the project yourself.
2. Mac systems typically provides very good sound device, so we don't expect any problems with audio on Mac.

### 2.2.3 Linux Desktop

All features are expected to work. Linux considerations:

1. Use our native ALSA backend instead of PortAudio because ALSA has less jitter than OSS and our backend is more lightweight than PortAudio

### 2.2.4 iOS for iPhone, iPad, and iPod Touch

All features except video are expected to work (video is coming soon!). Considerations for iOS:

1. You need to use TCP transport for SIP for the background feature to work
2. IP change (for example when user is changing access point) is a feature frequently asked by developers and you can find the documentation here: http://trac.pjsip.org/repos/wiki/IPAddressChange
3. There are some specific issues for iOS 7 and beyond, please see http://trac.pjsip.org/repos/ticket/1697
4. If SSL is needed, you need to compile OpenSSL for iOS

### 2.2.5 Android

All features except video are expected to work (video is coming soon!). Considerations for Android:

1. You can only use PJSUA2 Java binding for this target.
2. It has been reported that Android audio device is not so good in general, so some audio tuning may be needed. Echo cancellation also needs to be checked.
3. This is also a new platform for us.

### 2.2.6 Symbian

Symbian has been supported for a long time. In general all features (excluding video) are expected to work, but we're not going to do Symbian specific development anymore. Other considerations for Symbian:

1. The MDA audio is not very good (it has high latency), so normally you'd want to use Audio Proxy Server (APS) or VoIP Audio Service (VAS) for the audio device, which we support. Using these audio backends will also provide us with high quality echo cancellation as well as low bitrate codecs such as AMR-NB, G.729, and iLBC. But VAS and APS requires purchase of Nokia development certificate to sign the app, and also since APS and VAS only run on specific device type, you need to package the app carefully and manage the deployment to cover various device types.

### 2.2.7 BlackBerry 10

BlackBerry 10 (BB10) is supported since PJSIP version 2.2. As this is a relatively new platform for us, we are currently listening to developer's feedback regarding the port. But so far it seems to be working well. Some considerations for BB10 platform include:

1. IP change (for example when user is changing access point) is a feature frequently asked by developers and you can find the documentation here: http://trac.pjsip.org/repos/wiki/IPAddressChange

### 2.2.8 Windows Mobile

This is the old Windows Mobile platform that is based on WinCE. This platform has been supported for a long time. We expect all features except video to work, but there may be some errors every now and then because this target is not actively maintained. No new development will be done for this platform.

Other considerations for Windows Mobile platform are:

1. The quality of audio device on WM varies a lot, and this affects audio latency. Audio latency could go as high as hundreds of millisecond on bad hardware.
2. Echo cancellation could be a problem. We can only use basic echo suppressor due to hardware limitation, and combined with bad quality of audio device, it may cause ineffective echo cancellation. This could be mitigated by setting the audio level to low.

### 2.2.9 Windows Phone 8

Windows Phone 8 (WP8) support is being added and is still under development on *projects/winphone* branch. Specific considerations for this platform are:

1. WP8 governs specific interaction with WP8 GUI and framework that needs to be followed by application in order to make VoIP call work seamlessly on the device. Some lightweight process will be created by WP8 framework in order for background call to work and PJSIP needs to put its background processing in this process' context. Currently this feature is under development.

### 2.2.10 Embedded Linux

In general embedded Linux support is similar to Linux and we find no problems with it. We found some specific considerations for embedded Linux as follows:

1. The performance of the audio device is probably the one with most issues, as some development boards does not have a decent sound device. Typically there is high audio jitter (or burst) and latency. This will affect end to end audio latency and also the performance of the echo canceller. Also we found that ALSA generally works better than OSS, so if you can have ALSA up and running that will be better. Use our native ALSA backend audio device instead of PortAudio since it is simpler and lighter.

### 2.2.11 QNX or Other Posix Embedded OS

This is not part of our officially supported OS platforms, but users have run PJSIP on QNX and BlackBerry 10 is based on QNX too. Since QNX provides Posix API, and maybe by using the settings found in the configure-bb10 script, PJSIP should be able to run on it, but you need to develop PJMEDIA sound device wrapper for your audio device. Other than this, we don't have enough experience to comment on the platform.

### 2.2.12 Other Unix Desktop OSes

Community members, including myself, have occasionally run PJSIP on other Unix OSes such as Solaris, FreeBSD, and OpenBSD. We expect PJSIP to run on these platforms (maybe with a little kick).

### 2.2.13 Porting to Other Embedded OS

It is possible to port PJSIP to other embedded OS or even directly to device without OS and people have done so. In general, the closer resemblance the new OS to existing supported OS, the easier the porting job will be. The good thing is, PJSIP has been made to be very very portable, and system dependent features are localized in PJLIB and PJMEDIA audio device, so the effort is more quantifiable. Once you are able to successfully run *pjlib-test*, you are more or less there with your porting effort. Other than that, if you really want to port PJSIP to new platform, you probably already know what you're doing.

## 2.3 Which API to Use

### 2.3.1 PJSIP, PJMEDIA, and PJNATH Level

At the lowest level we have the individual PJSIP **C** libraries, which consist of PJSIP, PJMEDIA, and PJNATH, with PJLIB-UTIL and PJLIB as support libraries. This level provides the most flexibility, but it's also the hardest to use. The only reason you'd want to use this level is if:

1. You only need the individual library (say, PJNATH)
2. You need to be very very tight in footprint (say when things need to be measured in Kilobytes instead of Megabytes)
3. You are **not** developing a SIP client

Use the corresponding PJSIP, PJMEDIA, PJNATH manuals from http://trac.pjsip.org/repos/ for information on how to use the libraries. If you use PJSIP, the PJSIP Developer's Guide (PDF) from that page provides in-depth information about PJSIP library.

### 2.3.2 PJSUA-LIB API

Next up is PJSUA-LIB API that combines all those libraries into a high level, integrated client user agent library written in C. This is the library that most PJSIP users use, and the highest level abstraction before pjsua2 was created.

Motivations for using PJSUA-LIB library includes:

1. Developing client application (PJSUA-LIB is optimized for developing client app)
2. Better efficiency than higher level API

### 2.3.3 PJSUA2 C++ API

pjsua2 is a new, objected oriented, C++ API created on top of PJSUA-LIB. The API is different than PJSUA-LIB, but it should be even easier to use and it should have better documentation too (such as this book). The pjsua2 API removes most cruxes typically associated with PJSIP, such as the pool and pj_str_t, and add new features such as object persistence so you can save your configs to a file, for example. All data structures are rewritten for more clarity.

A C++ application can use pjsua2 natively, while at the same time still has access to the lower level objects if it needs to. This means that the C++ application should not lose any information from using the C++ abstraction, compared to if it is using PJSUA-LIB directly. The C++ application also should not lose the ability to extend the library. It would still be able to register a custom PJSIP module, pjmedia_port, pjmedia_transport, and so on.

Benefits of using pjsua2 C++ API include:

1. Cleaner object oriented API
2. Uniform API for higher level language such as Java and Python
3. Persistence API
4. The ability to access PJSUA-LIB and lower level libraries when needed (including the ability to extend the libraries, for example creating custom PJSIP module, pjmedia_port, pjmedia_transport, etc.)

Some considerations on PJSUA2 C++ API are:

1. Instead of returning error, the API uses exception for error reporting
2. It uses standard C++ library (STL)
3. The performance penalty due to the API abstraction should be negligible on typical modern device

### 2.3.4 PJSUA2 API for Java, Python, and Others

The PJSUA2 API is also available for non-native code via SWIG binding. Configurations for Java and Python are provided with the distribution. Thanks to SWIG, other language bindings may be generated relatively easily.

The pjsua2 API for non-native code is effectively the same as pjsua2 C++ API. However, unlike C++, you cannot access PJSUA-LIB and the underlying C libraries from the scripting language, hence you are limited to what pjsua2 provides.

You can use this API if native application development is not available in target platform (such as Android), or if you prefer to develop with non-native code instead of C/C++.

## 2.4 Network and Infrastructure Considerations

### 2.4.1 NAT Issues

TBD.

### 2.4.2 TCP Requirement

If you support iOS devices in your service, you need to use TCP, because only TCP will work on iOS device when it is in background mode. This means your infrastructure needs to support TCP.

## 2.5 Sound Device

### 2.5.1 Latency

TBD.

### 2.5.2 Echo Cancellation

TBD.

CHAPTER

THREE

# PJSUA2-HIGH LEVEL API

PJSUA2 is an object-oriented abstraction above PJSUA API. It provides high level API for constructing Session Initiation Protocol (SIP) multimedia user agent applications (a.k.a Voice over IP/VoIP softphones). It wraps together the signaling, media, and NAT traversal functionality into easy to use call control API, account management, buddy list management, presence, and instant messaging, along with multimedia features such as local conferencing, file streaming, local playback, and voice recording, and powerful NAT traversal techniques utilizing STUN, TURN, and ICE.

PJSUA2 is implemented on top of PJSUA-LIB API. The SIP and media features and object modelling follows what PJSUA-LIB provides (for example, we still have accounts, call, buddy, and so on), but the API to access them is different. These features will be described later in this chapter. PJSUA2 is a C++ library, which you can find under `pjsip` directory in the PJSIP distribution. The C++ library can be used by native C++ applications directly. But PJSUA2 is not just a C++ library. From the beginning, it has been designed to be accessible from high level non-native languages such as Java and Python. This is achieved by SWIG binding. And thanks to SWIG, binding to other languages can be added relatively easily in the future.

PJSUA2 API declaration can be found in `pjsip/include/pjsua2` while the source codes are located in `pjsip/src/pjsua2`. It will be automatically built when you compile PJSIP.

## 3.1 PJSUA2 Main Classes

Here are the main classes of the PJSUA2:

### 3.1.1 Endpoint

This is the main class of PJSUA2. You need to instantiate one and exactly one of this class, and from the instance you can then initialize and start the library.

### 3.1.2 Account

An account specifies the identity of the person (or endpoint) on one side of SIP conversation. At least one account instance needs to be created before anything else, and from the account instance you can start making/receiving calls as well as adding buddies.

### 3.1.3 Media

This is an abstract base class that represents a media element which is capable to either produce media or takes media. It is then subclassed into `AudioMedia`, which is then subclassed into concrete classes such as `AudioMediaPlayer` and `AudioMediaRecorder`.

### 3.1.4 Call

This class represents an ongoing call (or speaking technically, an INVITE session) and can be used to manipulate it, such as to answer the call, hangup the call, put the call on hold, transfer the call, etc.

### 3.1.5 Buddy

This class represents a remote buddy (a person, or a SIP endpoint). You can subscribe to presence status of a buddy to know whether the buddy is online/offline/etc., and you can send and receive instant messages to/from the buddy.

## 3.2 General Concepts

### 3.2.1 Class Usage Patterns

With the methods of the main classes above, you will be able to invoke various operations to the object quite easily. But how can we get events/notifications from these classes? Each of the main classes above (except Media) will get their events in the callback methods. So to handle these events, just derive a class from the corresponding class (Endpoint, Call, Account, or Buddy) and implement/override the relevant method (depending on which event you want to handle). More will be explained in later sections.

### 3.2.2 Error Handling

We use exceptions as means to report error, as this would make the program flows more naturally. Operations which yield error will raise Error exception. If you prefer to display the error in more structured manner, the Error class has several members to explain the error, such as the operation name that raised the error, the error code, and the error message itself.

### 3.2.3 Asynchronous Operations

If you have developed applications with PJSIP, you'll know about this already. In PJSIP, all operations that involve sending and receiving SIP messages are asynchronous, meaning that the function that invokes the operation will complete immediately, and you will be given the completion status as callbacks.

Take a look for example the makeCall() method of the Call class. This function is used to initiate outgoing call to a destination. When this function returns successfully, it does not mean that the call has been established, but rather it means that the call has been initiated successfully. You will be given the report of the call progress and/or completion in the onCallState() callback method of Call class.

### 3.2.4 Threading

For platforms that require polling, the PJSUA2 module provides its own worker thread to poll PJSIP, so it is not necessary to instantiate own your polling thread. Having said that the application should be prepared to have the callbacks called by different thread than the main thread. The PJSUA2 module itself is thread safe.

Often though, especially if you use PJSUA2 with high level languages such as Python, it is required to disable PJSUA2 internal worker threads by setting EpConfig.uaConfig.threadCnt to 0, because the high level environment doesn't like to be called by external thread (such as PJSIP's worker thread).

### 3.2.5 Problems with Garbage Collection

Garbage collection (GC) exists in Java and Python (and other languages, but we don't support those for now), and there are some problems with it when it comes to PJSUA2 usage:

- it delays the destruction of objects (including PJSUA2 objects), causing the code in object's destructor to be executed out of order
- the GC operation may run on different thread not previously registered to PJLIB, causing assertion

Due to problems above, application ''MUST immediately destroy PJSUA2 objects using object's delete() method (in Java)'', instead of relying on the GC to clean up the object.

For example, to delete an Account, it's **NOT** enough to just let it go out of scope. Application MUST delete it manually like this (in Java):

```
acc.delete();
```

### 3.2.6 Objects Persistence

PJSUA2 includes PersistentObject class to provide functionality to read/write data from/to a document (string or file). The data can be simple data types such as boolean, number, string, and string arrays, or a user defined object. Currently the implementation supports reading and writing from/to JSON document ([http://tools.ietf.org/html/rfc4627 RFC 4627]), but the framework allows application to extend the API to support other document formats.

As such, classes which inherit from PersistentObject, such as EpConfig (endpoint configuration), AccountConfig (account configuration), and BuddyConfig (buddy configuration) can be loaded/saved from/to a file. Heres an example to save a config to a file:

```
EpConfig epCfg;
JsonDocument jDoc;
epCfg.uaConfig.maxCalls = 61;
epCfg.uaConfig.userAgent = "Just JSON Test";
jDoc.writeObject(epCfg);
jDoc.saveFile("jsontest.js");
```

To load from the file:

```
EpConfig epCfg;
JsonDocument jDoc;
jDoc.loadFile("jsontest.js");
jDoc.readObject(epCfg);
```

## 3.3 Building PJSUA2

The PJSUA2 C++ library will be built by default by PJSIP build system. Standard C++ library is required.

## 3.4 Building Python and Java SWIG Modules

The SWIG modules for Python and Java are built by invoking `make` and `make install` manually from `pjsip-apps/src/swig` directory. The `make install` will install the Python SWIG module to user's `site-packages` directory.

### 3.4.1 Requirements

1. SWIG
2. JDK.
3. Python, version 2.7 or above is required. For **Linux/UNIX**, you will also need Python developent package (called python-devel (e.g. on Fedora) or python2.7-dev (e.g. on Ubuntu)). For **Windows**, you will need MinGW and Python SDK such as ActivePython-2.7.5 from ActiveState.

### 3.4.2 Testing The Installation

To test the installation, simply run python and import pjsua2 module:

```
$ python
> import pjsua2
> ^Z
```

## 3.5 Using in C++ Application

As mentioned in previous chapter, a C++ application can use *pjsua2* natively, while at the same time still has access to the lower level objects and the ability to extend the libraries if it needs to. Using the API will be exactly the same as the API reference that is written in this book.

Here is a sample complete C++ application to give you some idea about the API. The snippet below initializes the library and creates an account that registers to our pjsip.org SIP server.

```
#include <pjsua2.hpp>
#include <iostream>

using namespace pj;

// Subclass to extend the Account and get notifications etc.
class MyAccount : public Account {
public:
    virtual void onRegState(OnRegStateParam &prm) {
        AccountInfo ai = getInfo();
        std::cout << (ai.regIsActive? "*** Register:" : "*** Unregister:")
                  << " code=" << prm.code << std::endl;
    }
};

int main()
{
    Endpoint ep;

    ep.libCreate();

    // Initialize endpoint
    EpConfig ep_cfg;
    ep.libInit( ep_cfg );

    // Create SIP transport. Error handling sample is shown
    TransportConfig tcfg;
    tcfg.port = 5060;
    try {
```

```
        ep.transportCreate(PJSIP_TRANSPORT_UDP, tcfg);
    } catch (Error &err) {
        std::cout << err.info() << std::endl;
        return 1;
    }

    // Start the library (worker threads etc)
    ep.libStart();
    std::cout << "*** PJSUA2 STARTED ***" << std::endl;

    // Configure an AccountConfig
    AccountConfig acfg;
    acfg.idUri = "sip:test@pjsip.org";
    acfg.regConfig.registrarUri = "sip:pjsip.org";
    AuthCredInfo cred("digest", "*", "test", 0, "secret");
    acfg.sipConfig.authCreds.push_back( cred );

    // Create the account
    MyAccount *acc = new MyAccount;
    acc->create(acfg);

    // Here we don't have anything else to do..
    pj_thread_sleep(10000);

    // Delete the account. This will unregister from server
    delete acc;

    // This will implicitly shutdown the library
    return 0;
}
```

## 3.6 Using in Python Application

The equivalence of the C++ sample code above in Python is as follows:

```
# Subclass to extend the Account and get notifications etc.
class Account(pj.Account):
  def onRegState(self, prm):
      print "***OnRegState: " + prm.reason

# pjsua2 test function
def pjsua2_test():
  # Create and initialize the library
  ep_cfg = pj.EpConfig()
  ep = pj.Endpoint()
  ep.libCreate()
  ep.libInit(ep_cfg)

  # Create SIP transport. Error handling sample is shown
  sipTpConfig = pj.TransportConfig();
  sipTpConfig.port = 5060;
  ep.transportCreate(pj.PJSIP_TRANSPORT_UDP, sipTpConfig);
  # Start the library
  ep.libStart();

  acfg = pj.AccountConfig();
```

```
  acfg.idUri = "sip:test@pjsip.org";
  acfg.regConfig.registrarUri = "sip:pjsip.org";
  cred = pj.AuthCredInfo("digest", "*", "test", 0, "pwtest");
  acfg.sipConfig.authCreds.append( cred );
  # Create the account
  acc = Account();
  acc.create(acfg);
  # Here we don't have anything else to do..
  time.sleep(10);

  # Destroy the library
  ep.libDestroy()

#
# main()
#
if __name__ == "__main__":
  pjsua2_test()
```

## 3.7 Using in Java Application

The equivalence of the C++ sample code above in Java is as follows:

```
import org.pjsip.pjsua2.*;

// Subclass to extend the Account and get notifications etc.
class MyAccount extends Account {
  @Override
  public void onRegState(OnRegStateParam prm) {
      System.out.println("*** On registration state: " + prm.getCode() + prm.getReason());
  }
}

public class test {
  static {
      System.loadLibrary("pjsua2");
      System.out.println("Library loaded");
  }

  public static void main(String argv[]) {
      try {
          // Create endpoint
          Endpoint ep = new Endpoint();
          ep.libCreate();
          // Initialize endpoint
          EpConfig epConfig = new EpConfig();
          ep.libInit( epConfig );
          // Create SIP transport. Error handling sample is shown
          TransportConfig sipTpConfig = new TransportConfig();
          sipTpConfig.setPort(5060);
          ep.transportCreate(pjsip_transport_type_e.PJSIP_TRANSPORT_UDP, sipTpConfig);
          // Start the library
          ep.libStart();

          AccountConfig acfg = new AccountConfig();
          acfg.setIdUri("sip:test@pjsip.org");
```

```
            acfg.getRegConfig().setRegistrarUri("sip:pjsip.org");
            AuthCredInfo cred = new AuthCredInfo("digest", "*", "test", 0, "secret");
            acfg.getSipConfig().getAuthCreds().add( cred );
            // Create the account
            MyAccount acc = new MyAccount();
            acc.create(acfg);
            // Here we don't have anything else to do..
            Thread.sleep(10000);
            /* Explicitly delete the account.
             * This is to avoid GC to delete the endpoint first before deleting
             * the account.
             */
            acc.delete();

            // Explicitly destroy and delete endpoint
            ep.libDestroy();
            ep.delete();

        } catch (Exception e) {
            System.out.println(e);
            return;
        }
    }
}
```

CHAPTER

FOUR

# ENDPOINT

The Endpoint class is a singleton class, and application MUST create one and at most one of this class instance before it can do anything else, and similarly, once this class is destroyed, application must NOT call any library API. This class is the core class of PJSUA2, and it provides the following functions:

- Starting up and shutting down
- Customization of configurations, such as core UA (User Agent) SIP configuration, media configuration, and logging configuration

This chapter will describe the functions above.

To use the Endpoint class, normally application does not need to subclass it unless:

- application wants to implement/override Endpoints callback methods to get the events such as transport state change or NAT detection completion, or
- application schedules a timer using Endpoint.utilTimerSchedule() API. In this case, application needs to implement the onTimer() callback to get the notification when the timer expires.

## 4.1 Instantiating the Endpoint

Before anything else, you must instantiate the Endpoint class:

```
Endpoint *ep = new Endpoint;
```

Once the endpoint is instantiated, you can retrieve the Endpoint instance using Endpoint.instance() static method.

## 4.2 Creating the Library

Create the library by calling its libCreate() method:

```
try {
    ep->libCreate();
} catch(Error& err) {
    cout << "Startup error: " << err.info() << endl;
}
```

The libCreate() method will raise exception if error occurs, so we need to trap the exception using try/catch clause as above.

## 4.3 Initializing the Library and Configuring the Settings

The EpConfig class provides endpoint configuration which allows the customization of the following settings:

- UAConfig, to specify core SIP user agent settings.
- MediaConfig, to specify various media *global* settings
- LogConfig, to customize logging settings.

Note that some settings can be further specified on per account basis, in the AccountConfig.

To customize the settings, create instance of EpConfig class and specify them during the endpoint initialization (will be explained more later), for example:

```
EpConfig ep_cfg;
ep_cfg.logConfig.level = 5;
ep_cfg.uaConfig.maxCalls = 4;
ep_cfg.mediaConfig.sndClockRate = 16000;
```

Next, you can initialize the library by calling libInit():

```
try {
    EpConfig ep_cfg;
    // Specify customization of settings in ep_cfg
    ep->libInit(ep_cfg);
} catch(Error& err) {
    cout << "Initialization error: " << err.info() << endl;
}
```

The snippet above initializes the library with the default settings.

## 4.4 Creating One or More Transports

Application needs to create one or more transports before it can send or receive SIP messages:

```
try {
    TransportConfig tcfg;
    tcfg.port = 5060;
    TransportId tid = ep->transportCreate(PJSIP_TRANSPORT_UDP, tcfg);
} catch(Error& err) {
    cout << "Transport creation error: " << err.info() << endl;
}
```

The transportCreate() method returns the newly created Transport ID and it takes the transport type and TransportConfig object to customize the transport settings like bound address and listening port number. Without this, by default the transport will be bound to INADDR_ANY and any available port.

There is no real use of the Transport ID, except to create userless account (with Account.create(), as will be explained later), and perhaps to display the list of transports to user if the application wants it.

## 4.5 Starting the Library

Now we're ready to start the library. We need to start the library to finalize the initialization phase, e.g. to complete the initial STUN address resolution, initialize/start the sound device, etc. To start the library, call libStart() method:

```
try {
    ep->libStart();
} catch(Error& err) {
    cout << "Startup error: " << err.info() << endl;
}
```

## 4.6 Shutting Down the Library

Once the application exits, the library needs to be shutdown so that resources can be released back to the operating system. Although this can be done by deleting the Endpoint instance, which will internally call libDestroy(), it is better to call it manually because on Java or Python there are problems with garbage collection as explained earlier:

```
ep->libDestroy();
delete ep;
```

## 4.7 Class Reference

### 4.7.1 The Endpoint

*class* **pj::Endpoint**

*Endpoint* represents an instance of pjsua library.

There can only be one instance of pjsua library in an application, hence this class is a singleton.

*Public Functions*

**Endpoint**()

Default constructor.

**~Endpoint**()

Virtual destructor.

*Version* **libVersion**()

Get library version.

void **libCreate**()

Instantiate pjsua application.

Application must call this function before calling any other functions, to make sure that the underlying libraries are properly initialized. Once this function has returned success, application must call destroy() before quitting.

pjsua_state **libGetState**()

Get library state.

**Return**

library state.

void **libInit**(const *EpConfig* & prmEpConfig)

Initialize pjsua with the specified settings.

All the settings are optional, and the default values will be used when the config is not specified.

Note that create() MUST be called before calling this function.

**Parameters**

- `prmEpConfig` - *Endpoint* configurations

void **libStart**()

*Call* this function after all initialization is done, so that the library can do additional checking set up.

Application may call this function any time after init().

void **libRegisterWorkerThread**(const string & name)

Register a thread to poll for events.

This function should be called by an external worker thread, and it will block polling for events until the library is destroyed.

void **libStopWorkerThreads**()

Stop all worker threads.

int **libHandleEvents**(unsigned msec_timeout)

Poll pjsua for events, and if necessary block the caller thread for the specified maximum interval (in miliseconds).

Application doesn't normally need to call this function if it has configured worker thread (*thread_cnt* field) in pjsua_config structure, because polling then will be done by these worker threads instead.

If EpConfig::UaConfig::mainThreadOnly is enabled and this function is called from the main thread (by default the main thread is thread that calls *libCreate()*), this function will also scan and run any pending jobs in the list.

**Return**

The number of events that have been handled during the poll. Negative value indicates error, and application can retrieve the error as (status = -return_value).

**Parameters**

- `msec_timeout` - Maximum time to wait, in miliseconds.

void **libDestroy**(unsigned prmFlags = 0)

Destroy pjsua.

Application is recommended to perform graceful shutdown before calling this function (such as unregister the account from the SIP server, terminate presense subscription, and hangup active calls), however, this function will do all of these if it finds there are active sessions that need to be terminated. This function will block for few seconds to wait for replies from remote.

Application.may safely call this function more than once if it doesn't keep track of it's state.

**Parameters**

- `prmFlags` - Combination of pjsua_destroy_flag enumeration.

string **utilStrError**(pj_status_t prmErr)

Retrieve the error string for the specified status code.

**Parameters**

- `prmErr` - The error code.

void **utilLogWrite**(int prmLevel, const string & prmSender, const string & prmMsg)

Write a log message.

**Parameters**

- `prmLevel` - Log verbosity level (1-5)
- `prmSender` - The log sender.
- `prmMsg` - The log message.

void **utilLogWrite**(*LogEntry* & e)

Write a log entry.

**Parameters**

- `e` - The log entry.

pj_status_t **utilVerifySipUri**(const string & prmUri)

This is a utility function to verify that valid SIP url is given.

If the URL is a valid SIP/SIPS scheme, PJ_SUCCESS will be returned.

**Return**

PJ_SUCCESS on success, or the appropriate error code.

**See**

*utilVerifyUri()*

**Parameters**

- `prmUri` - The URL string.

pj_status_t **utilVerifyUri**(const string & prmUri)

This is a utility function to verify that valid URI is given.

Unlike *utilVerifySipUri()*, this function will return PJ_SUCCESS if tel: URI is given.

**Return**

PJ_SUCCESS on success, or the appropriate error code.

**See**

pjsua_verify_sip_url()

**Parameters**

- `prmUri` - The URL string.

*Token* **utilTimerSchedule**(unsigned prmMsecDelay, *Token* prmUserData)

Schedule a timer with the specified interval and user data.

When the interval elapsed, *onTimer()* callback will be called. Note that the callback may be executed by different thread, depending on whether worker thread is enabled or not.

**Return**

Token to identify the timer, which could be given to *utilTimerCancel()*.

**Parameters**

- `prmMsecDelay` - The time interval in msec.
- `prmUserData` - Arbitrary user data, to be given back to application in the callback.

void **utilTimerCancel**(*Token* prmToken)

Cancel previously scheduled timer with the specified timer token.

**Parameters**

- `prmToken` - The timer token, which was returned from previous *utilTimerSchedule()* call.

void **utilAddPendingJob**(*PendingJob* * job)

Utility to register a pending job to be executed by main thread.

If EpConfig::UaConfig::mainThreadOnly is false, the job will be executed immediately.

**Parameters**

- `job` - The job class.

*IntVector* **utilSslGetAvailableCiphers**()

Get cipher list supported by SSL/TLS backend.

void **natDetectType**(void)

This is a utility function to detect NAT type in front of this endpoint.

Once invoked successfully, this function will complete asynchronously and report the result in *onNatDetectionComplete()*.

After NAT has been detected and the callback is called, application can get the detected NAT type by calling *natGetType()*. Application can also perform NAT detection by calling *natDetectType()* again at later time.

Note that STUN must be enabled to run this function successfully.

pj_stun_nat_type **natGetType**()

Get the NAT type as detected by *natDetectType()* function.

This function will only return useful NAT type after *natDetectType()* has completed successfully and *onNatDetectionComplete()* callback has been called.

Exception: if this function is called while detection is in progress, PJ_EPENDING exception will be raised.

void **natCheckStunServers**(const *StringVector* & prmServers, bool prmWait, *Token* prmUserData)

Auxiliary function to resolve and contact each of the STUN server entries (sequentially) to find which is usable.

The *libInit()* must have been called before calling this function.

**See**

*natCancelCheckStunServers()*

**Parameters**

- `prmServers` - Array of STUN servers to try. The endpoint will try to resolve and contact each of the STUN server entry until it finds one that is usable. Each entry may be a domain name, host name, IP address, and it may contain an optional port number. For example:
- `prmWait` - Specify if the function should block until it gets the result. In this case, the function will block while the resolution is being done, and the callback will be called before this function returns.
- `prmUserData` - Arbitrary user data to be passed back to application in the callback.

void **natCancelCheckStunServers**(*Token* token, bool notify_cb = false)

Cancel pending STUN resolution which match the specified token.

Exception: PJ_ENOTFOUND if there is no matching one, or other error.

**Parameters**

- `token` - The token to match. This token was given to *natCheckStunServers()*
- `notify_cb` - Boolean to control whether the callback should be called for cancelled resolutions. When the callback is called, the status in the result will be set as PJ_ECANCELLED.

*TransportId* **transportCreate**(pjsip_transport_type_e type, const *TransportConfig* & cfg)

Create and start a new SIP transport according to the specified settings.

**Return**

The transport ID.

**Parameters**

- `type` - Transport type.
- `cfg` - Transport configuration.

*IntVector* **transportEnum**()

Enumerate all transports currently created in the system.

This function will return all transport IDs, and application may then call *transportGetInfo()* function to retrieve detailed information about the transport.

**Return**

Array of transport IDs.

*TransportInfo* **transportGetInfo**(*TransportId* id)

Get information about transport.

**Return**

Transport info.

**Parameters**

- `id` - Transport ID.

void **transportSetEnable**(*TransportId* id, bool enabled)

Disable a transport or re-enable it.

By default transport is always enabled after it is created. Disabling a transport does not necessarily close the socket, it will only discard incoming messages and prevent the transport from being used to send outgoing messages.

**Parameters**

- `id` - Transport ID.
- `enabled` - Enable or disable the transport.

void **transportClose**(*TransportId* id)

Close the transport.

The system will wait until all transactions are closed while preventing new users from using the transport, and will close the transport when its usage count reaches zero.

**Parameters**

- `id` - Transport ID.

void **hangupAllCalls**(void)

Terminate all calls.

This will initiate call hangup for all currently active calls.

void **mediaAdd**(*AudioMedia* & media)

Add media to the media list.

**Parameters**

- `media` - media to be added.

void **mediaRemove**(*AudioMedia* & media)

Remove media from the media list.

**Parameters**

- `media` - media to be removed.

bool **mediaExists**(const *AudioMedia* & media)

Check if media has been added to the media list.

**Return**

True if media has been added, false otherwise.

**Parameters**

- `media` - media to be check.

unsigned **mediaMaxPorts**()

Get maximum number of media port.

**Return**

Maximum number of media port in the conference bridge.

unsigned **mediaActivePorts**()

Get current number of active media port in the bridge.

**Return**

The number of active media port.

const *AudioMediaVector* & **mediaEnumPorts**()

Enumerate all media port.

**Return**

The list of media port.

*AudDevManager* & **audDevManager**()

Get the instance of Audio Device Manager.

**Return**

The Audio Device Manager.

const *CodecInfoVector* & **codecEnum**()

Enum all supported codecs in the system.

**Return**

Array of codec info.

void **codecSetPriority**(const string & codec_id, pj_uint8_t priority)

Change codec priority.

**Parameters**

- `codec_id` - Codec ID, which is a string that uniquely identify the codec (such as "speex/8000").
- `priority` - Codec priority, 0-255, where zero means to disable the codec.

*CodecParam* **codecGetParam**(const string & codec_id)

Get codec parameters.

**Return**

Codec parameters. If codec is not found, *Error* will be thrown.

**Parameters**

- `codec_id` - Codec ID.

void **codecSetParam**(const string & codec_id, const *CodecParam* param)

Set codec parameters.

**Parameters**

- `codec_id` - Codec ID.
- `param` - Codec parameter to set. Set to NULL to reset codec parameter to library default settings.

void **onNatDetectionComplete**(const *OnNatDetectionCompleteParam* & prm)

Callback when the *Endpoint* has finished performing NAT type detection that is initiated with *natDetectType()*.

**Parameters**

- `prm` - Callback parameters containing the detection result.

void **onNatCheckStunServersComplete**(const *OnNatCheckStunServersCompleteParam* & prm)

Callback when the *Endpoint* has finished performing STUN server checking that is initiated with *natCheckStunServers()*.

**Parameters**

- `prm` - Callback parameters.

void **onTransportState**(const *OnTransportStateParam* & prm)

This callback is called when transport state has changed.

**Parameters**

- `prm` - Callback parameters.

void **onTimer**(const *OnTimerParam* & prm)

Callback when a timer has fired.

The timer was scheduled by *utilTimerSchedule()*.

**Parameters**

- `prm` - Callback parameters.

void **onSelectAccount**(*OnSelectAccountParam* & prm)

This callback can be used by application to override the account to be used to handle an incoming message.

Initially, the account to be used will be calculated automatically by the library. This initial account will be used if application does not implement this callback, or application sets an invalid account upon returning from this callback.

Note that currently the incoming messages requiring account assignment are INVITE, MESSAGE, SUBSCRIBE, and unsolicited NOTIFY. This callback may be called before the callback of the SIP event itself, i.e: incoming call, pager, subscription, or unsolicited-event.

**Parameters**

- `prm` - Callback parameters.

*Public Static Functions*

*Endpoint* & **instance**()

Retrieve the singleton instance of the endpoint.

### 4.7.2 Endpoint Configurations

#### Endpoint

*struct* **pj::EpConfig**
*#include <endpoint.hpp>*

*Endpoint* configuration.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container.

**Parameters**

- `node` - Container to write values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container.

**Parameters**

- `node` - Container to write values to.

*Public Members*

*UaConfig* **uaConfig**

UA config.

*LogConfig* **logConfig**

Logging config.

*MediaConfig* **medConfig**

*Media* config.

### Media

*struct* **pj::MediaConfig**
*#include <endpoint.hpp>*

This structure describes media configuration, which will be specified when calling Lib::init().

*Public Functions*

**MediaConfig**()

Default constructor initialises with default values.

void **fromPj**(const pjsua_media_config & mc)

Construct from pjsua_media_config.

pjsua_media_config **toPj**()

Export.

void **readObject**(const *ContainerNode* & node)

Read this object from a container.

**Parameters**

- `node` - Container to write values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container.

**Parameters**

- `node` - Container to write values to.

*Public Members*

unsigned **clockRate**

Clock rate to be applied to the conference bridge.

If value is zero, default clock rate will be used (PJSUA_DEFAULT_CLOCK_RATE, which by default is 16KHz).

unsigned **sndClockRate**

Clock rate to be applied when opening the sound device.

If value is zero, conference bridge clock rate will be used.

unsigned **channelCount**

Channel count be applied when opening the sound device and conference bridge.

unsigned **audioFramePtime**

Specify audio frame ptime.

The value here will affect the samples per frame of both the sound device and the conference bridge. Specifying lower ptime will normally reduce the latency.

Default value: PJSUA_DEFAULT_AUDIO_FRAME_PTIME

unsigned **maxMediaPorts**

Specify maximum number of media ports to be created in the conference bridge.

Since all media terminate in the bridge (calls, file player, file recorder, etc), the value must be large enough to support all of them. However, the larger the value, the more computations are performed.

Default value: PJSUA_MAX_CONF_PORTS

bool **hasIoqueue**

Specify whether the media manager should manage its own ioqueue for the RTP/RTCP sockets.

If yes, ioqueue will be created and at least one worker thread will be created too. If no, the RTP/RTCP sockets will share the same ioqueue as SIP sockets, and no worker thread is needed.

Normally application would say yes here, unless it wants to run everything from a single thread.

unsigned **threadCnt**

Specify the number of worker threads to handle incoming RTP packets.

A value of one is recommended for most applications.

unsigned **quality**

*Media* quality, 0-10, according to this table: 5-10: resampling use large filter, 3-4: resampling use small filter, 1-2: resampling use linear.

The media quality also sets speex codec quality/complexity to the number.

Default: 5 (PJSUA_DEFAULT_CODEC_QUALITY).

unsigned **ptime**

Specify default codec ptime.

Default: 0 (codec specific)

bool **noVad**

Disable VAD?

Default: 0 (no (meaning VAD is enabled))

unsigned **ilbcMode**

iLBC mode (20 or 30).

Default: 30 (PJSUA_DEFAULT_ILBC_MODE)

unsigned **txDropPct**

Percentage of RTP packet to drop in TX direction (to simulate packet lost).

Default: 0

unsigned **rxDropPct**

Percentage of RTP packet to drop in RX direction (to simulate packet lost).

Default: 0

unsigned **ecOptions**

Echo canceller options (see pjmedia_echo_create())

Default: 0.

unsigned **ecTailLen**

Echo canceller tail length, in miliseconds.

Setting this to zero will disable echo cancellation.

Default: PJSUA_DEFAULT_EC_TAIL_LEN

unsigned **sndRecLatency**

Audio capture buffer length, in milliseconds.

Default: PJMEDIA_SND_DEFAULT_REC_LATENCY

unsigned **sndPlayLatency**

Audio playback buffer length, in milliseconds.

Default: PJMEDIA_SND_DEFAULT_PLAY_LATENCY

int **jbInit**

Jitter buffer initial prefetch delay in msec.

The value must be between jb_min_pre and jb_max_pre below.

Default: -1 (to use default stream settings, currently 150 msec)

int **jbMinPre**

Jitter buffer minimum prefetch delay in msec.

Default: -1 (to use default stream settings, currently 60 msec)

int **jbMaxPre**

Jitter buffer maximum prefetch delay in msec.

Default: -1 (to use default stream settings, currently 240 msec)

int **jbMax**

Set maximum delay that can be accomodated by the jitter buffer msec.

Default: -1 (to use default stream settings, currently 360 msec)

int **sndAutoCloseTime**

Specify idle time of sound device before it is automatically closed, in seconds.

Use value -1 to disable the auto-close feature of sound device

Default : 1

bool **vidPreviewEnableNative**

Specify whether built-in/native preview should be used if available.

In some systems, video input devices have built-in capability to show preview window of the device. Using this built-in preview is preferable as it consumes less CPU power. If built-in preview is not available, the library will perform software rendering of the input. If this field is set to PJ_FALSE, software preview will always be used.

Default: PJ_TRUE

### Logging

*struct* **pj::LogConfig**
*#include <endpoint.hpp>*

Logging configuration, which can be (optionally) specified when calling Lib::init().

*Public Functions*

**LogConfig**()

Default constructor initialises with default values.

void **fromPj**(const pjsua_logging_config & lc)

Construct from pjsua_logging_config.

pjsua_logging_config **toPj**()

Generate pjsua_logging_config.

void **readObject**(const *ContainerNode* & node)

Read this object from a container.

**Parameters**

- `node` - Container to write values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container.

**Parameters**

- `node` - Container to write values to.

*Public Members*

unsigned **msgLogging**

Log incoming and outgoing SIP message? Yes!

unsigned **level**

Input verbosity level.

Value 5 is reasonable.

unsigned **consoleLevel**

Verbosity level for console.

Value 4 is reasonable.

unsigned **decor**

Log decoration.

string **filename**

Optional log filename if app wishes the library to write to log file.

unsigned **fileFlags**

Additional flags to be given to pj_file_open() when opening the log file.

By default, the flag is PJ_O_WRONLY. Application may set PJ_O_APPEND here so that logs are appended to existing file instead of overwriting it.

Default is 0.

*LogWriter* * **writer**

Custom log writer, if required.

This instance will be destroyed by the endpoint when the endpoint is destroyed.

*class* **pj::LogWriter**

Interface for writing log messages.

Applications can inherit this class and supply it in the *LogConfig* structure to implement custom log writing facility.

*Public Functions*

**~LogWriter**()

Destructor.

void **write**(const *LogEntry* & entry)

Write a log entry.

*struct* **pj::LogEntry**
*#include <endpoint.hpp>*

Data containing log entry to be written by the *LogWriter*.

*Public Members*

int **level**

Log verbosity level of this message.

string **msg**

The log message.

long **threadId**

ID of current thread.

string **threadName**

The name of the thread that writes this log.

### User Agent

*struct* **pj::UaConfig**
*#include <endpoint.hpp>*

SIP User Agent related settings.

*Public Functions*

**UaConfig**()

Default constructor to initialize with default values.

void **fromPj**(const pjsua_config & ua_cfg)

Construct from pjsua_config.

pjsua_config **toPj**()

Export to pjsua_config.

void **readObject**(const *ContainerNode* & node)

Read this object from a container.

**Parameters**

- `node` - Container to write values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container.

**Parameters**

- `node` - Container to write values to.

*Public Members*

unsigned **maxCalls**

Maximum calls to support (default: 4).

The value specified here must be smaller than the compile time maximum settings PJSUA_MAX_CALLS, which by default is 32. To increase this limit, the library must be recompiled with new PJSUA_MAX_CALLS value.

unsigned **threadCnt**

Number of worker threads.

Normally application will want to have at least one worker thread, unless when it wants to poll the library periodically, which in this case the worker thread can be set to zero.

bool **mainThreadOnly**

When this flag is non-zero, all callbacks that come from thread other than main thread will be posted to the main thread and to be executed by *Endpoint::libHandleEvents()* function.

This includes the logging callback. Note that this will only work if threadCnt is set to zero and *Endpoint::libHandleEvents()* is performed by main thread. By default, the main thread is set from the thread that invoke *Endpoint::libCreate()*

Default: false

*StringVector* **nameserver**

Array of nameservers to be used by the SIP resolver subsystem.

The order of the name server specifies the priority (first name server will be used first, unless it is not reachable).

string **userAgent**

Optional user agent string (default empty).

If it's empty, no User-Agent header will be sent with outgoing requests.

*StringVector* **stunServer**

Array of STUN servers to try.

The library will try to resolve and contact each of the STUN server entry until it finds one that is usable. Each entry may be a domain name, host name, IP address, and it may contain an optional port number. For example:

When nameserver is configured in the *pjsua_config.nameserver* field, if entry is not an IP address, it will be resolved with DNS SRV resolution first, and it will fallback to use DNS A resolution if this fails. Port number may be specified even if the entry is a domain name, in case the DNS SRV resolution should fallback to a non-standard port.

When nameserver is not configured, entries will be resolved with pj_gethostbyname() if it's not an IP address. Port number may be specified if the server is not listening in standard STUN port.

bool **stunIgnoreFailure**

This specifies if the library startup should ignore failure with the STUN servers.

If this is set to PJ_FALSE, the library will refuse to start if it fails to resolve or contact any of the STUN servers.

Default: TRUE

int **natTypeInSdp**

Support for adding and parsing NAT type in the SDP to assist troubleshooting.

The valid values are:

Default: 1

bool **mwiUnsolicitedEnabled**

Handle unsolicited NOTIFY requests containing message waiting indication (MWI) info.

Unsolicited MWI is incoming NOTIFY requests which are not requested by client with SUBSCRIBE request.

If this is enabled, the library will respond 200/OK to the NOTIFY request and forward the request to Endpoint::onMwiInfo() callback.

See also *AccountMwiConfig.enabled*.

Default: PJ_TRUE

### 4.7.3 Callback Parameters

*struct* **pj::OnNatDetectionCompleteParam**
*#include <endpoint.hpp>*

Argument to *Endpoint::onNatDetectionComplete()* callback.

*Public Members*

pj_status_t **status**

Status of the detection process.

If this value is not PJ_SUCCESS, the detection has failed and *nat_type* field will contain PJ_STUN_NAT_TYPE_UNKNOWN.

string **reason**

The text describing the status, if the status is not PJ_SUCCESS.

pj_stun_nat_type **natType**

This contains the NAT type as detected by the detection procedure.

This value is only valid when the *status* is PJ_SUCCESS.

string **natTypeName**

Text describing that NAT type.

*struct* **pj::OnNatCheckStunServersCompleteParam**
*#include <endpoint.hpp>*

Argument to *Endpoint::onNatCheckStunServersComplete()* callback.

*Public Members*

*Token* **userData**

Arbitrary user data that was passed to *Endpoint::natCheckStunServers()* function.

pj_status_t **status**

This will contain PJ_SUCCESS if at least one usable STUN server is found, otherwise it will contain the last error code during the operation.

string **name**

The server name that yields successful result.

This will only contain value if status is successful.

*SocketAddress* **addr**

The server IP address and port in "IP:port" format.

This will only contain value if status is successful.

*struct* **pj::OnTimerParam**
*#include <endpoint.hpp>*

Parameter of *Endpoint::onTimer()* callback.

*Public Members*

*Token* **userData**

Arbitrary user data that was passed to *Endpoint::utilTimerSchedule()* function.

unsigned **msecDelay**

The interval of this timer, in miliseconds.

*struct* **pj::OnTransportStateParam**
*#include <endpoint.hpp>*

Parameter of *Endpoint::onTransportState()* callback.

*Public Members*

*TransportHandle* **hnd**

The transport handle.

pjsip_transport_state **state**

Transport current state.

pj_status_t **lastError**

The last error code related to the transport state.

*struct* **pj::OnSelectAccountParam**
*#include <endpoint.hpp>*

Parameter of *Endpoint::onSelectAccount()* callback.

*Public Members*

*SipRxData* **rdata**

The incoming request.

int **accountIndex**

The account index to be used to handle the request.

Upon entry, this will be filled by the account index chosen by the library. Application may change it to another value to use another account.

### 4.7.4 Other

*struct* **pj::PendingJob**

*Public Functions*

void **execute**(bool is_pending)

Perform the job.

**~PendingJob**()

Virtual destructor.

CHAPTER

FIVE

# ACCOUNTS

Accounts provide identity (or identities) of the user who is currently using the application. An account has one SIP Uniform Resource Identifier (URI) associated with it. In SIP terms, this URI acts as Address of Record (AOR) of the person and is used as the From header in outgoing requests.

Account may or may not have client registration associated with it. An account is also associated with route set and some authentication credentials, which are used when sending SIP request messages using the account. An account also has presence status, which will be reported to remote peer when they subscribe to the account's presence, or which is published to a presence server if presence publication is enabled for the account.

At least one account MUST be created in the application, since any outgoing requests require an account context. If no user association is required, application can create a userless account by calling Account.create(). A userless account identifies local endpoint instead of a particular user, and it corresponds to a particular transport ID.

Also one account must be set as the default account, which will be used as the account identity when pjsua fails to match incoming request with any accounts using the stricter matching rules.

## 5.1 Subclassing the Account class

To use the Account class, normally application SHOULD create its own subclass, in order to receive notifications for the account. For example:

```
class MyAccount : public Account
{
public:
    MyAccount() {}
    ~MyAccount() {}

    virtual void onRegState(OnRegStateParam &prm)
    {
        AccountInfo ai = getInfo();
        cout << (ai.regIsActive? "*** Register: code=" : "*** Unregister: code=")
             << prm.code << endl;
    }

    virtual void onIncomingCall(OnIncomingCallParam &iprm)
    {
        Call *call = new MyCall(*this, iprm.callId);

        // Just hangup for now
        CallOpParam op;
        op.statusCode = PJSIP_SC_DECLINE;
        call->hangup(op);
```

```
        // And delete the call
        delete call;
    }
};
```

In its subclass, application can implement the account callbacks, which is basically used to process events related to the account, such as:

- the status of SIP registration
- incoming calls
- incoming presence subscription requests
- incoming instant message not from buddy

Application needs to override the relevant callback methods in the derived class to handle these particular events.

If the events are not handled, default actions will be invoked:

- incoming calls will not be handled
- incoming presence subscription requests will be accepted
- incoming instant messages from non-buddy will be ignored

## 5.2 Creating Userless Accounts

A userless account identifies a particular SIP endpoint rather than a particular user. Some other SIP softphones may call this peer-to-peer mode, which means that we are calling another computer via its address rather than calling a particular user ID. For example, we might identify ourselves as "sip:192.168.0.15" (a userless account) rather than, say, "sip:alice@pjsip.org".

In the lower layer PJSUA-LIB API, a userless account is associated with a SIP transport, and is created with `pjsua_acc_add_local()` API. This concept has been deprecated in PJSUA2, and rather, a userless account is a "normal" account with a userless ID URI (e.g. "sip:192.168.0.15") and without registration. Thus creating a userless account is exactly the same as creating "normal" account.

## 5.3 Creating Account

We need to configure AccountConfig and call Account.create() to create the account. At the very minimum, pjsua only requires the account's ID, which is an URI to identify the account (or in SIP terms, it's called Address of Record/AOR). Here's a snippet:

```
AccountConfig acc_cfg;
acc_cfg.idUri = "sip:test1@pjsip.org";

MyAccount *acc = new MyAccount;
try {
    acc->create(acc_cfg);
} catch(Error& err) {
    cout << "Account creation error: " << err.info() << endl;
}
```

The account created above doesn't do anything except to provide identity in the "From:" header for outgoing requests. The account will not register to SIP server or anything.

Typically you will want the account to authenticate and register to your SIP server so that you can receive incoming calls. To do that you will need to configure some more settings in your AccountConfig, something like this:

```
AccountConfig acc_cfg;
acc_cfg.idUri = "sip:test1@pjsip.org";
acc_cfg.regConfig.registrarUri = "sip:pjsip.org";
acc_cfg.sipConfig.authCreds.push_back( AuthCredInfo("digest", "*", "test1", 0, "secret1") );

MyAccount *acc = new MyAccount;
try {
    acc->create(acc_cfg);
} catch(Error& err) {
    cout << "Account creation error: " << err.info() << endl;
}
```

## 5.4 Account Configurations

There are many more settings that can be specified in AccountConfig, like:

- AccountRegConfig, to specify registration settings, such as registrar server and retry interval.
- AccountSipConfig, to specify SIP settings, such as credential information and proxy server.
- AccountCallConfig, to specify call settings, such as whether reliable provisional response (SIP 100rel) is required.
- AccountPresConfig, to specify presence settings, such as whether presence publication (PUBLISH) is enabled.
- AccountMwiConfig, to specify MWI (Message Waiting Indication) settings.
- AccountNatConfig, to specify NAT settings, such as whether STUN or ICE is used.
- AccountMediaConfig, to specify media settings, such as Secure RTP (SRTP) related settings.
- AccountVideoConfig, to specify video settings, such as default capture and render device.

Please see AccountConfig reference documentation for more info.

## 5.5 Account Operations

Some of the operations to the Account object:

- manage registration
- manage buddies/contacts
- manage presence online status

Please see the reference documentation for Account for more info. Calls, presence, and buddy will be explained in later chapters.

## 5.6 Class Reference

### 5.6.1 Account

*class* **pj::Account**

*Account*.

*Public Functions*

**Account**()

Constructor.

**~Account**()

Destructor.

Note that if the account is deleted, it will also delete the corresponding account in the PJSUA-LIB.

void **create**(const *AccountConfig* & cfg, bool make_default = false)

Create the account.

**Parameters**

- `cfg` - The account config.
- `make_default` - Make this the default account.

void **modify**(const *AccountConfig* & cfg)

Modify the account to use the specified account configuration.

Depending on the changes, this may cause unregistration or reregistration on the account.

**Parameters**

- `cfg` - New account config to be applied to the account.

bool **isValid**()

Check if this account is still valid.

**Return**

True if it is.

void **setDefault**()

Set this as default account to be used when incoming and outgoing requests don't match any accounts.

**Return**

PJ_SUCCESS on success.

bool **isDefault**()

Check if this account is the default account.

Default account will be used for incoming and outgoing requests that don't match any other accounts.

**Return**

True if this is the default account.

int **getId**()

Get PJSUA-LIB account ID or index associated with this account.

**Return**

Integer greater than or equal to zero.

*AccountInfo* **getInfo**()

Get account info.

**Return**

*Account* info.

void **setRegistration**(bool renew)

Update registration or perform unregistration.

Application normally only needs to call this function if it wants to manually update the registration or to unregister from the server.

**Parameters**

- `renew` - If False, this will start unregistration process.

void **setOnlineStatus**(const *PresenceStatus* & pres_st)

Set or modify account's presence online status to be advertised to remote/presence subscribers.

This would trigger the sending of outgoing NOTIFY request if there are server side presence subscription for this account, and/or outgoing PUBLISH if presence publication is enabled for this account.

**Parameters**

- `pres_st` - Presence online status.

void **setTransport**(*TransportId* tp_id)

Lock/bind this account to a specific transport/listener.

Normally application shouldn't need to do this, as transports will be selected automatically by the library according to the destination.

When account is locked/bound to a specific transport, all outgoing requests from this account will use the specified transport (this includes SIP registration, dialog (call and event subscription), and out-of-dialog requests such as MESSAGE).

Note that transport id may be specified in *AccountConfig* too.

**Parameters**

- `tp_id` - The transport ID.

void **presNotify**(const *PresNotifyParam* & prm)

Send NOTIFY to inform account presence status or to terminate server side presence subscription.

If application wants to reject the incoming request, it should set the param PresNotifyParam.state to PJSIP_EVSUB_STATE_TERMINATED.

**Parameters**

- `prm` - The sending NOTIFY parameter.

const *BuddyVector* & **enumBuddies**()

Enumerate all buddies of the account.

**Return**

The buddy list.

*Buddy* * **findBuddy**(string uri, *FindBuddyMatch* * buddy_match = NULL)

Find a buddy in the buddy list with the specified URI.

Exception: if buddy is not found, PJ_ENOTFOUND will be thrown.

**Return**

The pointer to buddy.

**Parameters**

- `uri` - The buddy URI.
- `buddy_match` - The buddy match algo.

void **addBuddy**(*Buddy* * buddy)

An internal function to add a *Buddy* to *Account* buddy list.

This function must never be used by application.

void **removeBuddy**(*Buddy* * buddy)

An internal function to remove a *Buddy* from *Account* buddy list.

This function must never be used by application.

void **onIncomingCall**(*OnIncomingCallParam* & prm)

Notify application on incoming call.

**Parameters**

- `prm` - Callback parameter.

void **onRegStarted**(*OnRegStartedParam* & prm)

Notify application when registration or unregistration has been initiated.

Note that this only notifies the initial registration and unregistration. Once registration session is active, subsequent refresh will not cause this callback to be called.

**Parameters**

- `prm` - Callback parameter.

void **onRegState**(*OnRegStateParam* & prm)

Notify application when registration status has changed.

Application may then query the account info to get the registration details.

**Parameters**

- `prm` - Callback parameter.

void **onIncomingSubscribe**(*OnIncomingSubscribeParam* & prm)

Notification when incoming SUBSCRIBE request is received.

Application may use this callback to authorize the incoming subscribe request (e.g. ask user permission if the request should be granted).

If this callback is not implemented, all incoming presence subscription requests will be accepted.

If this callback is implemented, application has several choices on what to do with the incoming request:

Any IncomingSubscribeParam.code other than 200 and 202 will be treated as 200.

Application MUST return from this callback immediately (e.g. it must not block in this callback while waiting for user confirmation).

**Parameters**

- `prm` - Callback parameter.

void **onInstantMessage**(*OnInstantMessageParam* & prm)

Notify application on incoming instant message or pager (i.e.

MESSAGE request) that was received outside call context.

**Parameters**

- `prm` - Callback parameter.

void **onInstantMessageStatus**(*OnInstantMessageStatusParam* & prm)

Notify application about the delivery status of outgoing pager/instant message (i.e.

MESSAGE) request.

**Parameters**

- `prm` - Callback parameter.

void **onTypingIndication**(*OnTypingIndicationParam* & prm)

Notify application about typing indication.

**Parameters**

- `prm` - Callback parameter.

void **onMwiInfo**(*OnMwiInfoParam* & prm)

Notification about MWI (Message Waiting Indication) status change.

This callback can be called upon the status change of the SUBSCRIBE request (for example, 202/Accepted to SUBSCRIBE is received) or when a NOTIFY reqeust is received.

**Parameters**

- `prm` - Callback parameter.

*Public Static Functions*

*Account* * **lookup**(int acc_id)

Get the *Account* class for the specified account Id.

**Return**

The *Account* instance or NULL if not found.

**Parameters**

- `acc_id` - The account ID to lookup

### 5.6.2 AccountInfo

*struct* **pj::AccountInfo**
*#include <account.hpp>*

*Account* information.

Application can query the account information by calling *Account::getInfo()*.

*Public Functions*

void **fromPj**(const pjsua_acc_info & pai)

Import from pjsip data.

*Public Members*

pjsua_acc_id **id**

The account ID.

bool **isDefault**

Flag to indicate whether this is the default account.

string **uri**

*Account* URI.

bool **regIsConfigured**

Flag to tell whether this account has registration setting (reg_uri is not empty).

bool **regIsActive**

Flag to tell whether this account is currently registered (has active registration session).

int **regExpiresSec**

An up to date expiration interval for account registration session.

pjsip_status_code **regStatus**

Last registration status code.

If status code is zero, the account is currently not registered. Any other value indicates the SIP status code of the registration.

string **regStatusText**

String describing the registration status.

pj_status_t **regLastErr**

Last registration error code.

When the status field contains a SIP status code that indicates a registration failure, last registration error code contains the error code that causes the failure. In any other case, its value is zero.

bool **onlineStatus**

Presence online status for this account.

string **onlineStatusText**

Presence online status text.

### 5.6.3 Account Settings

#### AccountConfig

*struct* **pj::AccountConfig**
*#include <account.hpp>*

*Account* configuration.

*Public Functions*

**AccountConfig**()

Default constructor will initialize with default values.

void **toPj**(pjsua_acc_config & cfg)

This will return a temporary pjsua_acc_config instance, which contents are only valid as long as this *AccountConfig* structure remains valid AND no modifications are done to it AND no further *toPj()* function call is made.

Any call to *toPj()* function will invalidate the content of temporary pjsua_acc_config that was returned by the previous call.

void **fromPj**(const pjsua_acc_config & prm, const pjsua_media_config * mcfg)

Initialize from pjsip.

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

int **priority**

*Account* priority, which is used to control the order of matching incoming/outgoing requests.

The higher the number means the higher the priority is, and the account will be matched first.

string **idUri**

The Address of Record or AOR, that is full SIP URL that identifies the account.

The value can take name address or URL format, and will look something like "sip:account@serviceprovider".

This field is mandatory.

*AccountRegConfig* **regConfig**

Registration settings.

*AccountSipConfig* **sipConfig**

SIP settings.

*AccountCallConfig* **callConfig**

*Call* settings.

*AccountPresConfig* **presConfig**

Presence settings.

*AccountMwiConfig* **mwiConfig**

MWI (Message Waiting Indication) settings.

*AccountNatConfig* **natConfig**

NAT settings.

*AccountMediaConfig* **mediaConfig**

*Media* settings (applicable for both audio and video).

*AccountVideoConfig* **videoConfig**

Video settings.

### AccoutRegConfig

*struct* **pj::AccountRegConfig**
*#include <account.hpp>*

*Account* registration config.

This will be specified in *AccountConfig*.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

string **registrarUri**

This is the URL to be put in the request URI for the registration, and will look something like "sip:serviceprovider".

This field should be specified if registration is desired. If the value is empty, no account registration will be performed.

bool **registerOnAdd**

Specify whether the account should register as soon as it is added to the UA.

Application can set this to PJ_FALSE and control the registration manually with pjsua_acc_set_registration().

Default: True

*SipHeaderVector* **headers**

The optional custom SIP headers to be put in the registration request.

unsigned **timeoutSec**

Optional interval for registration, in seconds.

If the value is zero, default interval will be used (PJSUA_REG_INTERVAL, 300 seconds).

unsigned **retryIntervalSec**

Specify interval of auto registration retry upon registration failure (including caused by transport problem), in second.

Set to 0 to disable auto re-registration. Note that if the registration retry occurs because of transport failure, the first retry will be done after *firstRetryIntervalSec* seconds instead. Also note that the interval will be randomized slightly by approximately +/- ten seconds to avoid all clients re-registering at the same time.

See also *firstRetryIntervalSec* setting.

Default: PJSUA_REG_RETRY_INTERVAL

unsigned **firstRetryIntervalSec**

This specifies the interval for the first registration retry.

The registration retry is explained in *retryIntervalSec*. Note that the value here will also be randomized by +/- ten seconds.

Default: 0

unsigned **delayBeforeRefreshSec**

Specify the number of seconds to refresh the client registration before the registration expires.

Default: PJSIP_REGISTER_CLIENT_DELAY_BEFORE_REFRESH, 5 seconds

bool **dropCallsOnFail**

Specify whether calls of the configured account should be dropped after registration failure and an attempt of re-registration has also failed.

Default: FALSE (disabled)

unsigned **unregWaitSec**

Specify the maximum time to wait for unregistration requests to complete during library shutdown sequence.

Default: PJSUA_UNREG_TIMEOUT

unsigned **proxyUse**

Specify how the registration uses the outbound and account proxy settings.

This controls if and what Route headers will appear in the REGISTER request of this account. The value is bitmask combination of PJSUA_REG_USE_OUTBOUND_PROXY and PJSUA_REG_USE_ACC_PROXY bits. If the value is set to 0, the REGISTER request will not use any proxy (i.e. it will not have any Route headers).

Default: 3 (PJSUA_REG_USE_OUTBOUND_PROXY | PJSUA_REG_USE_ACC_PROXY)

### AccountSipConfig

*struct* **pj::AccountSipConfig**
*#include <account.hpp>*

Various SIP settings for the account.

This will be specified in *AccountConfig*.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

*AuthCredInfoVector* **authCreds**

Array of credentials.

If registration is desired, normally there should be at least one credential specified, to successfully authenticate against the service provider. More credentials can be

specified, for example when the requests are expected to be challenged by the proxies in the route set.

*StringVector* **proxies**

Array of proxy servers to visit for outgoing requests.

Each of the entry is translated into one Route URI.

string **contactForced**

Optional URI to be put as Contact for this account.

It is recommended that this field is left empty, so that the value will be calculated automatically based on the transport address.

string **contactParams**

Additional parameters that will be appended in the Contact header for this account.

This will affect the Contact header in all SIP messages sent on behalf of this account, including but not limited to REGISTER, INVITE, and SUBCRIBE requests or responses.

The parameters should be preceeded by semicolon, and all strings must be properly escaped. Example: ";my-param=X;another-param=Hi%20there"

string **contactUriParams**

Additional URI parameters that will be appended in the Contact URI for this account.

This will affect the Contact URI in all SIP messages sent on behalf of this account, including but not limited to REGISTER, INVITE, and SUBCRIBE requests or responses.

The parameters should be preceeded by semicolon, and all strings must be properly escaped. Example: ";my-param=X;another-param=Hi%20there"

bool **authInitialEmpty**

If this flag is set, the authentication client framework will send an empty Authorization header in each initial request.

Default is no.

string **authInitialAlgorithm**

Specify the algorithm to use when empty Authorization header is to be sent for each initial request (see above)

*TransportId* **transportId**

Optionally bind this account to specific transport.

This normally is not a good idea, as account should be able to send requests using any available transports according to the destination. But some application may want to have explicit control over the transport to use, so in that case it can set this field.

Default: -1 (PJSUA_INVALID_ID)

**See**

*Account::setTransport()*

### AccountCallConfig

*struct* **pj::AccountCallConfig**
*#include <account.hpp>*

*Account*'s call settings.

This will be specified in *AccountConfig*.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

pjsua_call_hold_type **holdType**

Specify how to offer call hold to remote peer.

Please see the documentation on pjsua_call_hold_type for more info.

Default: PJSUA_CALL_HOLD_TYPE_DEFAULT

pjsua_100rel_use **prackUse**

Specify how support for reliable provisional response (100rel/ PRACK) should be used for all sessions in this account.

See the documentation of pjsua_100rel_use enumeration for more info.

Default: PJSUA_100REL_NOT_USED

pjsua_sip_timer_use **timerUse**

Specify the usage of Session Timers for all sessions.

See the pjsua_sip_timer_use for possible values.

Default: PJSUA_SIP_TIMER_OPTIONAL

unsigned **timerMinSESec**

Specify minimum Session Timer expiration period, in seconds.

Must not be lower than 90. Default is 90.

unsigned **timerSessExpiresSec**

Specify Session Timer expiration period, in seconds.

Must not be lower than timerMinSE. Default is 1800.

### AccountPresConfig

*struct* **pj::AccountPresConfig**
*#include <account.hpp>*

*Account* presence config.

This will be specified in *AccountConfig*.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

*SipHeaderVector* **headers**

The optional custom SIP headers to be put in the presence subscription request.

bool **publishEnabled**

If this flag is set, the presence information of this account will be PUBLISH-ed to the server where the account belongs.

Default: PJ_FALSE

bool **publishQueue**

Specify whether the client publication session should queue the PUBLISH request should there be another PUBLISH transaction still pending.

If this is set to false, the client will return error on the PUBLISH request if there is another PUBLISH transaction still in progress.

Default: PJSIP_PUBLISHC_QUEUE_REQUEST (TRUE)

unsigned **publishShutdownWaitMsec**

Maximum time to wait for unpublication transaction(s) to complete during shutdown process, before sending unregistration.

The library tries to wait for the unpublication (un-PUBLISH) to complete before sending REGISTER request to unregister the account, during library shutdown process. If the value is set too short, it is possible that the unregistration is sent before unpublication completes, causing unpublication request to fail.

Value is in milliseconds.

Default: PJSUA_UNPUBLISH_MAX_WAIT_TIME_MSEC (2000)

string **pidfTupleId**

Optional PIDF tuple ID for outgoing PUBLISH and NOTIFY.

If this value is not specified, a random string will be used.

### AccountMwiConfig

*struct* **pj::AccountMwiConfig**
*#include <account.hpp>*

*Account* MWI (Message Waiting Indication) settings.

This will be specified in *AccountConfig*.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

bool **enabled**

Subscribe to message waiting indication events (RFC 3842).

See also *UaConfig.mwiUnsolicitedEnabled* setting.

Default: FALSE

unsigned **expirationSec**

Specify the default expiration time (in seconds) for Message Waiting Indication (RFC 3842) event subscription.

This must not be zero.

Default: PJSIP_MWI_DEFAULT_EXPIRES (3600)

### AccountNatConfig

*struct* **pj::AccountNatConfig**
*#include <account.hpp>*

*Account*'s NAT (Network Address Translation) settings.

This will be specified in *AccountConfig*.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

pjsua_stun_use **sipStunUse**

Control the use of STUN for the SIP signaling.

Default: PJSUA_STUN_USE_DEFAULT

pjsua_stun_use **mediaStunUse**

Control the use of STUN for the media transports.

Default: PJSUA_STUN_USE_DEFAULT

bool **iceEnabled**

Enable ICE for the media transport.

Default: False

int **iceMaxHostCands**

Set the maximum number of ICE host candidates.

Default: -1 (maximum not set)

bool **iceAggressiveNomination**

Specify whether to use aggressive nomination.

Default: True

unsigned **iceNominatedCheckDelayMsec**

For controlling agent if it uses regular nomination, specify the delay to perform nominated check (connectivity check with USE-CANDIDATE attribute) after all components have a valid pair.

Default value is PJ_ICE_NOMINATED_CHECK_DELAY.

int **iceWaitNominationTimeoutMsec**

For a controlled agent, specify how long it wants to wait (in milliseconds) for the controlling agent to complete sending connectivity check with nominated flag set to true for all components after the controlled agent has found that all connectivity checks in its checklist have been completed and there is at least one successful (but not nominated) check for every component.

Default value for this option is ICE_CONTROLLED_AGENT_WAIT_NOMINATION_TIMEOUT. Specify -1 to disable this timer.

bool **iceNoRtcp**

Disable RTCP component.

Default: False

bool **iceAlwaysUpdate**

Always send re-INVITE/UPDATE after ICE negotiation regardless of whether the default ICE transport address is changed or not.

When this is set to False, re-INVITE/UPDATE will be sent only when the default ICE transport address is changed.

Default: yes

bool **turnEnabled**

Enable TURN candidate in ICE.

string **turnServer**

Specify TURN domain name or host name, in in "DOMAIN:PORT" or "HOST:PORT" format.

pj_turn_tp_type **turnConnType**

Specify the connection type to be used to the TURN server.

Valid values are PJ_TURN_TP_UDP or PJ_TURN_TP_TCP.

Default: PJ_TURN_TP_UDP

string **turnUserName**

Specify the username to authenticate with the TURN server.

int **turnPasswordType**

Specify the type of password.

Currently this must be zero to indicate plain-text password will be used in the password.

string **turnPassword**

Specify the password to authenticate with the TURN server.

int **contactRewriteUse**

This option is used to update the transport address and the Contact header of REGISTER request.

When this option is enabled, the library will keep track of the public IP address from the response of REGISTER request. Once it detects that the address has changed, it will unregister current Contact, update the Contact with transport address learned from Via header, and register a new Contact to the registrar. This will also update the public name of UDP transport if STUN is configured.

See also contactRewriteMethod field.

Default: TRUE

int **contactRewriteMethod**

Specify how Contact update will be done with the registration, if *contactRewriteEnabled* is enabled.

The value is bitmask combination of *pjsua_contact_rewrite_method*. See also pjsua_contact_rewrite_method.

Value PJSUA_CONTACT_REWRITE_UNREGISTER(1) is the legacy behavior.

Default value: PJSUA_CONTACT_REWRITE_METHOD (PJSUA_CONTACT_REWRITE_NO_UNREG | PJSUA_CONTACT_REWRITE_ALWAYS_UPDATE)

int **viaRewriteUse**

This option is used to overwrite the "sent-by" field of the Via header for outgoing messages with the same interface address as the one in the REGISTER request, as long as the request uses the same transport instance as the previous REGISTER request.

Default: TRUE

int **sdpNatRewriteUse**

This option controls whether the IP address in SDP should be replaced with the IP address found in Via header of the REGISTER response, ONLY when STUN and ICE are not used.

If the value is FALSE (the original behavior), then the local IP address will be used. If TRUE, and when STUN and ICE are disabled, then the IP address found in registration response will be used.

Default: PJ_FALSE (no)

int **sipOutboundUse**

Control the use of SIP outbound feature.

SIP outbound is described in RFC 5626 to enable proxies or registrar to send inbound requests back to UA using the same connection initiated by the UA for its registration. This feature is highly useful in NAT-ed deployemtns, hence it is enabled by default.

Note: currently SIP outbound can only be used with TCP and TLS transports. If UDP is used for the registration, the SIP outbound feature will be silently ignored for the account.

Default: TRUE

string **sipOutboundInstanceId**

Specify SIP outbound (RFC 5626) instance ID to be used by this account.

If empty, an instance ID will be generated based on the hostname of this agent. If application specifies this parameter, the value will look like "<urn:uuid:00000000-0000-1000-8000-AABBCCDDEEFF>" without the double-quotes.

Default: empty

string **sipOutboundRegId**

Specify SIP outbound (RFC 5626) registration ID.

The default value is empty, which would cause the library to automatically generate a suitable value.

Default: empty

unsigned **udpKaIntervalSec**

Set the interval for periodic keep-alive transmission for this account.

If this value is zero, keep-alive will be disabled for this account. The keep-alive transmission will be sent to the registrar's address, after successful registration.

Default: 15 (seconds)

string **udpKaData**

Specify the data to be transmitted as keep-alive packets.

Default: CR-LF

### AccountMediaConfig

*struct* **pj::AccountMediaConfig**
*#include <account.hpp>*

*Account* media config (applicable for both audio and video).

This will be specified in *AccountConfig*.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

*TransportConfig* **transportConfig**

*Media* transport (RTP) configuration.

bool **lockCodecEnabled**

If remote sends SDP answer containing more than one format or codec in the media line, send re-INVITE or UPDATE with just one codec to lock which codec to use.

Default: True (Yes).

bool **streamKaEnabled**

Specify whether stream keep-alive and NAT hole punching with non-codec-VAD mechanism (see PJMEDIA_STREAM_ENABLE_KA) is enabled for this account.

Default: False

pjmedia_srtp_use **srtpUse**

Specify whether secure media transport should be used for this account.

Valid values are PJMEDIA_SRTP_DISABLED, PJMEDIA_SRTP_OPTIONAL, and PJMEDIA_SRTP_MANDATORY.

Default: PJSUA_DEFAULT_USE_SRTP

int **srtpSecureSignaling**

Specify whether SRTP requires secure signaling to be used.

This option is only used when *use_srtp* option above is non-zero.

Valid values are: 0: SRTP does not require secure signaling 1: SRTP requires secure transport such as TLS 2: SRTP requires secure end-to-end transport (SIPS)

Default: PJSUA_DEFAULT_SRTP_SECURE_SIGNALING

pjsua_ipv6_use **ipv6Use**

Specify whether IPv6 should be used on media.

Default is not used.

### AccountVideoConfig

*struct* **pj::AccountVideoConfig**
*#include <account.hpp>*

*Account* video config.

This will be specified in *AccountConfig*.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

bool **autoShowIncoming**

Specify whether incoming video should be shown to screen by default.

This applies to incoming call (INVITE), incoming re-INVITE, and incoming UPDATE requests.

Regardless of this setting, application can detect incoming video by implementing *on_call_media_state()* callback and enumerating the media stream(s) with pjsua_call_get_info(). Once incoming video is recognised, application may retrieve the window associated with the incoming video and show or hide it with pjsua_vid_win_set_show().

Default: False

bool **autoTransmitOutgoing**

Specify whether outgoing video should be activated by default when making outgoing calls and/or when incoming video is detected.

This applies to incoming and outgoing calls, incoming re-INVITE, and incoming UPDATE. If the setting is non-zero, outgoing video transmission will be started as soon as response to these requests is sent (or received).

Regardless of the value of this setting, application can start and stop outgoing video transmission with pjsua_call_set_vid_strm().

Default: False

unsigned **windowFlags**

Specify video window's flags.

The value is a bitmask combination of pjmedia_vid_dev_wnd_flag.

Default: 0

pjmedia_vid_dev_index **defaultCaptureDevice**

Specify the default capture device to be used by this account.

If vidOutAutoTransmit is enabled, this device will be used for capturing video.

Default: PJMEDIA_VID_DEFAULT_CAPTURE_DEV

pjmedia_vid_dev_index **defaultRenderDevice**

Specify the default rendering device to be used by this account.

Default: PJMEDIA_VID_DEFAULT_RENDER_DEV

pjmedia_vid_stream_rc_method **rateControlMethod**

Rate control method.

Default: PJMEDIA_VID_STREAM_RC_SIMPLE_BLOCKING.

unsigned **rateControlBandwidth**

Upstream/outgoing bandwidth.

If this is set to zero, the video stream will use codec maximum bitrate setting.

Default: 0 (follow codec maximum bitrate).

### 5.6.4 Callback Parameters

*struct* **pj::OnIncomingCallParam**
*#include <account.hpp>*

This structure contains parameters for onIncomingCall() account callback.

*Public Members*

int **callId**

The library call ID allocated for the new call.

*SipRxData* **rdata**

The incoming INVITE request.

*struct* **pj::OnRegStartedParam**
*#include <account.hpp>*

This structure contains parameters for onRegStarted() account callback.

*Public Members*

bool **renew**

True for registration and False for unregistration.

*struct* **pj::OnRegStateParam**
*#include <account.hpp>*

This structure contains parameters for onRegState() account callback.

*Public Members*

pj_status_t **status**

Registration operation status.

pjsip_status_code **code**

SIP status code received.

string **reason**

SIP reason phrase received.

*SipRxData* **rdata**

The incoming message.

int **expiration**

Next expiration interval.

*struct* **pj::OnIncomingSubscribeParam**
*#include <account.hpp>*

This structure contains parameters for onIncomingSubscribe() callback.

*Public Members*

void * **srvPres**

Server presence subscription instance.

If application delays the acceptance of the request, it will need to specify this object when calling *Account::presNotify()*.

string **fromUri**

Sender URI.

*SipRxData* **rdata**

The incoming message.

pjsip_status_code **code**

The status code to respond to the request.

The default value is 200. Application may set this to other final status code to accept or reject the request.

string **reason**

The reason phrase to respond to the request.

*SipTxOption* **txOption**

Additional data to be sent with the response, if any.

*struct* **pj::OnInstantMessageParam**
*#include <account.hpp>*

Parameters for onInstantMessage() account callback.

*Public Members*

string **fromUri**

Sender From URI.

string **toUri**

To URI of the request.

string **contactUri**

Contact URI of the sender.

string **contentType**

MIME type of the message body.

string **msgBody**

The message body.

*SipRxData* **rdata**

The whole message.

*struct* **pj::OnInstantMessageStatusParam**
*#include <account.hpp>*

Parameters for onInstantMessageStatus() account callback.

*Public Members*

*Token* **userData**

Token or a user data that was associated with the pager transmission.

string **toUri**

Destination URI.

string **msgBody**

The message body.

pjsip_status_code **code**

The SIP status code of the transaction.

string **reason**

The reason phrase of the transaction.

*SipRxData* **rdata**

The incoming response that causes this callback to be called.

If the transaction fails because of time out or transport error, the content will be empty.

*struct* **pj::OnTypingIndicationParam**
*#include <account.hpp>*

Parameters for onTypingIndication() account callback.

*Public Members*

string **fromUri**

Sender/From URI.

string **toUri**

To URI.

string **contactUri**

The Contact URI.

bool **isTyping**

Boolean to indicate if sender is typing.

*SipRxData* **rdata**

The whole message buffer.

*struct* **pj::OnMwiInfoParam**
*#include <account.hpp>*

Parameters for onMwiInfo() account callback.

*Public Members*

pjsip_evsub_state **state**

MWI subscription state.

*SipRxData* **rdata**

The whole message buffer.

*struct* **pj::PresNotifyParam**
*#include <account.hpp>*

Parameters for presNotify() account method.

*Public Members*

void * **srvPres**

Server presence subscription instance.

pjsip_evsub_state **state**

Server presence subscription state to set.

string **stateStr**

Optionally specify the state string name, if state is not "active", "pending", or "terminated".

string **reason**

If the new state is PJSIP_EVSUB_STATE_TERMINATED, optionally specify the termination reason.

bool **withBody**

If the new state is PJSIP_EVSUB_STATE_TERMINATED, this specifies whether the NOTIFY request should contain message body containing account's presence information.

*SipTxOption* **txOption**

Optional list of headers to be sent with the NOTIFY request.

### 5.6.5 Other

*class* **pj::FindBuddyMatch**

Wrapper class for *Buddy* matching algo.

Default algo is a simple substring lookup of search-token in the *Buddy* URIs, with case sensitive. Application can implement its own matching algo by overriding this class and specifying its instance in *Account::findBuddy()*.

*Public Functions*

bool **match**(const string & token, const *Buddy* & buddy)

Default algo implementation.

**~FindBuddyMatch**()

Destructor.

CHAPTER

SIX

# MEDIA

Media objects are objects that are capable to either produce media or takes media.

An important subclass of Media is AudioMedia which represents audio media. There are several type of audio media objects supported in PJSUA2:

- Capture device's AudioMedia, to capture audio from the sound device.
- Playback device's AudioMedia, to play audio to the sound device.
- Call's AudioMedia, to transmit and receive audio to/from remote person.
- AudioMediaPlayer, to play WAV file(s).
- AudioMediaRecorder, to record audio to a WAV file.

More media objects may be added in the future.

## 6.1 The Audio Conference Bridge

The conference bridge provides a simple but yet powerful concept to manage audio flow between the audio medias. The principle is very simple, that is you connect audio source to audio destination, and the bridge will make the audio flows from the source to destination, and that's it. If more than one sources are transmitting to the same destination, then the audio from the sources will be mixed. If one source is transmitting to more than one destinations, the bridge will take care of duplicating the audio from the source to the multiple destinations. The bridge will even take care medias with different clock rates and ptime.

In PJSUA2, all audio media objects are plugged-in to the central conference bridge for easier manipulation. At first, a plugged-in audio media will not be connected to anything, so media will not flow from/to any objects. An audio media source can start/stop the transmission to a destination by using the API AudioMedia.startTransmit() / AudioMedia.stopTransmit().

An audio media object plugged-in to the conference bridge will be given a port ID number that identifies the object in the bridge. Application can use the API AudioMedia.getPortId() to retrieve the port ID. Normally, application should not need to worry about the conference bridge and its port ID (as all will be taken care of by the Media class) unless application want to create its own custom audio media.

### 6.1.1 Playing a WAV File

To playback the WAV file to the sound device, just start the transmission of the WAV playback object to the sound device's playback media:

```
AudioMediaPlayer player;
AudioMedia& play_med = Endpoint::instance().audDevManager().getPlaybackDevMedia();
try {
    player.createPlayer("file.wav");
    player.startTransmit(play_med);
} catch(Error& err) {
}
```

By default, the WAV file will be played in a loop. To disable the loop, specify `PJMEDIA_FILE_NO_LOOP` when creating the player:

```
player.createPlayer("file.wav", PJMEDIA_FILE_NO_LOOP);
```

Without looping, silence will be played once the playback has reached the end of the WAV file.

Once you're done with the playback, just stop the transmission to stop the playback:

```
try {
    player.stopTransmit(play_med);
} catch(Error& err) {
}
```

Resuming the transmission after the playback is stopped will resume playback from the last play position. Use `player.setPos()` to set playback position to a desired location.

### 6.1.2 Recording to WAV File

Or if you want to record the audio from the sound device to the WAV file, simply do this:

```
AudioMediaRecorder recorder;
AudioMedia& cap_med = Endpoint::instance().audDevManager().getCaptureDevMedia();
try {
    recorder.createRecorder("file.wav");
    cap_med.startTransmit(recorder);
} catch(Error& err) {
}
```

And the media will flow from the sound device to the WAV record file. As usual, to stop or pause recording, just stop the transmission:

```
try {
   cap_med.stopTransmit(recorder);
} catch(Error& err) {
}
```

Note that stopping the transmission to the WAV recorder as above does not close the WAV file, and you can resume recording by connecting a source to the WAV recorder again. You cannot playback the recorded WAV file before you close it. To close the WAV recorder, simply delete it:

```
delete recorder;
```

### 6.1.3 Local Audio Loopback

A useful test to check whether the local sound device (capture and playback device) is working properly is by transmitting the audio from the capture device directly to the playback device (i.e. local loopback). You can do this by:

```
cap_med.startTransmit(play_med);
```

### 6.1.4 Looping Audio

If you want, you can loop the audio of an audio media object to itself (i.e. the audio received from the object will be transmitted to itself). You can loop-back audio from any objects, as long as the object has bidirectional media. That means you can loop the call's audio media, so that audio received from the remote person will be transmitted back to her/him. But you can't loop the WAV player or recorder since these objects can only play or record and not both.

### 6.1.5 Normal Call

A single call can have more than one media (for example, audio and video). Application can retrieve the audio media by using the API Call.getMedia(). Then for a normal call, we would want to establish bidirectional audio with the remote person, which can be done easily by connecting the sound device and the call audio media and vice versa:

```
CallInfo ci = call.getInfo();
AudioMedia *aud_med = NULL;

// Find out which media index is the audio
for (unsigned i=0; i<ci.media.size(); ++i) {
    if (ci.media[i].type == PJMEDIA_TYPE_AUDIO) {
        aud_med = (AudioMedia *)call.getMedia(i);
        break;
    }
}

if (aud_med) {
    // This will connect the sound device/mic to the call audio media
    cap_med.startTransmit(*aud_med);

    // And this will connect the call audio media to the sound device/speaker
    aud_med->startTransmit(play_med);
}
```

### 6.1.6 Second Call

Suppose we want to talk with two remote parties at the same time. Since we already have bidirectional media connection with one party, we just need to add bidirectional connection with the other party using the code below:

```
AudioMedia *aud_med2 = (AudioMedia *)call2.getMedia(aud_idx);
if (aud_med2) {
    cap_med->startTransmit(*aud_med2);
    aud_med2->startTransmit(play_med);
}
```

Now we can talk to both parties at the same time, and we will hear audio from either party. But at this stage, the remote parties can't talk or hear each other (i.e. we're not in full conference mode yet).

### 6.1.7 Conference Call

To enable both parties talk to each other, just establish bidirectional media between them:

```
aud_med->startTransmit(*aud_med2);
aud_med2->startTransmit(*aud_med);
```

Now the three parties (us and both remote parties) will be able to talk to each other.

### 6.1.8 Recording the Conference

While doing the conference, it perfectly makes sense to want to record the conference to a WAV file, and all we need to do is to connect the microphone and both calls to the WAV recorder:

```
cap_med.startTransmit(recorder);
aud_med->startTransmit(recorder);
aud_med2->startTransmit(recorder);
```

## 6.2 Audio Device Management

Please see Audio Device Framework below.

## 6.3 Class Reference

### 6.3.1 Media Framework

#### Classes

*class* **pj::Media**

*Media*.

*Public Functions*

**~Media**()

Virtual destructor.

pjmedia_type **getType**()

Get type of the media.

**Return**

The media type.

*class* **pj::AudioMedia**

Audio *Media*.

*Public Functions*

*ConfPortInfo* **getPortInfo**()

Get information about the specified conference port.

int **getPortId**()

Get port Id.

void **startTransmit**(const *AudioMedia* & sink)

Establish unidirectional media flow to sink.

This media port will act as a source, and it may transmit to multiple destinations/sink. And if multiple sources are transmitting to the same sink, the media will be mixed together. Source and sink may refer to the same *Media*, effectively looping the media.

If bidirectional media flow is desired, application needs to call this method twice, with the second one called from the opposite source media.

**Parameters**

- `sink` - The destination *Media*.

void **stopTransmit**(const *AudioMedia* & sink)

Stop media flow to destination/sink port.

**Parameters**

- `sink` - The destination media.

void **adjustRxLevel**(float level)

Adjust the signal level to be transmitted from the bridge to this media port by making it louder or quieter.

**Parameters**

- `level` - Signal level adjustment. Value 1.0 means no level adjustment, while value 0 means to mute the port.

void **adjustTxLevel**(float level)

Adjust the signal level to be received from this media port (to the bridge) by making it louder or quieter.

**Parameters**

- `level` - Signal level adjustment. Value 1.0 means no level adjustment, while value 0 means to mute the port.

unsigned **getRxLevel**()

Get the last received signal level.

**Return**

Signal level in percent.

unsigned **getTxLevel**()

Get the last transmitted signal level.

**Return**

Signal level in percent.

**~AudioMedia**()

Virtual Destructor.

*Public Static Functions*

*ConfPortInfo* **getPortInfoFromId**(int port_id)

Get information from specific port id.

*AudioMedia* * **typecastFromMedia**(*Media* * media)

Typecast from base class *Media*.

This is useful for application written in language that does not support downcasting such as Python.

**Return**

The object as *AudioMedia* instance

**Parameters**

- `media` - The object to be downcasted

*class* **pj::AudioMediaPlayer**

Audio *Media* Player.

*Public Functions*

**AudioMediaPlayer**()

Constructor.

void **createPlayer**(const string & file_name, unsigned options = 0)

Create a file player, and automatically add this player to the conference bridge.

**Parameters**

- `file_name` - The filename to be played. Currently only WAV files are supported, and the WAV file MUST be formatted as 16bit PCM mono/single channel (any clock rate is supported).
- `options` - Optional option flag. Application may specify PJMEDIA_FILE_NO_LOOP to prevent playback loop.

void **createPlaylist**(const *StringVector* & file_names, const string & label = "", unsigned options = 0)

Create a file playlist media port, and automatically add the port to the conference bridge.

**Parameters**

- `file_names` - Array of file names to be added to the play list. Note that the files must have the same clock rate, number of channels, and number of bits per sample.
- `label` - Optional label to be set for the media port.
- `options` - Optional option flag. Application may specify PJMEDIA_FILE_NO_LOOP to prevent looping.

void **setPos**(pj_uint32_t samples)

Set playback position.

This operation is not valid for playlist.

**Parameters**

- `samples` - The desired playback position, in samples.

**~AudioMediaPlayer**()

Virtual destructor.

*Public Static Functions*

*AudioMediaPlayer* * **typecastFromAudioMedia**(*AudioMedia* * media)

Typecast from base class *AudioMedia*.

This is useful for application written in language that does not support downcasting such as Python.

**Return**

The object as *AudioMediaPlayer* instance

**Parameters**

- `media` - The object to be downcasted

*class* **pj::AudioMediaRecorder**

Audio *Media* Recorder.

*Public Functions*

**AudioMediaRecorder**()

Constructor.

void **createRecorder**(const string & file_name, unsigned enc_type = 0, pj_ssize_t max_size = 0, unsigned options = 0)

Create a file recorder, and automatically connect this recorder to the conference bridge.

The recorder currently supports recording WAV file. The type of the recorder to use is determined by the extension of the file (e.g. ".wav").

**Parameters**

- `file_name` - Output file name. The function will determine the default format to be used based on the file extension. Currently ".wav" is supported on all platforms.
- `enc_type` - Optionally specify the type of encoder to be used to compress the media, if the file can support different encodings. This value must be zero for now.
- `max_size` - Maximum file size. Specify zero or -1 to remove size limitation. This value must be zero or -1 for now.
- `options` - Optional options, which can be used to specify the recording file format. Supported options are PJMEDIA_FILE_WRITE_PCM, PJMEDIA_FILE_WRITE_ALAW, and PJMEDIA_FILE_WRITE_ULAW. Default is zero or PJMEDIA_FILE_WRITE_PCM.

**~AudioMediaRecorder**()

Virtual destructor.

*Public Static Functions*

*AudioMediaRecorder* * **typecastFromAudioMedia**(*AudioMedia* * media)

Typecast from base class *AudioMedia*.

This is useful for application written in language that does not support downcasting such as Python.

**Return**

The object as *AudioMediaRecorder* instance

**Parameters**

- `media` - The object to be downcasted

### Formats and Info

*struct* **pj::MediaFormat**
*#include <media.hpp>*

This structure contains all the information needed to completely describe a media.

*Public Members*

pj_uint32_t **id**

The format id that specifies the audio sample or video pixel format.

Some well known formats ids are declared in pjmedia_format_id enumeration.

**See**

pjmedia_format_id

pjmedia_type **type**

The top-most type of the media, as an information.

*struct* **pj::MediaFormatAudio**
*#include <media.hpp>*

This structure describe detail information about an audio media.

*Public Functions*

void **fromPj**(const pjmedia_format & format)

Construct from pjmedia_format.

pjmedia_format **toPj**()

Export to pjmedia_format.

*Public Members*

unsigned **clockRate**

Audio clock rate in samples or Hz.

unsigned **channelCount**

Number of channels.

unsigned **frameTimeUsec**

Frame interval, in microseconds.

unsigned **bitsPerSample**

Number of bits per sample.

pj_uint32_t **avgBps**

Average bitrate.

pj_uint32_t **maxBps**

Maximum bitrate.

*struct* **pj::MediaFormatVideo**
*#include <media.hpp>*

This structure describe detail information about an video media.

*Public Members*

unsigned **width**

Video width.

unsigned **height**

Video height.

int **fpsNum**

Frames per second numerator.

int **fpsDenum**

Frames per second denumerator.

pj_uint32_t **avgBps**

Average bitrate.

pj_uint32_t **maxBps**

Maximum bitrate.

*struct* **pj::ConfPortInfo**
*#include <media.hpp>*

This structure descibes information about a particular media port that has been registered into the conference bridge.

*Public Functions*

void **fromPj**(const pjsua_conf_port_info & port_info)

Construct from pjsua_conf_port_info.

*Public Members*

int **portId**

Conference port number.

string **name**

Port name.

*MediaFormatAudio* **format**

*Media* audio format information.

float **txLevelAdj**

Tx level adjustment.

Value 1.0 means no adjustment, value 0 means the port is muted, value 2.0 means the level is amplified two times.

float **rxLevelAdj**

Rx level adjustment.

Value 1.0 means no adjustment, value 0 means the port is muted, value 2.0 means the level is amplified two times.

*IntVector* **listeners**

Array of listeners (in other words, ports where this port is transmitting to.

### 6.3.2 Audio Device Framework

#### Device Manager

*class* **pj::AudDevManager**

Audio device manager.

*Public Functions*

int **getCaptureDev**()

Get currently active capture sound devices.

If sound devices has not been created, it is possible that the function returns -1 as device IDs.

**Return**

Device ID of the capture device.

*AudioMedia* & **getCaptureDevMedia**()

Get the *AudioMedia* of the capture audio device.

**Return**

Audio media for the capture device.

int **getPlaybackDev**()

Get currently active playback sound devices.

If sound devices has not been created, it is possible that the function returns -1 as device IDs.

**Return**

Device ID of the playback device.

*AudioMedia* & **getPlaybackDevMedia**()

Get the *AudioMedia* of the speaker/playback audio device.

**Return**

Audio media for the speaker/playback device.

void **setCaptureDev**(int capture_dev)

Select or change capture sound device.

Application may call this function at any time to replace current sound device.

**Parameters**

- `capture_dev` - Device ID of the capture device.

void **setPlaybackDev**(int playback_dev)

Select or change playback sound device.

Application may call this function at any time to replace current sound device.

**Parameters**

- `playback_dev` - Device ID of the playback device.

const *AudioDevInfoVector* & **enumDev**()

Enum all audio devices installed in the system.

**Return**

The list of audio device info.

void **setNullDev**()

Set pjsua to use null sound device.

The null sound device only provides the timing needed by the conference bridge, and will not interract with any hardware.

*MediaPort* * **setNoDev**()

Disconnect the main conference bridge from any sound devices, and let application connect the bridge to it's own sound device/master port.

**Return**

The port interface of the conference bridge, so that application can connect this to it's own sound device or master port.

void **setEcOptions**(unsigned tail_msec, unsigned options)

Change the echo cancellation settings.

The behavior of this function depends on whether the sound device is currently active, and if it is, whether device or software AEC is being used.

If the sound device is currently active, and if the device supports AEC, this function will forward the change request to the device and it will be up to the device on whether support the request. If software AEC is being used (the software EC will be used if the device does not support AEC), this function will change the software EC settings. In all cases, the setting will be saved for future opening of the sound device.

If the sound device is not currently active, this will only change the default AEC settings and the setting will be applied next time the sound device is opened.

**Parameters**

- `tail_msec` - The tail length, in miliseconds. Set to zero to disable AEC.
- `options` - Options to be passed to pjmedia_echo_create(). Normally the value should be zero.

unsigned **getEcTail**()

Get current echo canceller tail length.

**Return**

The EC tail length in milliseconds, If AEC is disabled, the value will be zero.

bool **sndIsActive**()

Check whether the sound device is currently active.

The sound device may be inactive if the application has set the auto close feature to non-zero (the sndAutoCloseTime setting in *MediaConfig*), or if null sound device or no sound device has been configured via the *setNoDev()* function.

void **refreshDevs**()

Refresh the list of sound devices installed in the system.

This method will only refresh the list of audio device so all active audio streams will be unaffected. After refreshing the device list, application MUST make sure to update all index references to audio devices before calling any method that accepts audio device index as its parameter.

unsigned **getDevCount**()

Get the number of sound devices installed in the system.

**Return**

The number of sound devices installed in the system.

*AudioDevInfo* **getDevInfo**(int id)

Get device information.

**Return**

The device information which will be filled in by this method once it returns successfully.

**Parameters**

- `id` - The audio device ID.

int **lookupDev**(const string & drv_name, const string & dev_name)

Lookup device index based on the driver and device name.

**Return**

The device ID. If the device is not found, *Error* will be thrown.

**Parameters**

- `drv_name` - The driver name.
- `dev_name` - The device name.

string **capName**(pjmedia_aud_dev_cap cap)

Get string info for the specified capability.

**Return**

Capability name.

**Parameters**

- `cap` - The capability ID.

void **setExtFormat**(const *MediaFormatAudio* & format, bool keep = true)

This will configure audio format capability (other than PCM) to the sound device being used.

If sound device is currently active, the method will forward the setting to the sound device instance to be applied immediately, if it supports it.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_EXT_FORMAT capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

Note that in case the setting is kept for future use, it will be applied to any devices, even when application has changed the sound device to be used.

**Parameters**

- `format` - The audio format.

- `keep` - Specify whether the setting is to be kept for future use.

*MediaFormatAudio* **getExtFormat**()

Get the audio format capability (other than PCM) of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_EXT_FORMAT capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio format.

void **setInputLatency**(unsigned latency_msec, bool keep = true)

This will configure audio input latency control or query capability to the sound device being used.

If sound device is currently active, the method will forward the setting to the sound device instance to be applied immediately, if it supports it.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_INPUT_LATENCY capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

Note that in case the setting is kept for future use, it will be applied to any devices, even when application has changed the sound device to be used.

**Parameters**

- `latency_msec` - The input latency.
- `keep` - Specify whether the setting is to be kept for future use.

unsigned **getInputLatency**()

Get the audio input latency control or query capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_INPUT_LATENCY capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio input latency.

void **setOutputLatency**(unsigned latency_msec, bool keep = true)

This will configure audio output latency control or query capability to the sound device being used.

If sound device is currently active, the method will forward the setting to the sound device instance to be applied immediately, if it supports it.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_OUTPUT_LATENCY capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

Note that in case the setting is kept for future use, it will be applied to any devices, even when application has changed the sound device to be used.

**Parameters**

- `latency_msec` - The output latency.
- `keep` - Specify whether the setting is to be kept for future use.

unsigned **getOutputLatency**()

Get the audio output latency control or query capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_OUTPUT_LATENCY capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio output latency.

void **setInputVolume**(unsigned volume, bool keep = true)

This will configure audio input volume level capability to the sound device being used.

If sound device is currently active, the method will forward the setting to the sound device instance to be applied immediately, if it supports it.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_INPUT_VOLUME_SETTING capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

Note that in case the setting is kept for future use, it will be applied to any devices, even when application has changed the sound device to be used.

**Parameters**

- `volume` - The input volume level, in percent.

- `keep` - Specify whether the setting is to be kept for future use.

unsigned **getInputVolume**()

Get the audio input volume level capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_INPUT_VOLUME_SETTING capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown. *

**Return**

The audio input volume level, in percent.

void **setOutputVolume**(unsigned volume, bool keep = true)

This will configure audio output volume level capability to the sound device being used.

If sound device is currently active, the method will forward the setting to the sound device instance to be applied immediately, if it supports it.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_OUTPUT_VOLUME_SETTING capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

Note that in case the setting is kept for future use, it will be applied to any devices, even when application has changed the sound device to be used.

**Parameters**

- `volume` - The output volume level, in percent.
- `keep` - Specify whether the setting is to be kept for future use.

unsigned **getOutputVolume**()

Get the audio output volume level capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_OUTPUT_VOLUME_SETTING capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio output volume level, in percent.

unsigned **getInputSignal**()

Get the audio input signal level capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_INPUT_SIGNAL_METER capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio input signal level, in percent.

unsigned **getOutputSignal**()

Get the audio output signal level capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_OUTPUT_SIGNAL_METER capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio output signal level, in percent.

void **setInputRoute**(pjmedia_aud_dev_route route, bool keep = true)

This will configure audio input route capability to the sound device being used.

If sound device is currently active, the method will forward the setting to the sound device instance to be applied immediately, if it supports it.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_INPUT_ROUTE capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

Note that in case the setting is kept for future use, it will be applied to any devices, even when application has changed the sound device to be used.

**Parameters**

- `route` - The audio input route.
- `keep` - Specify whether the setting is to be kept for future use.

pjmedia_aud_dev_route **getInputRoute**()

Get the audio input route capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_INPUT_ROUTE capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio input route.

void **setOutputRoute**(pjmedia_aud_dev_route route, bool keep = true)

This will configure audio output route capability to the sound device being used.

If sound device is currently active, the method will forward the setting to the sound device instance to be applied immediately, if it supports it.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_OUTPUT_ROUTE capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

Note that in case the setting is kept for future use, it will be applied to any devices, even when application has changed the sound device to be used.

**Parameters**

- `route` - The audio output route.
- `keep` - Specify whether the setting is to be kept for future use.

pjmedia_aud_dev_route **getOutputRoute**()

Get the audio output route capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_OUTPUT_ROUTE capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio output route.

void **setVad**(bool enable, bool keep = true)

This will configure audio voice activity detection capability to the sound device being used.

If sound device is currently active, the method will forward the setting to the sound device instance to be applied immediately, if it supports it.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_VAD capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

Note that in case the setting is kept for future use, it will be applied to any devices, even when application has changed the sound device to be used.

**Parameters**

- `enable` - Enable/disable voice activity detection feature. Set true to enable.
- `keep` - Specify whether the setting is to be kept for future use.

bool **getVad**()

Get the audio voice activity detection capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_VAD capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio voice activity detection feature.

void **setCng**(bool enable, bool keep = true)

This will configure audio comfort noise generation capability to the sound device being used.

If sound device is currently active, the method will forward the setting to the sound device instance to be applied immediately, if it supports it.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_CNG capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

Note that in case the setting is kept for future use, it will be applied to any devices, even when application has changed the sound device to be used.

**Parameters**

- `enable` - Enable/disable comfort noise generation feature. Set true to enable.
- `keep` - Specify whether the setting is to be kept for future use.

bool **getCng**()

Get the audio comfort noise generation capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the

setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_CNG capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio comfort noise generation feature.

void **setPlc**(bool enable, bool keep = true)

This will configure audio packet loss concealment capability to the sound device being used.

If sound device is currently active, the method will forward the setting to the sound device instance to be applied immediately, if it supports it.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_PLC capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

Note that in case the setting is kept for future use, it will be applied to any devices, even when application has changed the sound device to be used.

**Parameters**

- `enable` - Enable/disable packet loss concealment feature. Set true to enable.
- `keep` - Specify whether the setting is to be kept for future use.

bool **getPlc**()

Get the audio packet loss concealment capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_PLC capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio packet loss concealment feature.

### Device Info

*struct* **pj::AudioDevInfo**
*#include <media.hpp>*

Audio device information structure.

*Public Functions*

void **fromPj**(const pjmedia_aud_dev_info & dev_info)

Construct from pjmedia_aud_dev_info.

**~AudioDevInfo**()

Destructor.

*Public Members*

string **name**

The device name.

unsigned **inputCount**

Maximum number of input channels supported by this device.

If the value is zero, the device does not support input operation (i.e. it is a playback only device).

unsigned **outputCount**

Maximum number of output channels supported by this device.

If the value is zero, the device does not support output operation (i.e. it is an input only device).

unsigned **defaultSamplesPerSec**

Default sampling rate.

string **driver**

The underlying driver name.

unsigned **caps**

Device capabilities, as bitmask combination of pjmedia_aud_dev_cap.

unsigned **routes**

Supported audio device routes, as bitmask combination of pjmedia_aud_dev_route.

The value may be zero if the device does not support audio routing.

*MediaFormatVector* **extFmt**

Array of supported extended audio formats.

CHAPTER

SEVEN

# CALLS

Calls are represented by Call class.

## 7.1 Subclassing the Call Class

To use the Call class, normally application SHOULD create its own subclass, such as:

```
class MyCall : public Call
{
public:
    MyCall(Account &acc, int call_id = PJSUA_INVALID_ID)
    : Call(acc, call_id)
    { }

    ~MyCall()
    { }

    // Notification when call's state has changed.
    virtual void onCallState(OnCallStateParam &prm);

    // Notification when call's media state has changed.
    virtual void onCallMediaState(OnCallMediaStateParam &prm);
};
```

In its subclass, application can implement the call callbacks, which is basically used to process events related to the call, such as call state change or incoming call transfer request.

## 7.2 Making Outgoing Calls

Making outgoing call is simple, just invoke makeCall() method of the Call object. Assuming you have the Account object as acc variable and destination URI string in dest_uri, you can initiate outgoing call with the snippet below:

```
Call *call = new MyCall(*acc);
CallOpParam prm(true); // Use default call settings
try {
    call->makeCall(dest_uri, prm);
} catch(Error& err) {
    cout << err.info() << endl;
}
```

The snippet above creates a Call object and initiates outgoing call to dest_uri using the default call settings. Subsequent operations to the call can use the method in the call instance, and events to the call will be reported to the callback. More on the callback will be explained a bit later.

## 7.3 Receiving Incoming Calls

Incoming calls are reported as onIncomingCall() of the Account class. You must derive a class from the Account class to handle incoming calls.

Below is a sample code of the callback implementation:

```
void MyAccount::onIncomingCall(OnIncomingCallParam &iprm)
{
    Call *call = new MyCall(*this, iprm.callId);
    CallOpParam prm;
    prm.statusCode = PJSIP_SC_OK;
    call->answer(prm);
}
```

For incoming calls, the call instance is created in the callback function as shown above. Application should make sure to store the call instance during the lifetime of the call (that is until the call is disconnected).

## 7.4 Call Properties

All call properties such as state, media state, remote peer information, etc. are stored as CallInfo class, which can be retrieved from the call object with using getInfo() method of the Call.

## 7.5 Call Disconnection

Call disconnection event is a special event since once the callback that reports this event returns, the call is no longer valid and any operations invoked to the call object will raise error exception. Thus, it is recommended to delete the call object inside the callback.

The call disconnection is reported in onCallState() method of Call and it can be detected as follows:

```
void MyCall::onCallState(OnCallStateParam &prm)
{
    CallInfo ci = getInfo();
    if (ci.state == PJSIP_INV_STATE_DISCONNECTED) {
        /* Delete the call */
        delete this;
    }
}
```

## 7.6 Working with Call's Audio Media

You can only operate with the call's audio media (e.g. connecting the call to the sound device in the conference bridge) when the call's audio media is ready (or active). The changes to the call's media state is reported in onCallMediaState() callback, and if the calls audio media is ready (or active) the function Call.getMedia() will return a valid audio media.

Below is a sample code to connect the call to the sound device when the media is active:

```
void MyCall::onCallMediaState(OnCallMediaStateParam &prm)
{
    CallInfo ci = getInfo();
    // Iterate all the call medias
    for (unsigned i = 0; i < ci.media.size(); i++) {
        if (ci.media[i].type==PJMEDIA_TYPE_AUDIO && getMedia(i)) {
            AudioMedia *aud_med = (AudioMedia *)getMedia(i);

            // Connect the call audio media to sound device
            AudDevManager& mgr = Endpoint::instance().audDevManager();
            aud_med->startTransmit(mgr.getPlaybackDevMedia());
            mgr.getCaptureDevMedia().startTransmit(*aud_med);
        }
    }
}
```

When the audio media becomes inactive (for example when the call is put on hold), there is no need to stop the audio media's transmission to/from the sound device since the call's audio media will be removed automatically from the conference bridge when it's no longer valid, and this will automatically remove all connections to/from the call.

## 7.7 Call Operations

You can invoke operations to the Call object, such as hanging up, putting the call on hold, sending re-INVITE, etc. Please see the reference documentation of Call for more info.

## 7.8 Instant Messaging(IM)

You can send IM within a call using Call.sendInstantMessage(). The transmission status of outgoing instant messages is reported in Call.onInstantMessageStatus() callback method.

In addition to sending instant messages, you can also send typing indication using Call.sendTypingIndication().

Incoming IM and typing indication received within a call will be reported in the callback functions Call.onInstantMessage() and Call.onTypingIndication().

Alternatively, you can send IM and typing indication outside a call by using Buddy.sendInstantMessage() and Buddy.sendTypingIndication(). For more information, please see Presence documentation.

## 7.9 Class Reference

### 7.9.1 Call

*class* **pj::Call**

*Call*.

*Public Functions*

**Call**(*Account* & acc, int call_id = PJSUA_INVALID_ID)

Constructor.

**~Call**()

Destructor.

*CallInfo* **getInfo**()

Obtain detail information about this call.

**Return**

*Call* info.

bool **isActive**()

Check if this call has active INVITE session and the INVITE session has not been disconnected.

**Return**

True if call is active.

int **getId**()

Get PJSUA-LIB call ID or index associated with this call.

**Return**

Integer greater than or equal to zero.

bool **hasMedia**()

Check if call has an active media session.

**Return**

True if yes.

*Media* * **getMedia**(unsigned med_idx)

Get media for the specified media index.

**Return**

The media or NULL if invalid or inactive.

**Parameters**

- `med_idx` - *Media* index.

pjsip_dialog_cap_status **remoteHasCap**(int htype, const string & hname, const string & token)

Check if remote peer support the specified capability.

**Return**

PJSIP_DIALOG_CAP_SUPPORTED if the specified capability is explicitly supported, see pjsip_dialog_cap_status for more info.

**Parameters**

- `htype` - The header type (pjsip_hdr_e) to be checked, which value may be:
- `hname` - If htype specifies PJSIP_H_OTHER, then the header name must be supplied in this argument. Otherwise the value must be set to empty string ("").
- `token` - The capability token to check. For example, if *htype* is PJSIP_H_ALLOW, then *token* specifies the method names; if *htype* is PJSIP_H_SUPPORTED, then *token* specifies the extension names such as "100rel".

void **setUserData**(*Token* user_data)

Attach application specific data to the call.

Application can then inspect this data by calling *getUserData()*.

**Parameters**

- `user_data` - Arbitrary data to be attached to the call.

*Token* **getUserData**()

Get user data attached to the call, which has been previously set with *setUserData()*.

**Return**

The user data.

pj_stun_nat_type **getRemNatType**()

Get the NAT type of remote's endpoint.

This is a proprietary feature of PJSUA-LIB which sends its NAT type in the SDP when *natTypeInSdp* is set in *UaConfig*.

This function can only be called after SDP has been received from remote, which means for incoming call, this function can be called as soon as call is received as long as incoming call contains SDP, and for outgoing call, this function can be called only after SDP is received (normally in 200/OK response to INVITE). As a general case, application should call this function after or in onCallMediaState() callback.

**Return**

The NAT type.

**See**

*Endpoint::natGetType()*, natTypeInSdp

void **makeCall**(const string & dst_uri, const *CallOpParam* & prm)

Make outgoing call to the specified URI.

**Parameters**

- `dst_uri` - URI to be put in the To header (normally is the same as the target URI).
- `prm.opt` - Optional call setting.
- `prm.txOption` - Optional headers etc to be added to outgoing INVITE request.

void **answer**(const *CallOpParam* & prm)

Send response to incoming INVITE request with call setting param.

Depending on the status code specified as parameter, this function may send provisional response, establish the call, or terminate the call. Notes about call setting:

**Parameters**

- `prm.opt` - Optional call setting.
- `prm.statusCode` - Status code, (100-699).
- `prm.reason` - Optional reason phrase. If empty, default text will be used.
- `prm.txOption` - Optional list of headers etc to be added to outgoing response message. Note that this message data will be persistent in all next answers/responses for this INVITE request.

void **hangup**(const *CallOpParam* & prm)

Hangup call by using method that is appropriate according to the call state.

This function is different than answering the call with 3xx-6xx response (with *answer()*), in that this function will hangup the call regardless of the state and role of the call, while *answer()* only works with incoming calls on EARLY state.

**Parameters**

- `prm.statusCode` - Optional status code to be sent when we're rejecting incoming call. If the value is zero, "603/Decline" will be sent.
- `prm.reason` - Optional reason phrase to be sent when we're rejecting incoming call. If empty, default text will be used.
- `prm.txOption` - Optional list of headers etc to be added to outgoing request/response message.

void **setHold**(const *CallOpParam* & prm)

Put the specified call on hold.

This will send re-INVITE with the appropriate SDP to inform remote that the call is being put on hold. The final status of the request itself will be reported on the onCallMediaState() callback, which inform the application that the media state of the call has changed.

**Parameters**

- `prm.options` - Bitmask of pjsua_call_flag constants. Currently, only the flag PJSUA_CALL_UPDATE_CONTACT can be used.
- `prm.txOption` - Optional message components to be sent with the request.

void **reinvite**(const *CallOpParam* & prm)

Send re-INVITE to release hold.

The final status of the request itself will be reported on the onCallMediaState() callback, which inform the application that the media state of the call has changed.

**Parameters**

- `prm.opt` - Optional call setting, if empty, the current call setting will remain unchanged.
- `prm.txOption` - Optional message components to be sent with the request.

void **update**(const *CallOpParam* & prm)

Send UPDATE request.

**Parameters**

- `prm.opt` - Optional call setting, if empty, the current call setting will remain unchanged.
- `prm.txOption` - Optional message components to be sent with the request.

void **xfer**(const string & dest, const *CallOpParam* & prm)

Initiate call transfer to the specified address.

This function will send REFER request to instruct remote call party to initiate a new INVITE session to the specified destination/target.

If application is interested to monitor the successfulness and the progress of the transfer request, it can implement onCallTransferStatus() callback which will report the progress of the call transfer request.

**Parameters**

- `dest` - URI of new target to be contacted. The URI may be in name address or addr-spec format.
- `prm.txOption` - Optional message components to be sent with the request.

void **xferReplaces**(const *Call* & dest_call, const *CallOpParam* & prm)

Initiate attended call transfer.

This function will send REFER request to instruct remote call party to initiate new INVITE session to the URL of *destCall*. The party at *dest_call* then should "replace" the call with us with the new call from the REFER recipient.

**Parameters**

- `dest_call` - The call to be replaced.
- `prm.options` - Application may specify PJSUA_XFER_NO_REQUIRE_REPLACES to suppress the inclusion of "Require: replaces" in the outgoing INVITE request created by the REFER request.
- `prm.txOption` - Optional message components to be sent with the request.

void **processRedirect**(pjsip_redirect_op cmd)

Accept or reject redirection response.

Application MUST call this function after it signaled PJSIP_REDIRECT_PENDING in the onCallRedirected() callback, to notify the call whether to accept or reject the redirection to the current target. Application can use the combination of PJSIP_REDIRECT_PENDING command in onCallRedirected() callback and this function to ask for user permission before redirecting the call.

Note that if the application chooses to reject or stop redirection (by using PJSIP_REDIRECT_REJECT or PJSIP_REDIRECT_STOP respectively), the call disconnection callback will be called before this function returns. And if the application rejects the target, the onCallRedirected() callback may also be called before this function returns if there is another target to try.

**Parameters**

- `cmd` - Redirection operation to be applied to the current target. The semantic of this argument is similar to the description in the onCallRedirected() callback, except that the PJSIP_REDIRECT_PENDING is not accepted here.

void **dialDtmf**(const string & digits)

Send DTMF digits to remote using RFC 2833 payload formats.

**Parameters**

- `digits` - DTMF string digits to be sent.

void **sendInstantMessage**(const *SendInstantMessageParam* & prm)

Send instant messaging inside INVITE session.

**Parameters**

- `prm.contentType` - MIME type.
- `prm.content` - The message content.

- `prm.txOption` - Optional list of headers etc to be included in outgoing request. The body descriptor in the txOption is ignored.
- `prm.userData` - Optional user data, which will be given back when the IM callback is called.

void **sendTypingIndication**(const *SendTypingIndicationParam* & prm)

Send IM typing indication inside INVITE session.

**Parameters**

- `prm.isTyping` - True to indicate to remote that local person is currently typing an IM.
- `prm.txOption` - Optional list of headers etc to be included in outgoing request.

void **sendRequest**(const *CallSendRequestParam* & prm)

Send arbitrary request with the call.

This is useful for example to send INFO request. Note that application should not use this function to send requests which would change the invite session's state, such as re-INVITE, UPDATE, PRACK, and BYE.

**Parameters**

- `prm.method` - SIP method of the request.
- `prm.txOption` - Optional message body and/or list of headers to be included in outgoing request.

string **dump**(bool with_media, const string indent)

Dump call and media statistics to string.

**Return**

*Call* dump and media statistics string.

**Parameters**

- `with_media` - True to include media information too.
- `indent` - Spaces for left indentation.

int **vidGetStreamIdx**()

Get the media stream index of the default video stream in the call.

Typically this will just retrieve the stream index of the first activated video stream in the call. If none is active, it will return the first inactive video stream.

**Return**

The media stream index or -1 if no video stream is present in the call.

bool **vidStreamIsRunning**(int med_idx, pjmedia_dir dir)

Determine if video stream for the specified call is currently running (i.e.

has been created, started, and not being paused) for the specified direction.

**Return**

True if stream is currently running for the specified direction.

**Parameters**

- `med_idx` - *Media* stream index, or -1 to specify default video media.
- `dir` - The direction to be checked.

void **vidSetStream**(pjsua_call_vid_strm_op op, const *CallVidSetStreamParam* & param)

Add, remove, modify, and/or manipulate video media stream for the specified call.

This may trigger a re-INVITE or UPDATE to be sent for the call.

**Parameters**

- `op` - The video stream operation to be performed, possible values are pjsua_call_vid_strm_op.
- `param` - The parameters for the video stream operation (see *CallVidSetStreamParam*).

*StreamInfo* **getStreamInfo**(unsigned med_idx)

Get media stream info for the specified media index.

**Return**

The stream info.

**Parameters**

- `med_idx` - *Media* stream index.

*StreamStat* **getStreamStat**(unsigned med_idx)

Get media stream statistic for the specified media index.

**Return**

The stream statistic.

**Parameters**

- `med_idx` - *Media* stream index.

*MediaTransportInfo* **getMedTransportInfo**(unsigned med_idx)

Get media transport info for the specified media index.

**Return**

The transport info.

**Parameters**

- `med_idx` - *Media* stream index.

void **processMediaUpdate**(*OnCallMediaStateParam* & prm)

Internal function (callled by *Endpoint*( to process update to call medias when call media state changes.

void **processStateChange**(*OnCallStateParam* & prm)

Internal function (called by *Endpoint*) to process call state change.

void **onCallState**(*OnCallStateParam* & prm)

Notify application when call state has changed.

Application may then query the call info to get the detail call states by calling *getInfo()* function.

**Parameters**

- `prm` - Callback parameter.

void **onCallTsxState**(*OnCallTsxStateParam* & prm)

This is a general notification callback which is called whenever a transaction within the call has changed state.

Application can implement this callback for example to monitor the state of outgoing requests, or to answer unhandled incoming requests (such as INFO) with a final response.

**Parameters**

- `prm` - Callback parameter.

void **onCallMediaState**(*OnCallMediaStateParam* & prm)

Notify application when media state in the call has changed.

Normal application would need to implement this callback, e.g. to connect the call's media to sound device. When ICE is used, this callback will also be called to report ICE negotiation failure.

**Parameters**

- `prm` - Callback parameter.

void **onCallSdpCreated**(*OnCallSdpCreatedParam* & prm)

Notify application when a call has just created a local SDP (for initial or subsequent SDP offer/answer).

Application can implement this callback to modify the SDP, before it is being sent and/or negotiated with remote SDP, for example to apply per account/call basis codecs priority or to add custom/proprietary SDP attributes.

**Parameters**

- `prm` - Callback parameter.

void **onStreamCreated**(*OnStreamCreatedParam* & prm)

Notify application when media session is created and before it is registered to the conference bridge.

Application may return different media port if it has added media processing port to the stream. This media port then will be added to the conference bridge instead.

**Parameters**

- `prm` - Callback parameter.

void **onStreamDestroyed**(*OnStreamDestroyedParam* & prm)

Notify application when media session has been unregistered from the conference bridge and about to be destroyed.

**Parameters**

- `prm` - Callback parameter.

void **onDtmfDigit**(*OnDtmfDigitParam* & prm)

Notify application upon incoming DTMF digits.

**Parameters**

- `prm` - Callback parameter.

void **onCallTransferRequest**(*OnCallTransferRequestParam* & prm)

Notify application on call being transferred (i.e.

REFER is received). Application can decide to accept/reject transfer request by setting the code (default is 202). When this callback is not implemented, the default behavior is to accept the transfer.

**Parameters**

- `prm` - Callback parameter.

void **onCallTransferStatus**(*OnCallTransferStatusParam* & prm)

Notify application of the status of previously sent call transfer request.

Application can monitor the status of the call transfer request, for example to decide whether to terminate existing call.

**Parameters**

- `prm` - Callback parameter.

void **onCallReplaceRequest**(*OnCallReplaceRequestParam* & prm)

Notify application about incoming INVITE with Replaces header.

Application may reject the request by setting non-2xx code.

**Parameters**

- `prm` - Callback parameter.

void **onCallReplaced**(*OnCallReplacedParam* & prm)

Notify application that an existing call has been replaced with a new call.

This happens when PJSUA-API receives incoming INVITE request with Replaces header.

After this callback is called, normally PJSUA-API will disconnect this call and establish a new call *newCallId*.

**Parameters**

- `prm` - Callback parameter.

void **onCallRxOffer**(*OnCallRxOfferParam* & prm)

Notify application when call has received new offer from remote (i.e.

re-INVITE/UPDATE with SDP is received). Application can decide to accept/reject the offer by setting the code (default is 200). If the offer is accepted, application can update the call setting to be applied in the answer. When this callback is not implemented, the default behavior is to accept the offer using current call setting.

**Parameters**

- `prm` - Callback parameter.

void **onInstantMessage**(*OnInstantMessageParam* & prm)

Notify application on incoming MESSAGE request.

**Parameters**

- `prm` - Callback parameter.

void **onInstantMessageStatus**(*OnInstantMessageStatusParam* & prm)

Notify application about the delivery status of outgoing MESSAGE request.

**Parameters**

- `prm` - Callback parameter.

void **onTypingIndication**(*OnTypingIndicationParam* & prm)

Notify application about typing indication.

**Parameters**

- `prm` - Callback parameter.

pjsip_redirect_op **onCallRedirected**(*OnCallRedirectedParam* & prm)

This callback is called when the call is about to resend the INVITE request to the specified target, following the previously received redirection response.

Application may accept the redirection to the specified target, reject this target only and make the session continue to try the next target in the list if such target exists, stop the whole redirection process altogether and cause the session to be disconnected, or defer the decision to ask for user confirmation.

This callback is optional, the default behavior is to NOT follow the redirection response.

**Return**

Action to be performed for the target. Set this parameter to one of the value below:

**Parameters**

- `prm` - Callback parameter.

void **onCallMediaTransportState**(*OnCallMediaTransportStateParam* & prm)

This callback is called when media transport state is changed.

**Parameters**

- `prm` - Callback parameter.

void **onCallMediaEvent**(*OnCallMediaEventParam* & prm)

Notification about media events such as video notifications.

This callback will most likely be called from media threads, thus application must not perform heavy processing in this callback. Especially, application must not destroy the call or media in this callback. If application needs to perform more complex tasks to handle the event, it should post the task to another thread.

**Parameters**

- `prm` - Callback parameter.

void **onCreateMediaTransport**(*OnCreateMediaTransportParam* & prm)

This callback can be used by application to implement custom media transport adapter for the call, or to replace the media transport with something completely new altogether.

This callback is called when a new call is created. The library has created a media transport for the call, and it is provided as the *mediaTp* argument of this callback. The callback may change it with the instance of media transport to be used by the call.

**Parameters**

- `prm` - Callback parameter.

*Public Static Functions*

*Call* * **lookup**(int call_id)

Get the *Call* class for the specified call Id.

**Return**

The *Call* instance or NULL if not found.

**Parameters**

- `call_id` - The call ID to lookup

### 7.9.2 Settings

*struct* **pj::CallSetting**
*#include <call.hpp>*

*Call* settings.

*Public Functions*

**CallSetting**(pj_bool_t useDefaultValues = false)

Default constructor initializes with empty or default values.

bool **isEmpty**()

Check if the settings are set with empty values.

**Return**

True if the settings are empty.

void **fromPj**(const pjsua_call_setting & prm)

Convert from pjsip.

pjsua_call_setting **toPj**()

Convert to pjsip.

*Public Members*

unsigned **flag**

Bitmask of pjsua_call_flag constants.

Default: PJSUA_CALL_INCLUDE_DISABLED_MEDIA

unsigned **reqKeyframeMethod**

This flag controls what methods to request keyframe are allowed on the call.

Value is bitmask of pjsua_vid_req_keyframe_method.

Default: PJSUA_VID_REQ_KEYFRAME_SIP_INFO | PJSUA_VID_REQ_KEYFRAME_RTCP_PLI

unsigned **audioCount**

Number of simultaneous active audio streams for this call.

Setting this to zero will disable audio in this call.

Default: 1

unsigned **videoCount**

Number of simultaneous active video streams for this call.

Setting this to zero will disable video in this call.

Default: 1 (if video feature is enabled, otherwise it is zero)

### 7.9.3 Info and Statistics

*struct* **pj::CallInfo**
*#include <call.hpp>*

*Call* information.

Application can query the call information by calling *Call::getInfo()*.

*Public Functions*

void **fromPj**(const pjsua_call_info & pci)

Convert from pjsip.

*Public Members*

pjsua_call_id **id**

*Call* identification.

pjsip_role_e **role**

Initial call role (UAC == caller)

pjsua_acc_id **accId**

The account ID where this call belongs.

string **localUri**

Local URI.

string **localContact**

Local Contact.

string **remoteUri**

Remote URI.

string **remoteContact**

Remote contact.

string **callIdString**

Dialog Call-ID string.

*CallSetting* **setting**

*Call* setting.

pjsip_inv_state **state**

*Call* state.

string **stateText**

Text describing the state.

pjsip_status_code **lastStatusCode**

Last status code heard, which can be used as cause code.

string **lastReason**

The reason phrase describing the last status.

*CallMediaInfoVector* **media**

Array of active media information.

*CallMediaInfoVector* **provMedia**

Array of provisional media information.

This contains the media info in the provisioning state, that is when the media session is being created/updated (SDP offer/answer is on progress).

*TimeVal* **connectDuration**

Up-to-date call connected duration (zero when call is not established)

*TimeVal* **totalDuration**

Total call duration, including set-up time.

bool **remOfferer**

Flag if remote was SDP offerer.

unsigned **remAudioCount**

Number of audio streams offered by remote.

unsigned **remVideoCount**

Number of video streams offered by remote.

*struct* **pj::CallMediaInfo**
*#include <call.hpp>*

*Call* media information.

*Public Functions*

**CallMediaInfo**()

Default constructor.

void **fromPj**(const pjsua_call_media_info & prm)

Convert from pjsip.

*Public Members*

unsigned **index**

*Media* index in SDP.

pjmedia_type **type**

*Media* type.

pjmedia_dir **dir**

*Media* direction.

pjsua_call_media_status **status**

*Call* media status.

int **audioConfSlot**

The conference port number for the call.

Only valid if the media type is audio.

pjsua_vid_win_id **videoIncomingWindowId**

The window id for incoming video, if any, or PJSUA_INVALID_ID.

Only valid if the media type is video.

pjmedia_vid_dev_index **videoCapDev**

The video capture device for outgoing transmission, if any, or PJMEDIA_VID_INVALID_DEV.

Only valid if the media type is video.

*struct* **pj::StreamInfo**
*#include <call.hpp>*

*Media* stream info.

*Public Functions*

void **fromPj**(const pjsua_stream_info & info)

Convert from pjsip.

*Public Members*

pjmedia_type **type**

*Media* type of this stream.

pjmedia_tp_proto **proto**

Transport protocol (RTP/AVP, etc.)

pjmedia_dir **dir**

*Media* direction.

*SocketAddress* **remoteRtpAddress**

Remote RTP address.

*SocketAddress* **remoteRtcpAddress**

Optional remote RTCP address.

unsigned **txPt**

Outgoing codec payload type.

unsigned **rxPt**

Incoming codec payload type.

string **codecName**

Codec name.

unsigned **codecClockRate**

Codec clock rate.

*CodecParam* **codecParam**

Optional codec param.

*struct* **pj::StreamStat**
*#include <call.hpp>*

*Media* stream statistic.

*Public Functions*

void **fromPj**(const pjsua_stream_stat & prm)

Convert from pjsip.

*Public Members*

*RtcpStat* **rtcp**

RTCP statistic.

*JbufState* **jbuf**

Jitter buffer statistic.

*struct* **pj::JbufState**
*#include <call.hpp>*

This structure describes jitter buffer state.

*Public Functions*

void **fromPj**(const pjmedia_jb_state & prm)

Convert from pjsip.

*Public Members*

unsigned **frameSize**

Individual frame size, in bytes.

unsigned **minPrefetch**

Minimum allowed prefetch, in frms.

unsigned **maxPrefetch**

Maximum allowed prefetch, in frms.

unsigned **burst**

Current burst level, in frames.

unsigned **prefetch**

Current prefetch value, in frames.

unsigned **size**

Current buffer size, in frames.

unsigned **avgDelayMsec**

Average delay, in ms.

unsigned **minDelayMsec**

Minimum delay, in ms.

unsigned **maxDelayMsec**

Maximum delay, in ms.

unsigned **devDelayMsec**

Standard deviation of delay, in ms.

unsigned **avgBurst**

Average burst, in frames.

unsigned **lost**

Number of lost frames.

unsigned **discard**

Number of discarded frames.

unsigned **empty**

Number of empty on GET events.

*struct* **pj::RtcpStat**
*#include <call.hpp>*

Bidirectional RTP stream statistics.

*Public Functions*

void **fromPj**(const pjmedia_rtcp_stat & prm)

Convert from pjsip.

*Public Members*

*TimeVal* **start**

Time when session was created.

*RtcpStreamStat* **txStat**

Encoder stream statistics.

*RtcpStreamStat* **rxStat**

Decoder stream statistics.

*MathStat* **rttUsec**

Round trip delay statistic.

pj_uint32_t **rtpTxLastTs**

Last TX RTP timestamp.

pj_uint16_t **rtpTxLastSeq**

Last TX RTP sequence.

*MathStat* **rxIpdvUsec**

Statistics of IP packet delay variation in receiving direction.

It is only used when PJMEDIA_RTCP_STAT_HAS_IPDV is set to non-zero.

*MathStat* **rxRawJitterUsec**

Statistic of raw jitter in receiving direction.

It is only used when PJMEDIA_RTCP_STAT_HAS_RAW_JITTER is set to non-zero.

*RtcpSdes* **peerSdes**

Peer SDES.

*struct* **pj::RtcpStreamStat**
*#include <call.hpp>*

Unidirectional RTP stream statistics.

*Public Functions*

void **fromPj**(const pjmedia_rtcp_stream_stat & prm)

Convert from pjsip.

*Public Members*

*TimeVal* **update**

Time of last update.

unsigned **updateCount**

Number of updates (to calculate avg)

unsigned **pkt**

Total number of packets.

unsigned **bytes**

Total number of payload/bytes.

unsigned **discard**

Total number of discarded packets.

unsigned **loss**

Total number of packets lost.

unsigned **reorder**

Total number of out of order packets.

unsigned **dup**

Total number of duplicates packets.

*MathStat* **lossPeriodUsec**

Loss period statistics.

unsigned **burst**

Burst/sequential packet lost detected.

unsigned **random**

Random packet lost detected.

struct pj::RtcpStreamStat::@0 **lossType**

Types of loss detected.

*MathStat* **jitterUsec**

Jitter statistics.

*struct* **pj::MathStat**
*#include <call.hpp>*

This structure describes statistics state.

*Public Functions*

**MathStat**()

Default constructor.

void **fromPj**(const pj_math_stat & prm)

Convert from pjsip.

*Public Members*

int **n**

number of samples

int **max**

maximum value

int **min**

minimum value

int **last**

last value

int **mean**

mean

*struct* **pj::MediaTransportInfo**
*#include <call.hpp>*

This structure describes media transport informations.

It corresponds to the pjmedia_transport_info structure.

*Public Functions*

void **fromPj**(const pjmedia_transport_info & info)

Convert from pjsip.

*Public Members*

*SocketAddress* **srcRtpName**

Remote address where RTP originated from.

*SocketAddress* **srcRtcpName**

Remote address where RTCP originated from.

### 7.9.4 Callback Parameters

*struct* **pj::OnCallStateParam**
*#include <call.hpp>*

This structure contains parameters for *Call::onCallState()* callback.

*Public Members*

*SipEvent* **e**

Event which causes the call state to change.

*struct* **pj::OnCallTsxStateParam**
*#include <call.hpp>*

This structure contains parameters for *Call::onCallTsxState()* callback.

*Public Members*

*SipEvent* **e**

Transaction event that caused the state change.

*struct* **pj::OnCallMediaStateParam**
*#include <call.hpp>*

This structure contains parameters for *Call::onCallMediaState()* callback.

*struct* **pj::OnCallSdpCreatedParam**
*#include <call.hpp>*

This structure contains parameters for *Call::onCallSdpCreated()* callback.

*Public Members*

*SdpSession* **sdp**

The SDP has just been created.

*SdpSession* **remSdp**

The remote SDP, will be empty if local is SDP offerer.

*struct* **pj::OnStreamCreatedParam**
*#include <call.hpp>*

This structure contains parameters for *Call::onStreamCreated()* callback.

*Public Members*

*MediaStream* **stream**

*Media* stream.

unsigned **streamIdx**

Stream index in the media session.

*MediaPort* **pPort**

On input, it specifies the media port of the stream.

Application may modify this pointer to point to different media port to be registered to the conference bridge.

*struct* **pj::OnStreamDestroyedParam**
*#include <call.hpp>*

This structure contains parameters for *Call::onStreamDestroyed()* callback.

*Public Members*

*MediaStream* **stream**

*Media* stream.

unsigned **streamIdx**

Stream index in the media session.

*struct* **pj::OnDtmfDigitParam**
*#include <call.hpp>*

This structure contains parameters for *Call::onDtmfDigit()* callback.

*Public Members*

string **digit**

DTMF ASCII digit.

*struct* **pj::OnCallTransferRequestParam**
*#include <call.hpp>*

This structure contains parameters for *Call::onCallTransferRequest()* callback.

*Public Members*

string **dstUri**

The destination where the call will be transferred to.

pjsip_status_code **statusCode**

Status code to be returned for the call transfer request.

On input, it contains status code 200.

*CallSetting* **opt**

The current call setting, application can update this setting for the call being transferred.

*struct* **pj::OnCallTransferStatusParam**
*#include <call.hpp>*

This structure contains parameters for *Call::onCallTransferStatus()* callback.

*Public Members*

pjsip_status_code **statusCode**

Status progress of the transfer request.

string **reason**

Status progress reason.

bool **finalNotify**

If true, no further notification will be reported.

The statusCode specified in this callback is the final status.

bool **cont**

Initially will be set to true, application can set this to false if it no longer wants to receive further notification (for example, after it hangs up the call).

*struct* **pj::OnCallReplaceRequestParam**
*#include <call.hpp>*

This structure contains parameters for *Call::onCallReplaceRequest()* callback.

*Public Members*

*SipRxData* **rdata**

The incoming INVITE request to replace the call.

pjsip_status_code **statusCode**

Status code to be set by application.

Application should only return a final status (200-699)

string **reason**

Optional status text to be set by application.

*CallSetting* **opt**

The current call setting, application can update this setting for the call being replaced.

*struct* **pj::OnCallReplacedParam**
*#include <call.hpp>*

This structure contains parameters for *Call::onCallReplaced()* callback.

*Public Members*

pjsua_call_id **newCallId**

The new call id.

*struct* **pj::OnCallRxOfferParam**
*#include <call.hpp>*

This structure contains parameters for *Call::onCallRxOffer()* callback.

*Public Members*

*SdpSession* **offer**

The new offer received.

pjsip_status_code **statusCode**

Status code to be returned for answering the offer.

On input, it contains status code 200. Currently, valid values are only 200 and 488.

*CallSetting* **opt**

The current call setting, application can update this setting for answering the offer.

*struct* **pj::OnCallRedirectedParam**
*#include <call.hpp>*

This structure contains parameters for *Call::onCallRedirected()* callback.

*Public Members*

string **targetUri**

The current target to be tried.

*SipEvent* **e**

The event that caused this callback to be called.

This could be the receipt of 3xx response, or 4xx/5xx response received for the INVITE sent to subsequent targets, or empty (e.type == PJSIP_EVENT_UNKNOWN) if this callback is called from within *Call::processRedirect()* context.

*struct* **pj::OnCallMediaEventParam**
*#include <call.hpp>*

This structure contains parameters for *Call::onCallMediaEvent()* callback.

*Public Members*

unsigned **medIdx**

The media stream index.

*MediaEvent* **ev**

The media event.

*struct* **pj::OnCallMediaTransportStateParam**
*#include <call.hpp>*

This structure contains parameters for *Call::onCallMediaTransportState()* callback.

*Public Members*

unsigned **medIdx**

The media index.

pjsua_med_tp_st **state**

The media transport state.

pj_status_t **status**

The last error code related to the media transport state.

int **sipErrorCode**

Optional SIP error code.

*struct* **pj::OnCreateMediaTransportParam**
*#include <call.hpp>*

This structure contains parameters for *Call::onCreateMediaTransport()* callback.

*Public Members*

unsigned **mediaIdx**

The media index in the SDP for which this media transport will be used.

*MediaTransport* **mediaTp**

The media transport which otherwise will be used by the call has this callback not been implemented.

Application can change this to its own instance of media transport to be used by the call.

unsigned **flags**

Bitmask from pjsua_create_media_transport_flag.

*struct* **pj::CallOpParam**
*#include <call.hpp>*

This structure contains parameters for *Call::answer()*, *Call::hangup()*, *Call::reinvite()*, *Call::update()*, *Call::xfer()*, *Call::xferReplaces()*, *Call::setHold()*.

*Public Functions*

**CallOpParam**(bool useDefaultCallSetting = false)

Default constructor initializes with zero/empty values.

Setting useDefaultCallSetting to true will initialize opt with default call setting values.

*Public Members*

*CallSetting* **opt**

The call setting.

pjsip_status_code **statusCode**

Status code.

string **reason**

Reason phrase.

unsigned **options**

Options.

*SipTxOption* **txOption**

List of headers etc to be added to outgoing response message.

Note that this message data will be persistent in all next answers/responses for this INVITE request.

*struct* **pj::CallSendRequestParam**
*#include <call.hpp>*

This structure contains parameters for *Call::sendRequest()*

*Public Functions*

**CallSendRequestParam**()

Default constructor initializes with zero/empty values.

*Public Members*

string **method**

SIP method of the request.

*SipTxOption* **txOption**

Message body and/or list of headers etc to be included in outgoing request.

*struct* **pj::CallVidSetStreamParam**
*#include <call.hpp>*

This structure contains parameters for *Call::vidSetStream()*

*Public Functions*

**CallVidSetStreamParam**()

Default constructor.

*Public Members*

int **medIdx**

Specify the media stream index.

This can be set to -1 to denote the default video stream in the call, which is the first active video stream or any first video stream if none is active.

This field is valid for all video stream operations, except PJSUA_CALL_VID_STRM_ADD.

Default: -1 (first active video stream, or any first video stream if none is active)

pjmedia_dir **dir**

Specify the media stream direction.

This field is valid for the following video stream operations: PJSUA_CALL_VID_STRM_ADD and PJSUA_CALL_VID_STRM_CHANGE_DIR.

Default: PJMEDIA_DIR_ENCODING_DECODING

pjmedia_vid_dev_index **capDev**

Specify the video capture device ID.

This can be set to PJMEDIA_VID_DEFAULT_CAPTURE_DEV to specify the default capture device as configured in the account.

This field is valid for the following video stream operations: PJSUA_CALL_VID_STRM_ADD and PJSUA_CALL_VID_STRM_CHANGE_CAP_DEV.

Default: PJMEDIA_VID_DEFAULT_CAPTURE_DEV.

### 7.9.5 Other

*struct* **pj::MediaEvent**
*#include* <*call.hpp*>

This structure describes a media event.

It corresponds to the pjmedia_event structure.

*Public Functions*

void **fromPj**(const pjmedia_event & ev)

Convert from pjsip.

*Public Members*

pjmedia_event_type **type**

The event type.

*MediaFmtChangedEvent* **fmtChanged**

*Media* format changed event data.

*GenericData* **ptr**

Pointer to storage to user event data, if it's outside this struct.

union pj::MediaEvent::@1 **data**

Additional data/parameters about the event.

The type of data will be specific to the event type being reported.

void * **pjMediaEvent**

Pointer to original pjmedia_event.

Only valid when the struct is converted from PJSIP's pjmedia_event.

*struct* **pj::MediaFmtChangedEvent**
*#include <call.hpp>*

This structure describes a media format changed event.

*Public Members*

unsigned **newWidth**

The new width.

unsigned **newHeight**

The new height.

*struct* **pj::SdpSession**
*#include <call.hpp>*

This structure describes SDP session description.

It corresponds to the pjmedia_sdp_session structure.

*Public Functions*

void **fromPj**(const pjmedia_sdp_session & sdp)

Convert from pjsip.

*Public Members*

string **wholeSdp**

The whole SDP as a string.

void * **pjSdpSession**

Pointer to its original pjmedia_sdp_session.

Only valid when the struct is converted from PJSIP's pjmedia_sdp_session.

*struct* **pj::RtcpSdes**
*#include <call.hpp>*

RTCP SDES structure.

*Public Functions*

void **fromPj**(const pjmedia_rtcp_sdes & prm)

Convert from pjsip.

*Public Members*

string **cname**

RTCP SDES type CNAME.

string **name**

RTCP SDES type NAME.

string **email**

RTCP SDES type EMAIL.

string **phone**

RTCP SDES type PHONE.

string **loc**

RTCP SDES type LOC.

string **tool**

RTCP SDES type TOOL.

string **note**

RTCP SDES type NOTE.

CHAPTER

EIGHT

# BUDDY (PRESENCE)

Presence feature in PJSUA2 centers around Buddy class. This class represents a remote buddy (a person, or a SIP endpoint).

## 8.1 Subclassing the Buddy class

To use the Buddy class, normally application SHOULD create its own subclass, such as:

```
class MyBuddy : public Buddy
{
public:
    MyBuddy() {}
    ~MyBuddy() {}

    virtual void onBuddyState();
};
```

In its subclass, application can implement the buddy callback to get the notifications on buddy state change.

## 8.2 Subscribing to Buddy's Presence Status

To subscribe to buddy's presence status, you need to add a buddy object and subscribe to buddy's presence status. The snippet below shows a sample code to achieve these:

```
BuddyConfig cfg;
cfg.uri = "sip:alice@example.com";
MyBuddy buddy;
try {
    buddy.create(*acc, cfg);
    buddy.subscribePresence(true);
} catch(Error& err) {
}
```

Then you can get the buddy's presence state change inside the onBuddyState() callback:

```
void MyBuddy::onBuddyState()
{
    BuddyInfo bi = getInfo();
    cout << "Buddy " << bi.uri << " is " << bi.presStatus.statusText << endl;
}
```

For more information, please see Buddy class reference documentation.

## 8.3 Responding to Presence Subscription Request

By default, incoming presence subscription to an account will be accepted automatically. You will probably want to change this behavior, for example only to automatically accept subscription if it comes from one of the buddy in the buddy list, and for anything else prompt the user if he/she wants to accept the request.

This can be done by overriding the onIncomingSubscribe() method of the Account class. Please see the documentation of this method for more info.

## 8.4 Changing Account's Presence Status

To change account's presence status, you can use the function Account.setOnlineStatus() to set basic account's presence status (i.e. available or not available) and optionally, some extended information (e.g. busy, away, on the phone, etc), such as:

```
try {
    PresenceStatus ps;
    ps.status = PJSUA_BUDDY_STATUS_ONLINE;
    // Optional, set the activity and some note
    ps.activity = PJRPID_ACTIVITY_BUSY;
    ps.note = "On the phone";
    acc->setOnlineStatus(ps);
} catch(Error& err) {
}
```

When the presence status is changed, the account will publish the new status to all of its presence subscriber, either with PUBLISH request or NOTIFY request, or both, depending on account configuration.

## 8.5 Instant Messaging(IM)

You can send IM using Buddy.sendInstantMessage(). The transmission status of outgoing instant messages is reported in Account.onInstantMessageStatus() callback method of Account class.

In addition to sending instant messages, you can also send typing indication to remote buddy using Buddy.sendTypingIndication().

Incoming IM and typing indication received not within the scope of a call will be reported in the callback functions Account.onInstantMessage() and Account.onTypingIndication().

Alternatively, you can send IM and typing indication within a call by using Call.sendInstantMessage() and Call.sendTypingIndication(). For more information, please see Call documentation.

## 8.6 Class Reference

### 8.6.1 Buddy

*class* **pj::Buddy**

*Buddy*.

*Public Functions*

**Buddy**()

Constructor.

**~Buddy**()

Destructor.

Note that if the *Buddy* instance is deleted, it will also delete the corresponding buddy in the PJSUA-LIB.

void **create**(*Account* & acc, const *BuddyConfig* & cfg)

Create buddy and register the buddy to PJSUA-LIB.

**Parameters**

- `acc` - The account for this buddy.
- `cfg` - The buddy config.

bool **isValid**()

Check if this buddy is valid.

**Return**

True if it is.

*BuddyInfo* **getInfo**()

Get detailed buddy info.

**Return**

*Buddy* info.

void **subscribePresence**(bool subscribe)

Enable/disable buddy's presence monitoring.

Once buddy's presence is subscribed, application will be informed about buddy's presence status changed via onBuddyState() callback.

**Parameters**

- `subscribe` - Specify true to activate presence subscription.

void **updatePresence**(void)

Update the presence information for the buddy.

Although the library periodically refreshes the presence subscription for all buddies, some application may want to refresh the buddy's presence subscription immediately, and in this case it can use this function to accomplish this.

Note that the buddy's presence subscription will only be initiated if presence monitoring is enabled for the buddy. See *subscribePresence()* for more info. Also if presence subscription for the buddy is already active, this function will not do anything.

Once the presence subscription is activated successfully for the buddy, application will be notified about the buddy's presence status in the onBuddyState() callback.

void **sendInstantMessage**(const *SendInstantMessageParam* & prm)

Send instant messaging outside dialog, using this buddy's specified account for route set and authentication.

**Parameters**

- `prm` - Sending instant message parameter.

void **sendTypingIndication**(const *SendTypingIndicationParam* & prm)

Send typing indication outside dialog.

**Parameters**

- `prm` - Sending instant message parameter.

void **onBuddyState**()

Notify application when the buddy state has changed.

Application may then query the buddy info to get the details.

### 8.6.2 Status

*struct* **pj::PresenceStatus**
*#include <presence.hpp>*

This describes presence status.

*Public Functions*

**PresenceStatus**()

Constructor.

*Public Members*

pjsua_buddy_status **status**

*Buddy*'s online status.

string **statusText**

Text to describe buddy's online status.

pjrpid_activity **activity**

Activity type.

string **note**

Optional text describing the person/element.

string **rpidId**

Optional RPID ID string.

### 8.6.3 Info

*struct* **pj::BuddyInfo**
*#include <presence.hpp>*

This structure describes buddy info, which can be retrieved by via *Buddy::getInfo()*.

*Public Functions*

void **fromPj**(const pjsua_buddy_info & pbi)

Import from pjsip structure.

*Public Members*

string **uri**

The full URI of the buddy, as specified in the configuration.

string **contact**

*Buddy*'s Contact, only available when presence subscription has been established to the buddy.

bool **presMonitorEnabled**

Flag to indicate that we should monitor the presence information for this buddy (normally yes, unless explicitly disabled).

pjsip_evsub_state **subState**

If *presMonitorEnabled* is true, this specifies the last state of the presence subscription.

If presence subscription session is currently active, the value will be PJSIP_EVSUB_STATE_ACTIVE. If presence subscription request has been rejected, the value will be PJSIP_EVSUB_STATE_TERMINATED, and the termination reason will be specified in *subTermReason*.

string **subStateName**

String representation of subscription state.

pjsip_status_code **subTermCode**

Specifies the last presence subscription termination code.

This would return the last status of the SUBSCRIBE request. If the subscription is terminated with NOTIFY by the server, this value will be set to 200, and subscription termination reason will be given in the *subTermReason* field.

string **subTermReason**

Specifies the last presence subscription termination reason.

If presence subscription is currently active, the value will be empty.

*PresenceStatus* **presStatus**

Presence status.

### 8.6.4 Config

*struct* **pj::BuddyConfig**
*#include <presence.hpp>*

This structure describes buddy configuration when adding a buddy to the buddy list with *Buddy::create()*.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

string **uri**

*Buddy* URL or name address.

bool **subscribe**

Specify whether presence subscription should start immediately.

CHAPTER

NINE

# PJSUA2 SAMPLE APPLICATIONS

## 9.1 Sample Apps

### 9.1.1 C++

There is a very simple C++ sample application available in `pjsip-apps/src/samples/pjsua2_demo.cpp`. The binary will be located in `pjsip-apps/bin/samples`.

### 9.1.2 Python GUI

This is a rather complete Python GUI sample apps, located in `pjsip-apps/src/pygui`. It requires Python 2.7 and above, and the Python SWIG module of course. To use the application, simply run:

```
python application.py
```

### 9.1.3 Android

Please see https://trac.pjsip.org/repos/wiki/Getting-Started/Android#pjsua2 for Android sample application.

### 9.1.4 Java

There is a Hello World type of application located in `pjsip-apps/src/swig/java`. This requires the Java SWIG module. After building the SWIG module, run `make test` from this directory to run the app.

## 9.2 Miscellaneous

### 9.2.1 How to

CHAPTER

TEN

# MEDIA QUALITY

## 10.1 Audio Quality

If you experience any problem with the audio quality, you may want to try the steps below:

1. Follow the guide: Test the sound device using pjsystest.
2. Identify the sound problem and troubleshoot it using the steps described in: Checking for sound problems.

It is probably easier to do the testing using lower level API such as PJSUA since we already have a built-in pjsua sample app located in pjsip-apps/bin to do the testing. However, you can also do the testing in your application using PJSUA2 API such as local audio loopback, recording to WAV file as explained in the Media chapter previously.

## 10.2 Video Quality

For video quality problems, the steps are as follows:

1. For lack of video, check account's AccountVideoConfig, especially the fields autoShowIncoming and autoTransmitOutgoing. More about the video API is explained in Video Users Guide.
2. Check local video preview using PJSUA API as described in Video Users Guide-Video Preview API.
3. Since video requires a larger bandwidth, we need to check for network impairments as described in Checking Network Impairments. The document is for troubleshooting audio problem but it applies for video as well.
4. Check the CPU utilization. If the CPU utilization is too high, you can try a different (less CPU-intensive) video codec or reduce the resolution/fps. A general guide on how to reduce CPU utilization can be found here: FAQ-CPU utilization.

CHAPTER

ELEVEN

# NETWORK PROBLEMS

## 11.1 IP Address Change

Please see the wiki Handling IP Address Change. Note that the guide is written using PJSUA API as a reference.

## 11.2 Blocked/Filtered Network

Please refer to the wiki Getting Around Blocked or Filtered VoIP Network.

CHAPTER

TWELVE

# PJSUA2 API REFERENCE MANUALS

## 12.1 endpoint.hpp

PJSUA2 Base Agent Operation.

*namespace* **pj**

PJSUA2 API is inside pj namespace.

*class* **OnNatDetectionCompleteParam**

Argument to *Endpoint::onNatDetectionComplete()* callback.

*Public Members*

pj_status_t **status**

Status of the detection process.

If this value is not PJ_SUCCESS, the detection has failed and *nat_type* field will contain PJ_STUN_NAT_TYPE_UNKNOWN.

string **reason**

The text describing the status, if the status is not PJ_SUCCESS.

pj_stun_nat_type **natType**

This contains the NAT type as detected by the detection procedure.

This value is only valid when the *status* is PJ_SUCCESS.

string **natTypeName**

Text describing that NAT type.

*class* **OnNatCheckStunServersCompleteParam**

Argument to *Endpoint::onNatCheckStunServersComplete()* callback.

*Public Members*

*Token* **userData**

Arbitrary user data that was passed to *Endpoint::natCheckStunServers()* function.

pj_status_t **status**

This will contain PJ_SUCCESS if at least one usable STUN server is found, otherwise it will contain the last error code during the operation.

string **name**

The server name that yields successful result.

This will only contain value if status is successful.

*SocketAddress* **addr**

The server IP address and port in "IP:port" format.

This will only contain value if status is successful.

*class* **OnTimerParam**

Parameter of *Endpoint::onTimer()* callback.

*Public Members*

*Token* **userData**

Arbitrary user data that was passed to *Endpoint::utilTimerSchedule()* function.

unsigned **msecDelay**

The interval of this timer, in miliseconds.

*class* **OnTransportStateParam**

Parameter of *Endpoint::onTransportState()* callback.

*Public Members*

*TransportHandle* **hnd**

The transport handle.

pjsip_transport_state **state**

Transport current state.

pj_status_t **lastError**

The last error code related to the transport state.

*class* **OnSelectAccountParam**

Parameter of *Endpoint::onSelectAccount()* callback.

*Public Members*

*SipRxData* **rdata**

The incoming request.

int **accountIndex**

The account index to be used to handle the request.

Upon entry, this will be filled by the account index chosen by the library. Application may change it to another value to use another account.

*class* **UaConfig**

SIP User Agent related settings.

*Public Functions*

**UaConfig**()

Default constructor to initialize with default values.

void **fromPj**(const pjsua_config & ua_cfg)

Construct from pjsua_config.

pjsua_config **toPj**()

Export to pjsua_config.

void **readObject**(const *ContainerNode* & node)

Read this object from a container.

**Parameters**

- `node` - Container to write values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container.

**Parameters**

- `node` - Container to write values to.

*Public Members*

unsigned **maxCalls**

Maximum calls to support (default: 4).

The value specified here must be smaller than the compile time maximum settings PJSUA_MAX_CALLS, which by default is 32. To increase this limit, the library must be recompiled with new PJSUA_MAX_CALLS value.

unsigned **threadCnt**

Number of worker threads.

Normally application will want to have at least one worker thread, unless when it wants to poll the library periodically, which in this case the worker thread can be set to zero.

bool **mainThreadOnly**

When this flag is non-zero, all callbacks that come from thread other than main thread will be posted to the main thread and to be executed by *Endpoint::libHandleEvents()* function.

This includes the logging callback. Note that this will only work if threadCnt is set to zero and *Endpoint::libHandleEvents()* is performed by main thread. By default, the main thread is set from the thread that invoke *Endpoint::libCreate()*

Default: false

*StringVector* **nameserver**

Array of nameservers to be used by the SIP resolver subsystem.

The order of the name server specifies the priority (first name server will be used first, unless it is not reachable).

string **userAgent**

Optional user agent string (default empty).

If it's empty, no User-Agent header will be sent with outgoing requests.

*StringVector* **stunServer**

Array of STUN servers to try.

The library will try to resolve and contact each of the STUN server entry until it finds one that is usable. Each entry may be a domain name, host name, IP address, and it may contain an optional port number. For example:

When nameserver is configured in the *pjsua_config.nameserver* field, if entry is not an IP address, it will be resolved with DNS SRV resolution first, and it will fallback to use DNS A resolution if this fails. Port number may be specified even if the entry is a domain name, in case the DNS SRV resolution should fallback to a non-standard port.

When nameserver is not configured, entries will be resolved with pj_gethostbyname() if it's not an IP address. Port number may be specified if the server is not listening in standard STUN port.

bool **stunIgnoreFailure**

This specifies if the library startup should ignore failure with the STUN servers.

If this is set to PJ_FALSE, the library will refuse to start if it fails to resolve or contact any of the STUN servers.

Default: TRUE

int **natTypeInSdp**

Support for adding and parsing NAT type in the SDP to assist troubleshooting.

The valid values are:

Default: 1

bool **mwiUnsolicitedEnabled**

Handle unsolicited NOTIFY requests containing message waiting indication (MWI) info.

Unsolicited MWI is incoming NOTIFY requests which are not requested by client with SUBSCRIBE request.

If this is enabled, the library will respond 200/OK to the NOTIFY request and forward the request to Endpoint::onMwiInfo() callback.

See also *AccountMwiConfig.enabled*.

Default: PJ_TRUE

*class* **LogEntry**

Data containing log entry to be written by the *LogWriter*.

*Public Members*

int **level**

Log verbosity level of this message.

string **msg**

The log message.

long **threadId**

ID of current thread.

string **threadName**

The name of the thread that writes this log.

*class* **LogWriter**

Interface for writing log messages.

Applications can inherit this class and supply it in the *LogConfig* structure to implement custom log writing facility.

*Public Functions*

**~LogWriter**()

Destructor.

void **write**(const *LogEntry* & entry)

Write a log entry.

*class* **LogConfig**

Logging configuration, which can be (optionally) specified when calling Lib::init().

*Public Functions*

**LogConfig**()

Default constructor initialises with default values.

void **fromPj**(const pjsua_logging_config & lc)

Construct from pjsua_logging_config.

pjsua_logging_config **toPj**()

Generate pjsua_logging_config.

void **readObject**(const *ContainerNode* & node)

Read this object from a container.

**Parameters**

- `node` - Container to write values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container.

**Parameters**

- `node` - Container to write values to.

*Public Members*

unsigned **msgLogging**

Log incoming and outgoing SIP message? Yes!

unsigned **level**

Input verbosity level.

Value 5 is reasonable.

unsigned **consoleLevel**

Verbosity level for console.

Value 4 is reasonable.

unsigned **decor**

Log decoration.

string **filename**

Optional log filename if app wishes the library to write to log file.

unsigned **fileFlags**

Additional flags to be given to pj_file_open() when opening the log file.

By default, the flag is PJ_O_WRONLY. Application may set PJ_O_APPEND here so that logs are appended to existing file instead of overwriting it.

Default is 0.

*LogWriter* * **writer**

Custom log writer, if required.

This instance will be destroyed by the endpoint when the endpoint is destroyed.

*class* **MediaConfig**

This structure describes media configuration, which will be specified when calling Lib::init().

*Public Functions*

**MediaConfig**()

Default constructor initialises with default values.

void **fromPj**(const pjsua_media_config & mc)

Construct from pjsua_media_config.

pjsua_media_config **toPj**()

Export.

void **readObject**(const *ContainerNode* & node)

Read this object from a container.

**Parameters**

- `node` - Container to write values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container.

**Parameters**

- `node` - Container to write values to.

*Public Members*

unsigned **clockRate**

Clock rate to be applied to the conference bridge.

If value is zero, default clock rate will be used (PJSUA_DEFAULT_CLOCK_RATE, which by default is 16KHz).

unsigned **sndClockRate**

Clock rate to be applied when opening the sound device.

If value is zero, conference bridge clock rate will be used.

unsigned **channelCount**

Channel count be applied when opening the sound device and conference bridge.

unsigned **audioFramePtime**

Specify audio frame ptime.

The value here will affect the samples per frame of both the sound device and the conference bridge. Specifying lower ptime will normally reduce the latency.

Default value: PJSUA_DEFAULT_AUDIO_FRAME_PTIME

unsigned **maxMediaPorts**

Specify maximum number of media ports to be created in the conference bridge.

Since all media terminate in the bridge (calls, file player, file recorder, etc), the value must be large enough to support all of them. However, the larger the value, the more computations are performed.

Default value: PJSUA_MAX_CONF_PORTS

bool **hasIoqueue**

Specify whether the media manager should manage its own ioqueue for the RTP/RTCP sockets.

If yes, ioqueue will be created and at least one worker thread will be created too. If no, the RTP/RTCP sockets will share the same ioqueue as SIP sockets, and no worker thread is needed.

Normally application would say yes here, unless it wants to run everything from a single thread.

unsigned **threadCnt**

Specify the number of worker threads to handle incoming RTP packets.

A value of one is recommended for most applications.

unsigned **quality**

*Media* quality, 0-10, according to this table: 5-10: resampling use large filter, 3-4: resampling use small filter, 1-2: resampling use linear.

The media quality also sets speex codec quality/complexity to the number.

Default: 5 (PJSUA_DEFAULT_CODEC_QUALITY).

unsigned **ptime**

Specify default codec ptime.

Default: 0 (codec specific)

bool **noVad**

Disable VAD?

Default: 0 (no (meaning VAD is enabled))

unsigned **ilbcMode**

iLBC mode (20 or 30).

Default: 30 (PJSUA_DEFAULT_ILBC_MODE)

unsigned **txDropPct**

Percentage of RTP packet to drop in TX direction (to simulate packet lost).

Default: 0

unsigned **rxDropPct**

Percentage of RTP packet to drop in RX direction (to simulate packet lost).

Default: 0

unsigned **ecOptions**

Echo canceller options (see pjmedia_echo_create())

Default: 0.

unsigned **ecTailLen**

Echo canceller tail length, in miliseconds.

Setting this to zero will disable echo cancellation.

Default: PJSUA_DEFAULT_EC_TAIL_LEN

unsigned **sndRecLatency**

Audio capture buffer length, in milliseconds.

Default: PJMEDIA_SND_DEFAULT_REC_LATENCY

unsigned **sndPlayLatency**

Audio playback buffer length, in milliseconds.

Default: PJMEDIA_SND_DEFAULT_PLAY_LATENCY

int **jbInit**

Jitter buffer initial prefetch delay in msec.

The value must be between jb_min_pre and jb_max_pre below.

Default: -1 (to use default stream settings, currently 150 msec)

int **jbMinPre**

Jitter buffer minimum prefetch delay in msec.

Default: -1 (to use default stream settings, currently 60 msec)

int **jbMaxPre**

Jitter buffer maximum prefetch delay in msec.

Default: -1 (to use default stream settings, currently 240 msec)

int **jbMax**

Set maximum delay that can be accomodated by the jitter buffer msec.

Default: -1 (to use default stream settings, currently 360 msec)

int **sndAutoCloseTime**

Specify idle time of sound device before it is automatically closed, in seconds.

Use value -1 to disable the auto-close feature of sound device

Default : 1

bool **vidPreviewEnableNative**

Specify whether built-in/native preview should be used if available.

In some systems, video input devices have built-in capability to show preview window of the device. Using this built-in preview is preferable as it consumes less CPU power. If built-in preview is not available, the library will perform software rendering of the input. If this field is set to PJ_FALSE, software preview will always be used.

Default: PJ_TRUE

*class* **EpConfig**

*Endpoint* configuration.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container.

**Parameters**

- `node` - Container to write values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container.

**Parameters**

- `node` - Container to write values to.

*Public Members*

*UaConfig* **uaConfig**

UA config.

*LogConfig* **logConfig**

Logging config.

*MediaConfig* **medConfig**

*Media* config.

*class* **PendingJob**

*Public Functions*

void **execute**(bool is_pending)

Perform the job.

**~PendingJob**()

Virtual destructor.

*class* **Endpoint**

*Endpoint* represents an instance of pjsua library.

There can only be one instance of pjsua library in an application, hence this class is a singleton.

*Public Functions*

**Endpoint**()

Default constructor.

**~Endpoint**()

Virtual destructor.

*Version* **libVersion**()

Get library version.

void **libCreate**()

Instantiate pjsua application.

Application must call this function before calling any other functions, to make sure that the underlying libraries are properly initialized. Once this function has returned success, application must call destroy() before quitting.

pjsua_state **libGetState**()

Get library state.

**Return**

library state.

void **libInit**(const *EpConfig* & prmEpConfig)

Initialize pjsua with the specified settings.

All the settings are optional, and the default values will be used when the config is not specified.

Note that create() MUST be called before calling this function.

**Parameters**

- `prmEpConfig` - *Endpoint* configurations

void **libStart**()

*Call* this function after all initialization is done, so that the library can do additional checking set up.

Application may call this function any time after init().

void **libRegisterWorkerThread**(const string & name)

Register a thread to poll for events.

This function should be called by an external worker thread, and it will block polling for events until the library is destroyed.

void **libStopWorkerThreads**()

Stop all worker threads.

int **libHandleEvents**(unsigned msec_timeout)

Poll pjsua for events, and if necessary block the caller thread for the specified maximum interval (in miliseconds).

Application doesn't normally need to call this function if it has configured worker thread (*thread_cnt* field) in pjsua_config structure, because polling then will be done by these worker threads instead.

If EpConfig::UaConfig::mainThreadOnly is enabled and this function is called from the main thread (by default the main thread is thread that calls *libCreate()*), this function will also scan and run any pending jobs in the list.

**Return**

The number of events that have been handled during the poll. Negative value indicates error, and application can retrieve the error as (status = -return_value).

**Parameters**

- `msec_timeout` - Maximum time to wait, in miliseconds.

void **libDestroy**(unsigned prmFlags = 0)

Destroy pjsua.

Application is recommended to perform graceful shutdown before calling this function (such as unregister the account from the SIP server, terminate presense subscription, and hangup active calls), however, this function will do all of these if it finds there are active sessions that need to be terminated. This function will block for few seconds to wait for replies from remote.

Application.may safely call this function more than once if it doesn't keep track of it's state.

**Parameters**

- `prmFlags` - Combination of pjsua_destroy_flag enumeration.

string **utilStrError**(pj_status_t prmErr)

Retrieve the error string for the specified status code.

**Parameters**

- `prmErr` - The error code.

void **utilLogWrite**(int prmLevel, const string & prmSender, const string & prmMsg)

Write a log message.

**Parameters**

- `prmLevel` - Log verbosity level (1-5)
- `prmSender` - The log sender.
- `prmMsg` - The log message.

void **utilLogWrite**(*LogEntry* & e)

Write a log entry.

**Parameters**

- `e` - The log entry.

pj_status_t **utilVerifySipUri**(const string & prmUri)

This is a utility function to verify that valid SIP url is given.

If the URL is a valid SIP/SIPS scheme, PJ_SUCCESS will be returned.

**Return**

PJ_SUCCESS on success, or the appropriate error code.

**See**

*utilVerifyUri()*

**Parameters**

- `prmUri` - The URL string.

pj_status_t **utilVerifyUri**(const string & prmUri)

This is a utility function to verify that valid URI is given.

Unlike *utilVerifySipUri()*, this function will return PJ_SUCCESS if tel: URI is given.

**Return**

PJ_SUCCESS on success, or the appropriate error code.

**See**

pjsua_verify_sip_url()

**Parameters**

- `prmUri` - The URL string.

*Token* **utilTimerSchedule**(unsigned prmMsecDelay, *Token* prmUserData)

Schedule a timer with the specified interval and user data.

When the interval elapsed, *onTimer()* callback will be called. Note that the callback may be executed by different thread, depending on whether worker thread is enabled or not.

**Return**

Token to identify the timer, which could be given to *utilTimerCancel()*.

**Parameters**

- `prmMsecDelay` - The time interval in msec.
- `prmUserData` - Arbitrary user data, to be given back to application in the callback.

void **utilTimerCancel**(*Token* prmToken)

Cancel previously scheduled timer with the specified timer token.

**Parameters**

- `prmToken` - The timer token, which was returned from previous *utilTimerSchedule()* call.

void **utilAddPendingJob**(*PendingJob* * job)

Utility to register a pending job to be executed by main thread.

If EpConfig::UaConfig::mainThreadOnly is false, the job will be executed immediately.

**Parameters**

- `job` - The job class.

*IntVector* **utilSslGetAvailableCiphers**()

Get cipher list supported by SSL/TLS backend.

void **natDetectType**(void)

This is a utility function to detect NAT type in front of this endpoint.

Once invoked successfully, this function will complete asynchronously and report the result in *onNatDetectionComplete()*.

After NAT has been detected and the callback is called, application can get the detected NAT type by calling *natGetType()*. Application can also perform NAT detection by calling *natDetectType()* again at later time.

Note that STUN must be enabled to run this function successfully.

pj_stun_nat_type **natGetType**()

Get the NAT type as detected by *natDetectType()* function.

This function will only return useful NAT type after *natDetectType()* has completed successfully and *onNatDetectionComplete()* callback has been called.

Exception: if this function is called while detection is in progress, PJ_EPENDING exception will be raised.

void **natCheckStunServers**(const *StringVector* & prmServers, bool prmWait, *Token* prmUserData)

Auxiliary function to resolve and contact each of the STUN server entries (sequentially) to find which is usable.

The *libInit()* must have been called before calling this function.

**See**

*natCancelCheckStunServers()*

**Parameters**

- `prmServers` - Array of STUN servers to try. The endpoint will try to resolve and contact each of the STUN server entry until it finds one that is usable. Each entry may be a domain name, host name, IP address, and it may contain an optional port number. For example:
- `prmWait` - Specify if the function should block until it gets the result. In this case, the function will block while the resolution is being done, and the callback will be called before this function returns.
- `prmUserData` - Arbitrary user data to be passed back to application in the callback.

void **natCancelCheckStunServers**(*Token* token, bool notify_cb = false)

Cancel pending STUN resolution which match the specified token.

Exception: PJ_ENOTFOUND if there is no matching one, or other error.

**Parameters**

- `token` - The token to match. This token was given to *natCheckStunServers()*
- `notify_cb` - Boolean to control whether the callback should be called for cancelled resolutions. When the callback is called, the status in the result will be set as PJ_ECANCELLED.

*TransportId* **transportCreate**(pjsip_transport_type_e type, const *TransportConfig* & cfg)

Create and start a new SIP transport according to the specified settings.

**Return**

The transport ID.

**Parameters**

- `type` - Transport type.
- `cfg` - Transport configuration.

*IntVector* **transportEnum**()

Enumerate all transports currently created in the system.

This function will return all transport IDs, and application may then call *transportGetInfo()* function to retrieve detailed information about the transport.

**Return**

Array of transport IDs.

*TransportInfo* **transportGetInfo**(*TransportId* id)

Get information about transport.

**Return**

Transport info.

**Parameters**

- `id` - Transport ID.

void **transportSetEnable**(*TransportId* id, bool enabled)

Disable a transport or re-enable it.

By default transport is always enabled after it is created. Disabling a transport does not necessarily close the socket, it will only discard incoming messages and prevent the transport from being used to send outgoing messages.

**Parameters**

- `id` - Transport ID.
- `enabled` - Enable or disable the transport.

void **transportClose**(*TransportId* id)

Close the transport.

The system will wait until all transactions are closed while preventing new users from using the transport, and will close the transport when its usage count reaches zero.

**Parameters**

- `id` - Transport ID.

void **hangupAllCalls**(void)

Terminate all calls.

This will initiate call hangup for all currently active calls.

void **mediaAdd**(*AudioMedia* & media)

Add media to the media list.

**Parameters**

- `media` - media to be added.

void **mediaRemove**(*AudioMedia* & media)

Remove media from the media list.

**Parameters**

- `media` - media to be removed.

bool **mediaExists**(const *AudioMedia* & media)

Check if media has been added to the media list.

**Return**

True if media has been added, false otherwise.

**Parameters**

- `media` - media to be check.

unsigned **mediaMaxPorts**()

Get maximum number of media port.

**Return**

Maximum number of media port in the conference bridge.

unsigned **mediaActivePorts**()

Get current number of active media port in the bridge.

**Return**

The number of active media port.

const *AudioMediaVector* & **mediaEnumPorts**()

Enumerate all media port.

**Return**

The list of media port.

*AudDevManager* & **audDevManager**()

Get the instance of Audio Device Manager.

**Return**

The Audio Device Manager.

const *CodecInfoVector* & **codecEnum**()

Enum all supported codecs in the system.

**Return**

Array of codec info.

void **codecSetPriority**(const string & codec_id, pj_uint8_t priority)

Change codec priority.

**Parameters**

- `codec_id` - Codec ID, which is a string that uniquely identify the codec (such as "speex/8000").
- `priority` - Codec priority, 0-255, where zero means to disable the codec.

*CodecParam* **codecGetParam**(const string & codec_id)

Get codec parameters.

**Return**

Codec parameters. If codec is not found, *Error* will be thrown.

**Parameters**

- `codec_id` - Codec ID.

void **codecSetParam**(const string & codec_id, const *CodecParam* param)

Set codec parameters.

**Parameters**

- `codec_id` - Codec ID.
- `param` - Codec parameter to set. Set to NULL to reset codec parameter to library default settings.

void **onNatDetectionComplete**(const *OnNatDetectionCompleteParam* & prm)

Callback when the *Endpoint* has finished performing NAT type detection that is initiated with *natDetectType()*.

**Parameters**

- `prm` - Callback parameters containing the detection result.

void **onNatCheckStunServersComplete**(const *OnNatCheckStunServersCompleteParam* & prm)

Callback when the *Endpoint* has finished performing STUN server checking that is initiated with *natCheckStunServers()*.

**Parameters**

- `prm` - Callback parameters.

void **onTransportState**(const *OnTransportStateParam* & prm)

This callback is called when transport state has changed.

**Parameters**

- `prm` - Callback parameters.

void **onTimer**(const *OnTimerParam* & prm)

Callback when a timer has fired.

The timer was scheduled by *utilTimerSchedule()*.

**Parameters**

- `prm` - Callback parameters.

void **onSelectAccount**(*OnSelectAccountParam* & prm)

This callback can be used by application to override the account to be used to handle an incoming message.

Initially, the account to be used will be calculated automatically by the library. This initial account will be used if application does not implement this callback, or application sets an invalid account upon returning from this callback.

Note that currently the incoming messages requiring account assignment are INVITE, MESSAGE, SUBSCRIBE, and unsolicited NOTIFY. This callback may be called before the callback of the SIP event itself, i.e: incoming call, pager, subscription, or unsolicited-event.

**Parameters**

- `prm` - Callback parameters.

*Public Static Functions*

*Endpoint* & **instance**()

Retrieve the singleton instance of the endpoint.

*Private Functions*

void **performPendingJobs**()

void **clearCodecInfoList**()

*Private Members*

*LogWriter* * **writer**

*AudioMediaVector* **mediaList**

*AudDevManager* **audioDevMgr**

*CodecInfoVector* **codecInfoList**

bool **mainThreadOnly**

void * **mainThread**

unsigned **pendingJobSize**

std::list< *PendingJob* * > **pendingJobs**

*Private Static Functions*

void **logFunc**(int level, const char * data, int len)

void **stun_resolve_cb**(const pj_stun_resolve_result * result)

void **on_timer**(pj_timer_heap_t * timer_heap, struct pj_timer_entry * entry)

void **on_nat_detect**(const pj_stun_nat_detect_result * res)

void **on_transport_state**(pjsip_transport * tp, pjsip_transport_state state, const pjsip_transport_state_info * info)

*Account* * **lookupAcc**(int acc_id, const char * op)

*Call* * **lookupCall**(int call_id, const char * op)

void **on_incoming_call**(pjsua_acc_id acc_id, pjsua_call_id call_id, pjsip_rx_data * rdata)

void **on_reg_started**(pjsua_acc_id acc_id, pj_bool_t renew)

void **on_reg_state2**(pjsua_acc_id acc_id, pjsua_reg_info * info)

void **on_incoming_subscribe**(pjsua_acc_id acc_id, pjsua_srv_pres * srv_pres, pjsua_buddy_id buddy_id, const pj_str_t * from, pjsip_rx_data * rdata, pjsip_status_code * code, pj_str_t * reason, pjsua_msg_data * msg_data)

void **on_pager2**(pjsua_call_id call_id, const pj_str_t * from, const pj_str_t * to, const pj_str_t * contact, const pj_str_t * mime_type, const pj_str_t * body, pjsip_rx_data * rdata, pjsua_acc_id acc_id)

void **on_pager_status2**(pjsua_call_id call_id, const pj_str_t * to, const pj_str_t * body, void * user_data, pjsip_status_code status, const pj_str_t * reason, pjsip_tx_data * tdata, pjsip_rx_data * rdata, pjsua_acc_id acc_id)

void **on_typing2**(pjsua_call_id call_id, const pj_str_t * from, const pj_str_t * to, const pj_str_t * contact, pj_bool_t is_typing, pjsip_rx_data * rdata, pjsua_acc_id acc_id)

void **on_mwi_info**(pjsua_acc_id acc_id, pjsua_mwi_info * mwi_info)

void **on_buddy_state**(pjsua_buddy_id buddy_id)

void **on_call_state**(pjsua_call_id call_id, pjsip_event * e)

void **on_call_tsx_state**(pjsua_call_id call_id, pjsip_transaction * tsx, pjsip_event * e)

void **on_call_media_state**(pjsua_call_id call_id)

void **on_call_sdp_created**(pjsua_call_id call_id, pjmedia_sdp_session * sdp, pj_pool_t * pool, const pjmedia_sdp_session * rem_sdp)

void **on_stream_created**(pjsua_call_id call_id, pjmedia_stream * strm, unsigned stream_idx, pjmedia_port ** p_port)

void **on_stream_destroyed**(pjsua_call_id call_id, pjmedia_stream * strm, unsigned stream_idx)

void **on_dtmf_digit**(pjsua_call_id call_id, int digit)

void **on_call_transfer_request**(pjsua_call_id call_id, const pj_str_t * dst, pjsip_status_code * code)

void **on_call_transfer_request2**(pjsua_call_id call_id, const pj_str_t * dst, pjsip_status_code * code, pjsua_call_setting * opt)

void **on_call_transfer_status**(pjsua_call_id call_id, int st_code, const pj_str_t * st_text, pj_bool_t final, pj_bool_t * p_cont)

void **on_call_replace_request**(pjsua_call_id call_id, pjsip_rx_data * rdata, int * st_code, pj_str_t * st_text)

void **on_call_replace_request2**(pjsua_call_id call_id, pjsip_rx_data * rdata, int * st_code, pj_str_t * st_text, pjsua_call_setting * opt)

void **on_call_replaced**(pjsua_call_id old_call_id, pjsua_call_id new_call_id)

void **on_call_rx_offer**(pjsua_call_id call_id, const pjmedia_sdp_session * offer, void * reserved, pjsip_status_code * code, pjsua_call_setting * opt)

pjsip_redirect_op **on_call_redirected**(pjsua_call_id call_id, const pjsip_uri * target, const pjsip_event * e)

pj_status_t **on_call_media_transport_state**(pjsua_call_id call_id, const pjsua_med_tp_state_info * info)

void **on_call_media_event**(pjsua_call_id call_id, unsigned med_idx, pjmedia_event * event)

pjmedia_transport * **on_create_media_transport**(pjsua_call_id call_id, unsigned media_idx, pjmedia_transport * base_tp, unsigned flags)

*Private Static Attributes*

*Endpoint* * **instance_**

## 12.2 account.hpp

PJSUA2 Account operations.

*namespace* **pj**

PJSUA2 API is inside pj namespace.

*Typedefs*

typedef std::vector< *AuthCredInfo* > **AuthCredInfoVector**

Array of SIP credentials.

*class* **AccountRegConfig**

*Account* registration config.

This will be specified in *AccountConfig*.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

string **registrarUri**

This is the URL to be put in the request URI for the registration, and will look something like "sip:serviceprovider".

This field should be specified if registration is desired. If the value is empty, no account registration will be performed.

bool **registerOnAdd**

Specify whether the account should register as soon as it is added to the UA.

Application can set this to PJ_FALSE and control the registration manually with pjsua_acc_set_registration().

Default: True

*SipHeaderVector* **headers**

The optional custom SIP headers to be put in the registration request.

unsigned **timeoutSec**

Optional interval for registration, in seconds.

If the value is zero, default interval will be used (PJSUA_REG_INTERVAL, 300 seconds).

unsigned **retryIntervalSec**

Specify interval of auto registration retry upon registration failure (including caused by transport problem), in second.

Set to 0 to disable auto re-registration. Note that if the registration retry occurs because of transport failure, the first retry will be done after *firstRetryIntervalSec* seconds instead. Also note that the interval will be randomized slightly by approximately +/- ten seconds to avoid all clients re-registering at the same time.

See also *firstRetryIntervalSec* setting.

Default: PJSUA_REG_RETRY_INTERVAL

unsigned **firstRetryIntervalSec**

This specifies the interval for the first registration retry.

The registration retry is explained in *retryIntervalSec*. Note that the value here will also be randomized by +/- ten seconds.

Default: 0

unsigned **delayBeforeRefreshSec**

Specify the number of seconds to refresh the client registration before the registration expires.

Default: PJSIP_REGISTER_CLIENT_DELAY_BEFORE_REFRESH, 5 seconds

bool **dropCallsOnFail**

Specify whether calls of the configured account should be dropped after registration failure and an attempt of re-registration has also failed.

Default: FALSE (disabled)

unsigned **unregWaitSec**

Specify the maximum time to wait for unregistration requests to complete during library shutdown sequence.

Default: PJSUA_UNREG_TIMEOUT

unsigned **proxyUse**

Specify how the registration uses the outbound and account proxy settings.

This controls if and what Route headers will appear in the REGISTER request of this account. The value is bitmask combination of PJSUA_REG_USE_OUTBOUND_PROXY and PJSUA_REG_USE_ACC_PROXY bits. If the value is set to 0, the REGISTER request will not use any proxy (i.e. it will not have any Route headers).

Default: 3 (PJSUA_REG_USE_OUTBOUND_PROXY | PJSUA_REG_USE_ACC_PROXY)

*class* **AccountSipConfig**

Various SIP settings for the account.

This will be specified in *AccountConfig*.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

*AuthCredInfoVector* **authCreds**

Array of credentials.

If registration is desired, normally there should be at least one credential specified, to successfully authenticate against the service provider. More credentials can be specified, for example when the requests are expected to be challenged by the proxies in the route set.

*StringVector* **proxies**

Array of proxy servers to visit for outgoing requests.

Each of the entry is translated into one Route URI.

string **contactForced**

Optional URI to be put as Contact for this account.

It is recommended that this field is left empty, so that the value will be calculated automatically based on the transport address.

string **contactParams**

Additional parameters that will be appended in the Contact header for this account.

This will affect the Contact header in all SIP messages sent on behalf of this account, including but not limited to REGISTER, INVITE, and SUBCRIBE requests or responses.

The parameters should be preceeded by semicolon, and all strings must be properly escaped. Example: ";my-param=X;another-param=Hi%20there"

string **contactUriParams**

Additional URI parameters that will be appended in the Contact URI for this account.

This will affect the Contact URI in all SIP messages sent on behalf of this account, including but not limited to REGISTER, INVITE, and SUBCRIBE requests or responses.

The parameters should be preceeded by semicolon, and all strings must be properly escaped. Example: ";my-param=X;another-param=Hi%20there"

bool **authInitialEmpty**

If this flag is set, the authentication client framework will send an empty Authorization header in each initial request.

Default is no.

string **authInitialAlgorithm**

Specify the algorithm to use when empty Authorization header is to be sent for each initial request (see above)

*TransportId* **transportId**

Optionally bind this account to specific transport.

This normally is not a good idea, as account should be able to send requests using any available transports according to the destination. But some application may want to have explicit control over the transport to use, so in that case it can set this field.

Default: -1 (PJSUA_INVALID_ID)

**See**

*Account::setTransport()*

*class* **AccountCallConfig**

*Account*'s call settings.

This will be specified in *AccountConfig*.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

pjsua_call_hold_type **holdType**

Specify how to offer call hold to remote peer.

Please see the documentation on pjsua_call_hold_type for more info.

Default: PJSUA_CALL_HOLD_TYPE_DEFAULT

pjsua_100rel_use **prackUse**

Specify how support for reliable provisional response (100rel/ PRACK) should be used for all sessions in this account.

See the documentation of pjsua_100rel_use enumeration for more info.

Default: PJSUA_100REL_NOT_USED

pjsua_sip_timer_use **timerUse**

Specify the usage of Session Timers for all sessions.

See the pjsua_sip_timer_use for possible values.

Default: PJSUA_SIP_TIMER_OPTIONAL

unsigned **timerMinSESec**

Specify minimum Session Timer expiration period, in seconds.

Must not be lower than 90. Default is 90.

unsigned **timerSessExpiresSec**

Specify Session Timer expiration period, in seconds.

Must not be lower than timerMinSE. Default is 1800.

*class* **AccountPresConfig**

*Account* presence config.

This will be specified in *AccountConfig*.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

*SipHeaderVector* **headers**

The optional custom SIP headers to be put in the presence subscription request.

bool **publishEnabled**

If this flag is set, the presence information of this account will be PUBLISHed to the server where the account belongs.

Default: PJ_FALSE

bool **publishQueue**

Specify whether the client publication session should queue the PUBLISH request should there be another PUBLISH transaction still pending.

If this is set to false, the client will return error on the PUBLISH request if there is another PUBLISH transaction still in progress.

Default: PJSIP_PUBLISHC_QUEUE_REQUEST (TRUE)

unsigned **publishShutdownWaitMsec**

Maximum time to wait for unpublication transaction(s) to complete during shutdown process, before sending unregistration.

The library tries to wait for the unpublication (un-PUBLISH) to complete before sending REGISTER request to unregister the account, during library shutdown process. If the value is set too short, it is possible that the unregistration is sent before unpublication completes, causing unpublication request to fail.

Value is in milliseconds.

Default: PJSUA_UNPUBLISH_MAX_WAIT_TIME_MSEC (2000)

string **pidfTupleId**

Optional PIDF tuple ID for outgoing PUBLISH and NOTIFY.

If this value is not specified, a random string will be used.

*class* **AccountMwiConfig**

*Account* MWI (Message Waiting Indication) settings.

This will be specified in *AccountConfig*.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

bool **enabled**

Subscribe to message waiting indication events (RFC 3842).

See also *UaConfig.mwiUnsolicitedEnabled* setting.

Default: FALSE

unsigned **expirationSec**

Specify the default expiration time (in seconds) for Message Waiting Indication (RFC 3842) event subscription.

This must not be zero.

Default: PJSIP_MWI_DEFAULT_EXPIRES (3600)

*class* **AccountNatConfig**

*Account*'s NAT (Network Address Translation) settings.

This will be specified in *AccountConfig*.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

pjsua_stun_use **sipStunUse**

Control the use of STUN for the SIP signaling.

Default: PJSUA_STUN_USE_DEFAULT

pjsua_stun_use **mediaStunUse**

Control the use of STUN for the media transports.

Default: PJSUA_STUN_USE_DEFAULT

bool **iceEnabled**

Enable ICE for the media transport.

Default: False

int **iceMaxHostCands**

Set the maximum number of ICE host candidates.

Default: -1 (maximum not set)

bool **iceAggressiveNomination**

Specify whether to use aggressive nomination.

Default: True

unsigned **iceNominatedCheckDelayMsec**

For controlling agent if it uses regular nomination, specify the delay to perform nominated check (connectivity check with USE-CANDIDATE attribute) after all components have a valid pair.

Default value is PJ_ICE_NOMINATED_CHECK_DELAY.

int **iceWaitNominationTimeoutMsec**

For a controlled agent, specify how long it wants to wait (in milliseconds) for the controlling agent to complete sending connectivity check with nominated flag set to true for all components after the controlled agent has found that all connectivity checks in its checklist have been completed and there is at least one successful (but not nominated) check for every component.

Default value for this option is ICE_CONTROLLED_AGENT_WAIT_NOMINATION_TIMEOUT. Specify -1 to disable this timer.

bool **iceNoRtcp**

Disable RTCP component.

Default: False

bool **iceAlwaysUpdate**

Always send re-INVITE/UPDATE after ICE negotiation regardless of whether the default ICE transport address is changed or not.

When this is set to False, re-INVITE/UPDATE will be sent only when the default ICE transport address is changed.

Default: yes

bool **turnEnabled**

Enable TURN candidate in ICE.

string **turnServer**

Specify TURN domain name or host name, in in "DOMAIN:PORT" or "HOST:PORT" format.

pj_turn_tp_type **turnConnType**

Specify the connection type to be used to the TURN server.

Valid values are PJ_TURN_TP_UDP or PJ_TURN_TP_TCP.

Default: PJ_TURN_TP_UDP

string **turnUserName**

Specify the username to authenticate with the TURN server.

int **turnPasswordType**

Specify the type of password.

Currently this must be zero to indicate plain-text password will be used in the password.

string **turnPassword**

Specify the password to authenticate with the TURN server.

int **contactRewriteUse**

This option is used to update the transport address and the Contact header of REGISTER request.

When this option is enabled, the library will keep track of the public IP address from the response of REGISTER request. Once it detects that the address has changed, it will unregister current Contact, update the Contact with transport address learned from Via header, and register a new Contact to the

registrar. This will also update the public name of UDP transport if STUN is configured.

See also contactRewriteMethod field.

Default: TRUE

int **contactRewriteMethod**

Specify how Contact update will be done with the registration, if *contactRewriteEnabled* is enabled.

The value is bitmask combination of *pjsua_contact_rewrite_method*. See also pjsua_contact_rewrite_method.

Value PJSUA_CONTACT_REWRITE_UNREGISTER(1) is the legacy behavior.

Default value: PJSUA_CONTACT_REWRITE_METHOD (PJSUA_CONTACT_REWRITE_NO_UNREG | PJSUA_CONTACT_REWRITE_ALWAYS_UPDATE)

int **viaRewriteUse**

This option is used to overwrite the "sent-by" field of the Via header for outgoing messages with the same interface address as the one in the REGISTER request, as long as the request uses the same transport instance as the previous REGISTER request.

Default: TRUE

int **sdpNatRewriteUse**

This option controls whether the IP address in SDP should be replaced with the IP address found in Via header of the REGISTER response, ONLY when STUN and ICE are not used.

If the value is FALSE (the original behavior), then the local IP address will be used. If TRUE, and when STUN and ICE are disabled, then the IP address found in registration response will be used.

Default: PJ_FALSE (no)

int **sipOutboundUse**

Control the use of SIP outbound feature.

SIP outbound is described in RFC 5626 to enable proxies or registrar to send inbound requests back to UA using the same connection initiated by the UA for its registration. This feature is highly useful in NAT-ed deployemtns, hence it is enabled by default.

Note: currently SIP outbound can only be used with TCP and TLS transports. If UDP is used for the registration, the SIP outbound feature will be silently ignored for the account.

Default: TRUE

string **sipOutboundInstanceId**

Specify SIP outbound (RFC 5626) instance ID to be used by this account.

If empty, an instance ID will be generated based on the hostname of this agent. If application specifies this parameter, the value will look

like "<urn:uuid:00000000-0000-1000-8000-AABBCCDDEEFF>" without the double-quotes.

Default: empty

string **sipOutboundRegId**

Specify SIP outbound (RFC 5626) registration ID.

The default value is empty, which would cause the library to automatically generate a suitable value.

Default: empty

unsigned **udpKaIntervalSec**

Set the interval for periodic keep-alive transmission for this account.

If this value is zero, keep-alive will be disabled for this account. The keep-alive transmission will be sent to the registrar's address, after successful registration.

Default: 15 (seconds)

string **udpKaData**

Specify the data to be transmitted as keep-alive packets.

Default: CR-LF

*class* **AccountMediaConfig**

*Account* media config (applicable for both audio and video).

This will be specified in *AccountConfig*.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

*TransportConfig* **transportConfig**

*Media* transport (RTP) configuration.

bool **lockCodecEnabled**

If remote sends SDP answer containing more than one format or codec in the media line, send re-INVITE or UPDATE with just one codec to lock which codec to use.

Default: True (Yes).

bool **streamKaEnabled**

Specify whether stream keep-alive and NAT hole punching with non-codec-VAD mechanism (see PJMEDIA_STREAM_ENABLE_KA) is enabled for this account.

Default: False

pjmedia_srtp_use **srtpUse**

Specify whether secure media transport should be used for this account.

Valid values are PJMEDIA_SRTP_DISABLED, PJMEDIA_SRTP_OPTIONAL, and PJMEDIA_SRTP_MANDATORY.

Default: PJSUA_DEFAULT_USE_SRTP

int **srtpSecureSignaling**

Specify whether SRTP requires secure signaling to be used.

This option is only used when *use_srtp* option above is non-zero.

Valid values are: 0: SRTP does not require secure signaling 1: SRTP requires secure transport such as TLS 2: SRTP requires secure end-to-end transport (SIPS)

Default: PJSUA_DEFAULT_SRTP_SECURE_SIGNALING

pjsua_ipv6_use **ipv6Use**

Specify whether IPv6 should be used on media.

Default is not used.

*class* **AccountVideoConfig**

*Account* video config.

This will be specified in *AccountConfig*.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

bool **autoShowIncoming**

Specify whether incoming video should be shown to screen by default.

This applies to incoming call (INVITE), incoming re-INVITE, and incoming UPDATE requests.

Regardless of this setting, application can detect incoming video by implementing *on_call_media_state()* callback and enumerating the media stream(s) with pjsua_call_get_info(). Once incoming video is recognised, application may retrieve the window associated with the incoming video and show or hide it with pjsua_vid_win_set_show().

Default: False

bool **autoTransmitOutgoing**

Specify whether outgoing video should be activated by default when making outgoing calls and/or when incoming video is detected.

This applies to incoming and outgoing calls, incoming re-INVITE, and incoming UPDATE. If the setting is non-zero, outgoing video transmission will be started as soon as response to these requests is sent (or received).

Regardless of the value of this setting, application can start and stop outgoing video transmission with pjsua_call_set_vid_strm().

Default: False

unsigned **windowFlags**

Specify video window's flags.

The value is a bitmask combination of pjmedia_vid_dev_wnd_flag.

Default: 0

pjmedia_vid_dev_index **defaultCaptureDevice**

Specify the default capture device to be used by this account.

If vidOutAutoTransmit is enabled, this device will be used for capturing video.

Default: PJMEDIA_VID_DEFAULT_CAPTURE_DEV

pjmedia_vid_dev_index **defaultRenderDevice**

Specify the default rendering device to be used by this account.

Default: PJMEDIA_VID_DEFAULT_RENDER_DEV

pjmedia_vid_stream_rc_method **rateControlMethod**

Rate control method.

Default: PJMEDIA_VID_STREAM_RC_SIMPLE_BLOCKING.

unsigned **rateControlBandwidth**

Upstream/outgoing bandwidth.

If this is set to zero, the video stream will use codec maximum bitrate setting.

Default: 0 (follow codec maximum bitrate).

*class* **AccountConfig**

*Account* configuration.

*Public Functions*

**AccountConfig**()

Default constructor will initialize with default values.

void **toPj**(pjsua_acc_config & cfg)

This will return a temporary pjsua_acc_config instance, which contents are only valid as long as this *AccountConfig* structure remains valid AND no modifications are done to it AND no further *toPj()* function call is made.

Any call to *toPj()* function will invalidate the content of temporary pjsua_acc_config that was returned by the previous call.

void **fromPj**(const pjsua_acc_config & prm, const pjsua_media_config * mcfg)

Initialize from pjsip.

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

int **priority**

*Account* priority, which is used to control the order of matching incoming/outgoing requests.

The higher the number means the higher the priority is, and the account will be matched first.

string **idUri**

The Address of Record or AOR, that is full SIP URL that identifies the account.

The value can take name address or URL format, and will look something like "sip:account@serviceprovider".

This field is mandatory.

*AccountRegConfig* **regConfig**

Registration settings.

*AccountSipConfig* **sipConfig**

SIP settings.

*AccountCallConfig* **callConfig**

*Call* settings.

*AccountPresConfig* **presConfig**

Presence settings.

*AccountMwiConfig* **mwiConfig**

MWI (Message Waiting Indication) settings.

*AccountNatConfig* **natConfig**

NAT settings.

*AccountMediaConfig* **mediaConfig**

*Media* settings (applicable for both audio and video).

*AccountVideoConfig* **videoConfig**

Video settings.

*class* **AccountInfo**

*Account* information.

Application can query the account information by calling *Account::getInfo()*.

*Public Functions*

void **fromPj**(const pjsua_acc_info & pai)

Import from pjsip data.

*Public Members*

pjsua_acc_id **id**

The account ID.

bool **isDefault**

Flag to indicate whether this is the default account.

string **uri**

*Account* URI.

bool **regIsConfigured**

Flag to tell whether this account has registration setting (reg_uri is not empty).

bool **regIsActive**

Flag to tell whether this account is currently registered (has active registration session).

int **regExpiresSec**

An up to date expiration interval for account registration session.

pjsip_status_code **regStatus**

Last registration status code.

If status code is zero, the account is currently not registered. Any other value indicates the SIP status code of the registration.

string **regStatusText**

String describing the registration status.

pj_status_t **regLastErr**

Last registration error code.

When the status field contains a SIP status code that indicates a registration failure, last registration error code contains the error code that causes the failure. In any other case, its value is zero.

bool **onlineStatus**

Presence online status for this account.

string **onlineStatusText**

Presence online status text.

*class* **OnIncomingCallParam**

This structure contains parameters for onIncomingCall() account callback.

*Public Members*

int **callId**

The library call ID allocated for the new call.

*SipRxData* **rdata**

The incoming INVITE request.

*class* **OnRegStartedParam**

This structure contains parameters for onRegStarted() account callback.

*Public Members*

bool **renew**

True for registration and False for unregistration.

*class* **OnRegStateParam**

This structure contains parameters for onRegState() account callback.

*Public Members*

pj_status_t **status**

Registration operation status.

pjsip_status_code **code**

SIP status code received.

string **reason**

SIP reason phrase received.

*SipRxData* **rdata**

The incoming message.

int **expiration**

Next expiration interval.

*class* **OnIncomingSubscribeParam**

This structure contains parameters for onIncomingSubscribe() callback.

*Public Members*

void * **srvPres**

Server presence subscription instance.

If application delays the acceptance of the request, it will need to specify this object when calling *Account::presNotify()*.

string **fromUri**

Sender URI.

*SipRxData* **rdata**

The incoming message.

pjsip_status_code **code**

The status code to respond to the request.

The default value is 200. Application may set this to other final status code to accept or reject the request.

string **reason**

The reason phrase to respond to the request.

*SipTxOption* **txOption**

Additional data to be sent with the response, if any.

*class* **OnInstantMessageParam**

Parameters for onInstantMessage() account callback.

*Public Members*

string **fromUri**

Sender From URI.

string **toUri**

To URI of the request.

string **contactUri**

Contact URI of the sender.

string **contentType**

MIME type of the message body.

string **msgBody**

The message body.

*SipRxData* **rdata**

The whole message.

*class* **OnInstantMessageStatusParam**

Parameters for onInstantMessageStatus() account callback.

*Public Members*

*Token* **userData**

Token or a user data that was associated with the pager transmission.

string **toUri**

Destination URI.

string **msgBody**

The message body.

pjsip_status_code **code**

The SIP status code of the transaction.

string **reason**

The reason phrase of the transaction.

*SipRxData* **rdata**

The incoming response that causes this callback to be called.

If the transaction fails because of time out or transport error, the content will be empty.

*class* **OnTypingIndicationParam**

Parameters for onTypingIndication() account callback.

*Public Members*

string **fromUri**

Sender/From URI.

string **toUri**

To URI.

string **contactUri**

The Contact URI.

bool **isTyping**

Boolean to indicate if sender is typing.

*SipRxData* **rdata**

The whole message buffer.

*class* **OnMwiInfoParam**

Parameters for onMwiInfo() account callback.

*Public Members*

pjsip_evsub_state **state**

MWI subscription state.

*SipRxData* **rdata**

The whole message buffer.

*class* **PresNotifyParam**

Parameters for presNotify() account method.

*Public Members*

void * **srvPres**

Server presence subscription instance.

pjsip_evsub_state **state**

Server presence subscription state to set.

string **stateStr**

Optionally specify the state string name, if state is not "active", "pending", or "terminated".

string **reason**

If the new state is PJSIP_EVSUB_STATE_TERMINATED, optionally specify the termination reason.

bool **withBody**

If the new state is PJSIP_EVSUB_STATE_TERMINATED, this specifies whether the NOTIFY request should contain message body containing account's presence information.

*SipTxOption* **txOption**

Optional list of headers to be sent with the NOTIFY request.

*class* **FindBuddyMatch**

Wrapper class for *Buddy* matching algo.

Default algo is a simple substring lookup of search-token in the *Buddy* URIs, with case sensitive. Application can implement its own matching algo by overriding this class and specifying its instance in *Account::findBuddy()*.

*Public Functions*

bool **match**(const string & token, const *Buddy* & buddy)

Default algo implementation.

**~FindBuddyMatch**()

Destructor.

*class* **Account**

*Account*.

*Public Functions*

**Account**()

Constructor.

**~Account**()

Destructor.

Note that if the account is deleted, it will also delete the corresponding account in the PJSUA-LIB.

void **create**(const *AccountConfig* & cfg, bool make_default = false)

Create the account.

**Parameters**

- `cfg` - The account config.
- `make_default` - Make this the default account.

void **modify**(const *AccountConfig* & cfg)

Modify the account to use the specified account configuration.

Depending on the changes, this may cause unregistration or reregistration on the account.

**Parameters**

- `cfg` - New account config to be applied to the account.

bool **isValid**()

Check if this account is still valid.

**Return**

True if it is.

void **setDefault**()

Set this as default account to be used when incoming and outgoing requests don't match any accounts.

**Return**

PJ_SUCCESS on success.

bool **isDefault**()

Check if this account is the default account.

Default account will be used for incoming and outgoing requests that don't match any other accounts.

**Return**

True if this is the default account.

int **getId**()

Get PJSUA-LIB account ID or index associated with this account.

**Return**

Integer greater than or equal to zero.

*AccountInfo* **getInfo**()

Get account info.

**Return**

*Account* info.

void **setRegistration**(bool renew)

Update registration or perform unregistration.

Application normally only needs to call this function if it wants to manually update the registration or to unregister from the server.

**Parameters**

- `renew` - If False, this will start unregistration process.

void **setOnlineStatus**(const *PresenceStatus* & pres_st)

Set or modify account's presence online status to be advertised to remote/presence subscribers.

This would trigger the sending of outgoing NOTIFY request if there are server side presence subscription for this account, and/or outgoing PUBLISH if presence publication is enabled for this account.

**Parameters**

- `pres_st` - Presence online status.

void **setTransport**(*TransportId* tp_id)

Lock/bind this account to a specific transport/listener.

Normally application shouldn't need to do this, as transports will be selected automatically by the library according to the destination.

When account is locked/bound to a specific transport, all outgoing requests from this account will use the specified transport (this includes SIP registration, dialog (call and event subscription), and out-of-dialog requests such as MESSAGE).

Note that transport id may be specified in *AccountConfig* too.

**Parameters**

- `tp_id` - The transport ID.

void **presNotify**(const *PresNotifyParam* & prm)

Send NOTIFY to inform account presence status or to terminate server side presence subscription.

If application wants to reject the incoming request, it should set the param PresNotifyParam.state to PJSIP_EVSUB_STATE_TERMINATED.

**Parameters**

- `prm` - The sending NOTIFY parameter.

const *BuddyVector* & **enumBuddies**()

Enumerate all buddies of the account.

**Return**

The buddy list.

*Buddy* * **findBuddy**(string uri, *FindBuddyMatch* * buddy_match = NULL)

Find a buddy in the buddy list with the specified URI.

Exception: if buddy is not found, PJ_ENOTFOUND will be thrown.

**Return**

The pointer to buddy.

**Parameters**

- `uri` - The buddy URI.
- `buddy_match` - The buddy match algo.

void **addBuddy**(*Buddy* * buddy)

An internal function to add a *Buddy* to *Account* buddy list.

This function must never be used by application.

void **removeBuddy**(*Buddy* * buddy)

An internal function to remove a *Buddy* from *Account* buddy list.

This function must never be used by application.

void **onIncomingCall**(*OnIncomingCallParam* & prm)

Notify application on incoming call.

**Parameters**

- `prm` - Callback parameter.

void **onRegStarted**(*OnRegStartedParam* & prm)

Notify application when registration or unregistration has been initiated.

Note that this only notifies the initial registration and unregistration. Once registration session is active, subsequent refresh will not cause this callback to be called.

**Parameters**

- `prm` - Callback parameter.

void **onRegState**(*OnRegStateParam* & prm)

Notify application when registration status has changed.

Application may then query the account info to get the registration details.

**Parameters**

- `prm` - Callback parameter.

void **onIncomingSubscribe**(*OnIncomingSubscribeParam* & prm)

Notification when incoming SUBSCRIBE request is received.

Application may use this callback to authorize the incoming subscribe request (e.g. ask user permission if the request should be granted).

If this callback is not implemented, all incoming presence subscription requests will be accepted.

If this callback is implemented, application has several choices on what to do with the incoming request:

Any IncomingSubscribeParam.code other than 200 and 202 will be treated as 200.

Application MUST return from this callback immediately (e.g. it must not block in this callback while waiting for user confirmation).

**Parameters**

- `prm` - Callback parameter.

void **onInstantMessage**(*OnInstantMessageParam* & prm)

Notify application on incoming instant message or pager (i.e.

MESSAGE request) that was received outside call context.

**Parameters**

- `prm` - Callback parameter.

void **onInstantMessageStatus**(*OnInstantMessageStatusParam* & prm)

Notify application about the delivery status of outgoing pager/instant message (i.e.

MESSAGE) request.

**Parameters**

- `prm` - Callback parameter.

void **onTypingIndication**(*OnTypingIndicationParam* & prm)

Notify application about typing indication.

**Parameters**

- `prm` - Callback parameter.

void **onMwiInfo**(*OnMwiInfoParam* & prm)

Notification about MWI (Message Waiting Indication) status change.

This callback can be called upon the status change of the SUBSCRIBE request (for example, 202/Accepted to SUBSCRIBE is received) or when a NOTIFY reqeust is received.

**Parameters**

- `prm` - Callback parameter.

*Public Static Functions*

*Account* * **lookup**(int acc_id)

Get the *Account* class for the specified account Id.

**Return**

The *Account* instance or NULL if not found.

**Parameters**

- `acc_id` - The account ID to lookup

*Private Members*

pjsua_acc_id **id**

string **tmpReason**

*BuddyVector* **buddyList**

*Friends*

friend class **Endpoint**

## 12.3 media.hpp

PJSUA2 media operations.

*namespace* **pj**

PJSUA2 API is inside pj namespace.

*Typedefs*

typedef std::vector< *MediaFormat* * > **MediaFormatVector**

Array of *MediaFormat*.

typedef void * **MediaPort**

*Media* port, corresponds to pjmedia_port.

typedef std::vector< *AudioMedia* * > **AudioMediaVector**

Array of Audio *Media*.

typedef std::vector< *AudioDevInfo* * > **AudioDevInfoVector**

Array of audio device info.

typedef std::vector< *CodecInfo* * > **CodecInfoVector**

Array of codec info.

*class* **MediaFormat**

This structure contains all the information needed to completely describe a media.

*Public Members*

pj_uint32_t **id**

The format id that specifies the audio sample or video pixel format.

Some well known formats ids are declared in pjmedia_format_id enumeration.

**See**

pjmedia_format_id

pjmedia_type **type**

The top-most type of the media, as an information.

*class* **MediaFormatAudio**

This structure describe detail information about an audio media.

*Public Functions*

void **fromPj**(const pjmedia_format & format)

Construct from pjmedia_format.

pjmedia_format **toPj**()

Export to pjmedia_format.

*Public Members*

unsigned **clockRate**

Audio clock rate in samples or Hz.

unsigned **channelCount**

Number of channels.

unsigned **frameTimeUsec**

Frame interval, in microseconds.

unsigned **bitsPerSample**

Number of bits per sample.

pj_uint32_t **avgBps**

Average bitrate.

pj_uint32_t **maxBps**

Maximum bitrate.

*class* **MediaFormatVideo**

This structure describe detail information about an video media.

*Public Members*

unsigned **width**

Video width.

unsigned **height**

Video height.

int **fpsNum**

Frames per second numerator.

int **fpsDenum**

Frames per second denumerator.

pj_uint32_t **avgBps**

Average bitrate.

pj_uint32_t **maxBps**

Maximum bitrate.

*class* **ConfPortInfo**

This structure descibes information about a particular media port that has been registered into the conference bridge.

*Public Functions*

void **fromPj**(const pjsua_conf_port_info & port_info)

Construct from pjsua_conf_port_info.

*Public Members*

int **portId**

Conference port number.

string **name**

Port name.

*MediaFormatAudio* **format**

*Media* audio format information.

float **txLevelAdj**

Tx level adjustment.

Value 1.0 means no adjustment, value 0 means the port is muted, value 2.0 means the level is amplified two times.

float **rxLevelAdj**

Rx level adjustment.

Value 1.0 means no adjustment, value 0 means the port is muted, value 2.0 means the level is amplified two times.

*IntVector* **listeners**

Array of listeners (in other words, ports where this port is transmitting to.

*class* **Media**

*Media*.

*Public Functions*

**~Media**()

Virtual destructor.

pjmedia_type **getType**()

Get type of the media.

**Return**

The media type.

*Protected Functions*

**Media**(pjmedia_type med_type)

Constructor.

*Private Members*

pjmedia_type **type**

*Media* type.

*class* **AudioMedia**

Audio *Media*.

*Public Functions*

*ConfPortInfo* **getPortInfo**()

Get information about the specified conference port.

int **getPortId**()

Get port Id.

void **startTransmit**(const *AudioMedia* & sink)

Establish unidirectional media flow to sink.

This media port will act as a source, and it may transmit to multiple destinations/sink. And if multiple sources are transmitting to the same sink, the media will be mixed together. Source and sink may refer to the same *Media*, effectively looping the media.

If bidirectional media flow is desired, application needs to call this method twice, with the second one called from the opposite source media.

**Parameters**

- `sink` - The destination *Media*.

void **stopTransmit**(const *AudioMedia* & sink)

Stop media flow to destination/sink port.

**Parameters**

- `sink` - The destination media.

void **adjustRxLevel**(float level)

Adjust the signal level to be transmitted from the bridge to this media port by making it louder or quieter.

**Parameters**

- `level` - Signal level adjustment. Value 1.0 means no level adjustment, while value 0 means to mute the port.

void **adjustTxLevel**(float level)

Adjust the signal level to be received from this media port (to the bridge) by making it louder or quieter.

**Parameters**

- `level` - Signal level adjustment. Value 1.0 means no level adjustment, while value 0 means to mute the port.

unsigned **getRxLevel**()

Get the last received signal level.

**Return**

Signal level in percent.

unsigned **getTxLevel**()

Get the last transmitted signal level.

**Return**

Signal level in percent.

**~AudioMedia**()

Virtual Destructor.

*Public Static Functions*

*ConfPortInfo* **getPortInfoFromId**(int port_id)

Get information from specific port id.

*AudioMedia* * **typecastFromMedia**(*Media* * media)

Typecast from base class *Media*.

This is useful for application written in language that does not support down-casting such as Python.

**Return**

The object as *AudioMedia* instance

**Parameters**

- `media` - The object to be downcasted

*Protected Functions*

**AudioMedia**()

Default Constructor.

void **registerMediaPort**(*MediaPort* port)

This method needs to be called by descendants of this class to register the media port created to the conference bridge and *Endpoint*'s media list.

param port the media port to be registered to the conference bridge.

void **unregisterMediaPort**()

This method needs to be called by descendants of this class to remove the media port from the conference bridge and *Endpoint*'s media list.

Descendant should only call this method if it has registered the media with the previous call to *registerMediaPort()*.

*Protected Attributes*

int **id**

Conference port Id.

*Private Functions*

unsigned **getSignalLevel**(bool is_rx = true)

*Private Members*

pj_caching_pool **mediaCachingPool**

pj_pool_t * **mediaPool**

*class* **AudioMediaPlayer**

Audio *Media* Player.

*Public Functions*

**AudioMediaPlayer**()

Constructor.

void **createPlayer**(const string & file_name, unsigned options = 0)

Create a file player, and automatically add this player to the conference bridge.

**Parameters**

- `file_name` - The filename to be played. Currently only WAV files are supported, and the WAV file MUST be formatted as 16bit PCM mono/single channel (any clock rate is supported).
- `options` - Optional option flag. Application may specify PJMEDIA_FILE_NO_LOOP to prevent playback loop.

void **createPlaylist**(const *StringVector* & file_names, const string & label = “”, unsigned options = 0)

Create a file playlist media port, and automatically add the port to the conference bridge.

**Parameters**

- `file_names` - Array of file names to be added to the play list. Note that the files must have the same clock rate, number of channels, and number of bits per sample.
- `label` - Optional label to be set for the media port.
- `options` - Optional option flag. Application may specify PJMEDIA_FILE_NO_LOOP to prevent looping.

void **setPos**(pj_uint32_t samples)

Set playback position.

This operation is not valid for playlist.

**Parameters**

- `samples` - The desired playback position, in samples.

**~AudioMediaPlayer**()

Virtual destructor.

*Public Static Functions*

*AudioMediaPlayer* * **typecastFromAudioMedia**(*AudioMedia* * media)

Typecast from base class *AudioMedia*.

This is useful for application written in language that does not support downcasting such as Python.

**Return**

The object as *AudioMediaPlayer* instance

**Parameters**

- `media` - The object to be downcasted

*Private Members*

int **playerId**

Player Id.

*class* **AudioMediaRecorder**

Audio *Media* Recorder.

*Public Functions*

**AudioMediaRecorder**()

Constructor.

void **createRecorder**(const string & file_name, unsigned enc_type = 0, pj_ssize_t max_size = 0, unsigned options = 0)

Create a file recorder, and automatically connect this recorder to the conference bridge.

The recorder currently supports recording WAV file. The type of the recorder to use is determined by the extension of the file (e.g. ".wav").

**Parameters**

- `file_name` - Output file name. The function will determine the default format to be used based on the file extension. Currently ".wav" is supported on all platforms.
- `enc_type` - Optionally specify the type of encoder to be used to compress the media, if the file can support different encodings. This value must be zero for now.
- `max_size` - Maximum file size. Specify zero or -1 to remove size limitation. This value must be zero or -1 for now.
- `options` - Optional options, which can be used to specify the recording file format. Supported options are PJMEDIA_FILE_WRITE_PCM, PJMEDIA_FILE_WRITE_ALAW, and PJMEDIA_FILE_WRITE_ULAW. Default is zero or PJMEDIA_FILE_WRITE_PCM.

**~AudioMediaRecorder**()

Virtual destructor.

*Public Static Functions*

*AudioMediaRecorder* * **typecastFromAudioMedia**(*AudioMedia* * media)

Typecast from base class *AudioMedia*.

This is useful for application written in language that does not support downcasting such as Python.

**Return**

The object as *AudioMediaRecorder* instance

**Parameters**

- `media` - The object to be downcasted

*Private Members*

int **recorderId**

Recorder Id.

*class* **AudioDevInfo**

Audio device information structure.

*Public Functions*

void **fromPj**(const pjmedia_aud_dev_info & dev_info)

Construct from pjmedia_aud_dev_info.

**~AudioDevInfo**()

Destructor.

*Public Members*

string **name**

The device name.

unsigned **inputCount**

Maximum number of input channels supported by this device.

If the value is zero, the device does not support input operation (i.e. it is a playback only device).

unsigned **outputCount**

Maximum number of output channels supported by this device.

If the value is zero, the device does not support output operation (i.e. it is an input only device).

unsigned **defaultSamplesPerSec**

Default sampling rate.

string **driver**

The underlying driver name.

unsigned **caps**

Device capabilities, as bitmask combination of pjmedia_aud_dev_cap.

unsigned **routes**

Supported audio device routes, as bitmask combination of pjmedia_aud_dev_route.

The value may be zero if the device does not support audio routing.

*MediaFormatVector* **extFmt**

Array of supported extended audio formats.

*class* **AudDevManager**

Audio device manager.

*Public Functions*

int **getCaptureDev**()

Get currently active capture sound devices.

If sound devices has not been created, it is possible that the function returns -1 as device IDs.

**Return**

Device ID of the capture device.

*AudioMedia* & **getCaptureDevMedia**()

Get the *AudioMedia* of the capture audio device.

**Return**

Audio media for the capture device.

int **getPlaybackDev**()

Get currently active playback sound devices.

If sound devices has not been created, it is possible that the function returns -1 as device IDs.

**Return**

Device ID of the playback device.

*AudioMedia* & **getPlaybackDevMedia**()

Get the *AudioMedia* of the speaker/playback audio device.

**Return**

Audio media for the speaker/playback device.

void **setCaptureDev**(int capture_dev)

Select or change capture sound device.

Application may call this function at any time to replace current sound device.

**Parameters**

- `capture_dev` - Device ID of the capture device.

void **setPlaybackDev**(int playback_dev)

Select or change playback sound device.

Application may call this function at any time to replace current sound device.

**Parameters**

- `playback_dev` - Device ID of the playback device.

const *AudioDevInfoVector* & **enumDev**()

Enum all audio devices installed in the system.

**Return**

The list of audio device info.

void **setNullDev**()

Set pjsua to use null sound device.

The null sound device only provides the timing needed by the conference bridge, and will not interract with any hardware.

*MediaPort* * **setNoDev**()

Disconnect the main conference bridge from any sound devices, and let application connect the bridge to it's own sound device/master port.

**Return**

The port interface of the conference bridge, so that application can connect this to it's own sound device or master port.

void **setEcOptions**(unsigned tail_msec, unsigned options)

Change the echo cancellation settings.

The behavior of this function depends on whether the sound device is currently active, and if it is, whether device or software AEC is being used.

If the sound device is currently active, and if the device supports AEC, this function will forward the change request to the device and it will be up to the device on whether support the request. If software AEC is being used (the software EC will be used if the device does not support AEC), this function will change the software EC settings. In all cases, the setting will be saved for future opening of the sound device.

If the sound device is not currently active, this will only change the default AEC settings and the setting will be applied next time the sound device is opened.

**Parameters**

- `tail_msec` - The tail length, in miliseconds. Set to zero to disable AEC.
- `options` - Options to be passed to pjmedia_echo_create(). Normally the value should be zero.

unsigned **getEcTail**()

Get current echo canceller tail length.

**Return**

The EC tail length in milliseconds, If AEC is disabled, the value will be zero.

bool **sndIsActive**()

Check whether the sound device is currently active.

The sound device may be inactive if the application has set the auto close feature to non-zero (the sndAutoCloseTime setting in *MediaConfig*), or if null sound device or no sound device has been configured via the *setNoDev()* function.

void **refreshDevs**()

Refresh the list of sound devices installed in the system.

This method will only refresh the list of audio device so all active audio streams will be unaffected. After refreshing the device list, application MUST make sure to update all index references to audio devices before calling any method that accepts audio device index as its parameter.

unsigned **getDevCount**()

Get the number of sound devices installed in the system.

**Return**

The number of sound devices installed in the system.

*AudioDevInfo* **getDevInfo**(int id)

Get device information.

**Return**

The device information which will be filled in by this method once it returns successfully.

**Parameters**

- `id` - The audio device ID.

int **lookupDev**(const string & drv_name, const string & dev_name)

Lookup device index based on the driver and device name.

**Return**

The device ID. If the device is not found, *Error* will be thrown.

**Parameters**

- `drv_name` - The driver name.
- `dev_name` - The device name.

string **capName**(pjmedia_aud_dev_cap cap)

Get string info for the specified capability.

**Return**

Capability name.

**Parameters**

- `cap` - The capability ID.

void **setExtFormat**(const *MediaFormatAudio* & format, bool keep = true)

This will configure audio format capability (other than PCM) to the sound device being used.

If sound device is currently active, the method will forward the setting to the sound device instance to be applied immediately, if it supports it.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_EXT_FORMAT capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

Note that in case the setting is kept for future use, it will be applied to any devices, even when application has changed the sound device to be used.

**Parameters**

- `format` - The audio format.
- `keep` - Specify whether the setting is to be kept for future use.

*MediaFormatAudio* **getExtFormat**()

Get the audio format capability (other than PCM) of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_EXT_FORMAT capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio format.

void **setInputLatency**(unsigned latency_msec, bool keep = true)

This will configure audio input latency control or query capability to the sound device being used.

If sound device is currently active, the method will forward the setting to the sound device instance to be applied immediately, if it supports it.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_INPUT_LATENCY capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

Note that in case the setting is kept for future use, it will be applied to any devices, even when application has changed the sound device to be used.

**Parameters**

- `latency_msec` - The input latency.
- `keep` - Specify whether the setting is to be kept for future use.

unsigned **getInputLatency**()

Get the audio input latency control or query capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_INPUT_LATENCY capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio input latency.

void **setOutputLatency**(unsigned latency_msec, bool keep = true)

This will configure audio output latency control or query capability to the sound device being used.

If sound device is currently active, the method will forward the setting to the sound device instance to be applied immediately, if it supports it.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_OUTPUT_LATENCY capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

Note that in case the setting is kept for future use, it will be applied to any devices, even when application has changed the sound device to be used.

**Parameters**

- `latency_msec` - The output latency.
- `keep` - Specify whether the setting is to be kept for future use.

unsigned **getOutputLatency**()

Get the audio output latency control or query capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_OUTPUT_LATENCY capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio output latency.

void **setInputVolume**(unsigned volume, bool keep = true)

This will configure audio input volume level capability to the sound device being used.

If sound device is currently active, the method will forward the setting to the sound device instance to be applied immediately, if it supports it.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_INPUT_VOLUME_SETTING capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

Note that in case the setting is kept for future use, it will be applied to any devices, even when application has changed the sound device to be used.

**Parameters**

- `volume` - The input volume level, in percent.
- `keep` - Specify whether the setting is to be kept for future use.

unsigned **getInputVolume**()

Get the audio input volume level capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_INPUT_VOLUME_SETTING capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown. *

**Return**

The audio input volume level, in percent.

void **setOutputVolume**(unsigned volume, bool keep = true)

This will configure audio output volume level capability to the sound device being used.

If sound device is currently active, the method will forward the setting to the sound device instance to be applied immediately, if it supports it.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_OUTPUT_VOLUME_SETTING capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

Note that in case the setting is kept for future use, it will be applied to any devices, even when application has changed the sound device to be used.

**Parameters**

- `volume` - The output volume level, in percent.
- `keep` - Specify whether the setting is to be kept for future use.

unsigned **getOutputVolume**()

Get the audio output volume level capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_OUTPUT_VOLUME_SETTING capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio output volume level, in percent.

unsigned **getInputSignal**()

Get the audio input signal level capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_INPUT_SIGNAL_METER capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio input signal level, in percent.

unsigned **getOutputSignal**()

Get the audio output signal level capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_OUTPUT_SIGNAL_METER capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio output signal level, in percent.

void **setInputRoute**(pjmedia_aud_dev_route route, bool keep = true)

This will configure audio input route capability to the sound device being used.

If sound device is currently active, the method will forward the setting to the sound device instance to be applied immediately, if it supports it.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_INPUT_ROUTE capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

Note that in case the setting is kept for future use, it will be applied to any devices, even when application has changed the sound device to be used.

**Parameters**

- `route` - The audio input route.
- `keep` - Specify whether the setting is to be kept for future use.

pjmedia_aud_dev_route **getInputRoute**()

Get the audio input route capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_INPUT_ROUTE capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio input route.

void **setOutputRoute**(pjmedia_aud_dev_route route, bool keep = true)

This will configure audio output route capability to the sound device being used.

If sound device is currently active, the method will forward the setting to the sound device instance to be applied immediately, if it supports it.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_OUTPUT_ROUTE capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

Note that in case the setting is kept for future use, it will be applied to any devices, even when application has changed the sound device to be used.

**Parameters**

- `route` - The audio output route.
- `keep` - Specify whether the setting is to be kept for future use.

pjmedia_aud_dev_route **getOutputRoute**()

Get the audio output route capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_OUTPUT_ROUTE capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio output route.

void **setVad**(bool enable, bool keep = true)

This will configure audio voice activity detection capability to the sound device being used.

If sound device is currently active, the method will forward the setting to the sound device instance to be applied immediately, if it supports it.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_VAD capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

Note that in case the setting is kept for future use, it will be applied to any devices, even when application has changed the sound device to be used.

**Parameters**

- `enable` - Enable/disable voice activity detection feature. Set true to enable.
- `keep` - Specify whether the setting is to be kept for future use.

bool **getVad**()

Get the audio voice activity detection capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_VAD capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio voice activity detection feature.

void **setCng**(bool enable, bool keep = true)

This will configure audio comfort noise generation capability to the sound device being used.

If sound device is currently active, the method will forward the setting to the sound device instance to be applied immediately, if it supports it.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_CNG capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

Note that in case the setting is kept for future use, it will be applied to any devices, even when application has changed the sound device to be used.

**Parameters**

- `enable` - Enable/disable comfort noise generation feature. Set true to enable.
- `keep` - Specify whether the setting is to be kept for future use.

bool **getCng**()

Get the audio comfort noise generation capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_CNG capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio comfort noise generation feature.

void **setPlc**(bool enable, bool keep = true)

This will configure audio packet loss concealment capability to the sound device being used.

If sound device is currently active, the method will forward the setting to the sound device instance to be applied immediately, if it supports it.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_PLC capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

Note that in case the setting is kept for future use, it will be applied to any devices, even when application has changed the sound device to be used.

**Parameters**

- `enable` - Enable/disable packet loss concealment feature. Set true to enable.
- `keep` - Specify whether the setting is to be kept for future use.

bool **getPlc**()

Get the audio packet loss concealment capability of the sound device being used.

If sound device is currently active, the method will forward the request to the sound device. If sound device is currently inactive, and if application had previously set the setting and mark the setting as kept, then that setting will be returned. Otherwise, this method will raise error.

This method is only valid if the device has PJMEDIA_AUD_DEV_CAP_PLC capability in *AudioDevInfo.caps* flags, otherwise *Error* will be thrown.

**Return**

The audio packet loss concealment feature.

*Private Functions*

**AudDevManager**()

Constructor.

**~AudDevManager**()

Destructor.

void **clearAudioDevList**()

int **getActiveDev**(bool is_capture)

*Private Members*

*AudioDevInfoVector* **audioDevList**

*AudioMedia* * **devMedia**

*Friends*

friend class **Endpoint**

*class* **CodecInfo**

This structure describes codec information.

*Public Functions*

void **fromPj**(const pjsua_codec_info & codec_info)

Construct from pjsua_codec_info.

*Public Members*

string **codecId**

Codec unique identification.

pj_uint8_t **priority**

Codec priority (integer 0-255).

string **desc**

Codec description.

## 12.4 call.hpp

PJSUA2 Call manipulation.

*namespace* **pj**

PJSUA2 API is inside pj namespace.

*Typedefs*

typedef void * **CodecParam**

Codec parameters, corresponds to pjmedia_codec_param or pjmedia_vid_codec_param.

typedef void * **MediaStream**

*Media* stream, corresponds to pjmedia_stream.

typedef void * **MediaTransport**

*Media* transport, corresponds to pjmedia_transport.

typedef std::vector< *CallMediaInfo* > **CallMediaInfoVector**

Array of call media info.

*class* **MathStat**

This structure describes statistics state.

*Public Functions*

**MathStat**()

Default constructor.

void **fromPj**(const pj_math_stat & prm)

Convert from pjsip.

*Public Members*

int **n**

number of samples

int **max**

maximum value

int **min**

minimum value

int **last**

last value

int **mean**

mean

*class* **RtcpStreamStat**

Unidirectional RTP stream statistics.

*Public Functions*

void **fromPj**(const pjmedia_rtcp_stream_stat & prm)

Convert from pjsip.

*Public Members*

*TimeVal* **update**

Time of last update.

unsigned **updateCount**

Number of updates (to calculate avg)

unsigned **pkt**

Total number of packets.

unsigned **bytes**

Total number of payload/bytes.

unsigned **discard**

Total number of discarded packets.

unsigned **loss**

Total number of packets lost.

unsigned **reorder**

Total number of out of order packets.

unsigned **dup**

Total number of duplicates packets.

*MathStat* **lossPeriodUsec**

Loss period statistics.

unsigned **burst**

Burst/sequential packet lost detected.

unsigned **random**

Random packet lost detected.

struct pj::RtcpStreamStat::@0 **lossType**

Types of loss detected.

*MathStat* **jitterUsec**

Jitter statistics.

*class* **RtcpSdes**

RTCP SDES structure.

*Public Functions*

void **fromPj**(const pjmedia_rtcp_sdes & prm)

Convert from pjsip.

*Public Members*

string **cname**

RTCP SDES type CNAME.

string **name**

RTCP SDES type NAME.

string **email**

RTCP SDES type EMAIL.

string **phone**

RTCP SDES type PHONE.

string **loc**

RTCP SDES type LOC.

string **tool**

RTCP SDES type TOOL.

string **note**

RTCP SDES type NOTE.

*class* **RtcpStat**

Bidirectional RTP stream statistics.

*Public Functions*

void **fromPj**(const pjmedia_rtcp_stat & prm)

Convert from pjsip.

*Public Members*

*TimeVal* **start**

Time when session was created.

*RtcpStreamStat* **txStat**

Encoder stream statistics.

*RtcpStreamStat* **rxStat**

Decoder stream statistics.

*MathStat* **rttUsec**

Round trip delay statistic.

pj_uint32_t **rtpTxLastTs**

Last TX RTP timestamp.

pj_uint16_t **rtpTxLastSeq**

Last TX RTP sequence.

*MathStat* **rxIpdvUsec**

Statistics of IP packet delay variation in receiving direction.

It is only used when PJMEDIA_RTCP_STAT_HAS_IPDV is set to non-zero.

*MathStat* **rxRawJitterUsec**

Statistic of raw jitter in receiving direction.

It is only used when PJMEDIA_RTCP_STAT_HAS_RAW_JITTER is set to non-zero.

*RtcpSdes* **peerSdes**

Peer SDES.

*class* **JbufState**

This structure describes jitter buffer state.

*Public Functions*

void **fromPj**(const pjmedia_jb_state & prm)

Convert from pjsip.

*Public Members*

unsigned **frameSize**

Individual frame size, in bytes.

unsigned **minPrefetch**

Minimum allowed prefetch, in frms.

unsigned **maxPrefetch**

Maximum allowed prefetch, in frms.

unsigned **burst**

Current burst level, in frames.

unsigned **prefetch**

Current prefetch value, in frames.

unsigned **size**

Current buffer size, in frames.

unsigned **avgDelayMsec**

Average delay, in ms.

unsigned **minDelayMsec**

Minimum delay, in ms.

unsigned **maxDelayMsec**

Maximum delay, in ms.

unsigned **devDelayMsec**

Standard deviation of delay, in ms.

unsigned **avgBurst**

Average burst, in frames.

unsigned **lost**

Number of lost frames.

unsigned **discard**

Number of discarded frames.

unsigned **empty**

Number of empty on GET events.

*class* **SdpSession**

This structure describes SDP session description.

It corresponds to the pjmedia_sdp_session structure.

*Public Functions*

void **fromPj**(const pjmedia_sdp_session & sdp)

Convert from pjsip.

*Public Members*

string **wholeSdp**

The whole SDP as a string.

void * **pjSdpSession**

Pointer to its original pjmedia_sdp_session.

Only valid when the struct is converted from PJSIP's pjmedia_sdp_session.

*class* **MediaFmtChangedEvent**

This structure describes a media format changed event.

*Public Members*

unsigned **newWidth**

The new width.

unsigned **newHeight**

The new height.

*class* **MediaEvent**

This structure describes a media event.

It corresponds to the pjmedia_event structure.

*Public Functions*

void **fromPj**(const pjmedia_event & ev)

Convert from pjsip.

*Public Members*

pjmedia_event_type **type**

The event type.

*MediaFmtChangedEvent* **fmtChanged**

*Media* format changed event data.

*GenericData* **ptr**

Pointer to storage to user event data, if it's outside this struct.

union pj::MediaEvent::@1 **data**

Additional data/parameters about the event.

The type of data will be specific to the event type being reported.

void * **pjMediaEvent**

Pointer to original pjmedia_event.

Only valid when the struct is converted from PJSIP's pjmedia_event.

*class* **MediaTransportInfo**

This structure describes media transport informations.

It corresponds to the pjmedia_transport_info structure.

*Public Functions*

void **fromPj**(const pjmedia_transport_info & info)

Convert from pjsip.

*Public Members*

*SocketAddress* **srcRtpName**

Remote address where RTP originated from.

*SocketAddress* **srcRtcpName**

Remote address where RTCP originated from.

*class* **CallSetting**

*Call* settings.

*Public Functions*

**CallSetting**(pj_bool_t useDefaultValues = false)

Default constructor initializes with empty or default values.

bool **isEmpty**()

Check if the settings are set with empty values.

**Return**

True if the settings are empty.

void **fromPj**(const pjsua_call_setting & prm)

Convert from pjsip.

pjsua_call_setting **toPj**()

Convert to pjsip.

*Public Members*

unsigned **flag**

Bitmask of pjsua_call_flag constants.

Default: PJSUA_CALL_INCLUDE_DISABLED_MEDIA

unsigned **reqKeyframeMethod**

This flag controls what methods to request keyframe are allowed on the call.

Value is bitmask of pjsua_vid_req_keyframe_method.

Default: PJSUA_VID_REQ_KEYFRAME_SIP_INFO | PJSUA_VID_REQ_KEYFRAME_RTCP_PLI

unsigned **audioCount**

Number of simultaneous active audio streams for this call.

Setting this to zero will disable audio in this call.

Default: 1

unsigned **videoCount**

Number of simultaneous active video streams for this call.

Setting this to zero will disable video in this call.

Default: 1 (if video feature is enabled, otherwise it is zero)

*class* **CallMediaInfo**

*Call* media information.

*Public Functions*

**CallMediaInfo**()

Default constructor.

void **fromPj**(const pjsua_call_media_info & prm)

Convert from pjsip.

*Public Members*

unsigned **index**

*Media* index in SDP.

pjmedia_type **type**

*Media* type.

pjmedia_dir **dir**

*Media* direction.

pjsua_call_media_status **status**

*Call* media status.

int **audioConfSlot**

The conference port number for the call.

Only valid if the media type is audio.

pjsua_vid_win_id **videoIncomingWindowId**

The window id for incoming video, if any, or PJSUA_INVALID_ID.

Only valid if the media type is video.

pjmedia_vid_dev_index **videoCapDev**

The video capture device for outgoing transmission, if any, or PJMEDIA_VID_INVALID_DEV.

Only valid if the media type is video.

*class* **CallInfo**

*Call* information.

Application can query the call information by calling *Call::getInfo()*.

*Public Functions*

void **fromPj**(const pjsua_call_info & pci)

Convert from pjsip.

*Public Members*

pjsua_call_id **id**

*Call* identification.

pjsip_role_e **role**

Initial call role (UAC == caller)

pjsua_acc_id **accId**

The account ID where this call belongs.

string **localUri**

Local URI.

string **localContact**

Local Contact.

string **remoteUri**

Remote URI.

string **remoteContact**

Remote contact.

string **callIdString**

Dialog Call-ID string.

*CallSetting* **setting**

*Call* setting.

pjsip_inv_state **state**

*Call* state.

string **stateText**

Text describing the state.

pjsip_status_code **lastStatusCode**

Last status code heard, which can be used as cause code.

string **lastReason**

The reason phrase describing the last status.

*CallMediaInfoVector* **media**

Array of active media information.

*CallMediaInfoVector* **provMedia**

Array of provisional media information.

This contains the media info in the provisioning state, that is when the media session is being created/updated (SDP offer/answer is on progress).

*TimeVal* **connectDuration**

Up-to-date call connected duration (zero when call is not established)

*TimeVal* **totalDuration**

Total call duration, including set-up time.

bool **remOfferer**

Flag if remote was SDP offerer.

unsigned **remAudioCount**

Number of audio streams offered by remote.

unsigned **remVideoCount**

Number of video streams offered by remote.

*class* **StreamInfo**

*Media* stream info.

*Public Functions*

void **fromPj**(const pjsua_stream_info & info)

Convert from pjsip.

*Public Members*

pjmedia_type **type**

*Media* type of this stream.

pjmedia_tp_proto **proto**

Transport protocol (RTP/AVP, etc.)

pjmedia_dir **dir**

*Media* direction.

*SocketAddress* **remoteRtpAddress**

Remote RTP address.

*SocketAddress* **remoteRtcpAddress**

Optional remote RTCP address.

unsigned **txPt**

Outgoing codec payload type.

unsigned **rxPt**

Incoming codec payload type.

string **codecName**

Codec name.

unsigned **codecClockRate**

Codec clock rate.

*CodecParam* **codecParam**

Optional codec param.

*class* **StreamStat**

*Media* stream statistic.

*Public Functions*

void **fromPj**(const pjsua_stream_stat & prm)

Convert from pjsip.

*Public Members*

*RtcpStat* **rtcp**

RTCP statistic.

*JbufState* **jbuf**

Jitter buffer statistic.

*class* **OnCallStateParam**

This structure contains parameters for *Call::onCallState()* callback.

*Public Members*

*SipEvent* **e**

Event which causes the call state to change.

*class* **OnCallTsxStateParam**

This structure contains parameters for *Call::onCallTsxState()* callback.

*Public Members*

*SipEvent* **e**

Transaction event that caused the state change.

*class* **OnCallMediaStateParam**

This structure contains parameters for *Call::onCallMediaState()* callback.

*class* **OnCallSdpCreatedParam**

This structure contains parameters for *Call::onCallSdpCreated()* callback.

*Public Members*

*SdpSession* **sdp**

The SDP has just been created.

*SdpSession* **remSdp**

The remote SDP, will be empty if local is SDP offerer.

*class* **OnStreamCreatedParam**

This structure contains parameters for *Call::onStreamCreated()* callback.

*Public Members*

*MediaStream* **stream**

*Media* stream.

unsigned **streamIdx**

Stream index in the media session.

*MediaPort* **pPort**

On input, it specifies the media port of the stream.

Application may modify this pointer to point to different media port to be registered to the conference bridge.

*class* **OnStreamDestroyedParam**

This structure contains parameters for *Call::onStreamDestroyed()* callback.

*Public Members*

*MediaStream* **stream**

*Media* stream.

unsigned **streamIdx**

Stream index in the media session.

*class* **OnDtmfDigitParam**

This structure contains parameters for *Call::onDtmfDigit()* callback.

*Public Members*

string **digit**

DTMF ASCII digit.

*class* **OnCallTransferRequestParam**

This structure contains parameters for *Call::onCallTransferRequest()* callback.

*Public Members*

string **dstUri**

The destination where the call will be transferred to.

pjsip_status_code **statusCode**

Status code to be returned for the call transfer request.

On input, it contains status code 200.

*CallSetting* **opt**

The current call setting, application can update this setting for the call being transferred.

*class* **OnCallTransferStatusParam**

This structure contains parameters for *Call::onCallTransferStatus()* callback.

*Public Members*

pjsip_status_code **statusCode**

Status progress of the transfer request.

string **reason**

Status progress reason.

bool **finalNotify**

If true, no further notification will be reported.

The statusCode specified in this callback is the final status.

bool **cont**

Initially will be set to true, application can set this to false if it no longer wants to receive further notification (for example, after it hangs up the call).

*class* **OnCallReplaceRequestParam**

This structure contains parameters for *Call::onCallReplaceRequest()* callback.

*Public Members*

*SipRxData* **rdata**

The incoming INVITE request to replace the call.

pjsip_status_code **statusCode**

Status code to be set by application.

Application should only return a final status (200-699)

string **reason**

Optional status text to be set by application.

*CallSetting* **opt**

The current call setting, application can update this setting for the call being replaced.

*class* **OnCallReplacedParam**

This structure contains parameters for *Call::onCallReplaced()* callback.

*Public Members*

pjsua_call_id **newCallId**

The new call id.

*class* **OnCallRxOfferParam**

This structure contains parameters for *Call::onCallRxOffer()* callback.

*Public Members*

*SdpSession* **offer**

The new offer received.

pjsip_status_code **statusCode**

Status code to be returned for answering the offer.

On input, it contains status code 200. Currently, valid values are only 200 and 488.

*CallSetting* **opt**

The current call setting, application can update this setting for answering the offer.

*class* **OnCallRedirectedParam**

This structure contains parameters for *Call::onCallRedirected()* callback.

*Public Members*

string **targetUri**

The current target to be tried.

*SipEvent* **e**

The event that caused this callback to be called.

This could be the receipt of 3xx response, or 4xx/5xx response received for the INVITE sent to subsequent targets, or empty (e.type == PJSIP_EVENT_UNKNOWN) if this callback is called from within *Call::processRedirect()* context.

*class* **OnCallMediaEventParam**

This structure contains parameters for *Call::onCallMediaEvent()* callback.

*Public Members*

unsigned **medIdx**

The media stream index.

*MediaEvent* **ev**

The media event.

*class* **OnCallMediaTransportStateParam**

This structure contains parameters for *Call::onCallMediaTransportState()* callback.

*Public Members*

unsigned **medIdx**

The media index.

pjsua_med_tp_st **state**

The media transport state.

pj_status_t **status**

The last error code related to the media transport state.

int **sipErrorCode**

Optional SIP error code.

*class* **OnCreateMediaTransportParam**

This structure contains parameters for *Call::onCreateMediaTransport()* callback.

*Public Members*

unsigned **mediaIdx**

The media index in the SDP for which this media transport will be used.

*MediaTransport* **mediaTp**

The media transport which otherwise will be used by the call has this callback not been implemented.

Application can change this to its own instance of media transport to be used by the call.

unsigned **flags**

Bitmask from pjsua_create_media_transport_flag.

*class* **CallOpParam**

This structure contains parameters for *Call::answer()*, *Call::hangup()*, *Call::reinvite()*, *Call::update()*, *Call::xfer()*, *Call::xferReplaces()*, *Call::setHold()*.

*Public Functions*

**CallOpParam**(bool useDefaultCallSetting = false)

Default constructor initializes with zero/empty values.

Setting useDefaultCallSetting to true will initialize opt with default call setting values.

*Public Members*

*CallSetting* **opt**

The call setting.

pjsip_status_code **statusCode**

Status code.

string **reason**

Reason phrase.

unsigned **options**

Options.

*SipTxOption* **txOption**

List of headers etc to be added to outgoing response message.

Note that this message data will be persistent in all next answers/responses for this INVITE request.

*class* **CallSendRequestParam**

This structure contains parameters for *Call::sendRequest()*

*Public Functions*

**CallSendRequestParam**()

Default constructor initializes with zero/empty values.

*Public Members*

string **method**

SIP method of the request.

*SipTxOption* **txOption**

Message body and/or list of headers etc to be included in outgoing request.

*class* **CallVidSetStreamParam**

This structure contains parameters for *Call::vidSetStream()*

*Public Functions*

**CallVidSetStreamParam**()

Default constructor.

*Public Members*

int **medIdx**

Specify the media stream index.

This can be set to -1 to denote the default video stream in the call, which is the first active video stream or any first video stream if none is active.

This field is valid for all video stream operations, except PJSUA_CALL_VID_STRM_ADD.

Default: -1 (first active video stream, or any first video stream if none is active)

pjmedia_dir **dir**

Specify the media stream direction.

This field is valid for the following video stream operations: PJSUA_CALL_VID_STRM_ADD and PJSUA_CALL_VID_STRM_CHANGE_DIR.

Default: PJMEDIA_DIR_ENCODING_DECODING

pjmedia_vid_dev_index **capDev**

Specify the video capture device ID.

This can be set to PJMEDIA_VID_DEFAULT_CAPTURE_DEV to specify the default capture device as configured in the account.

This field is valid for the following video stream operations: PJSUA_CALL_VID_STRM_ADD and PJSUA_CALL_VID_STRM_CHANGE_CAP_DEV.

Default: PJMEDIA_VID_DEFAULT_CAPTURE_DEV.

*class* **Call**

*Call*.

*Public Functions*

**Call**(*Account* & acc, int call_id = PJSUA_INVALID_ID)

Constructor.

**~Call**()

Destructor.

*CallInfo* **getInfo**()

Obtain detail information about this call.

**Return**

*Call* info.

bool **isActive**()

Check if this call has active INVITE session and the INVITE session has not been disconnected.

**Return**

True if call is active.

int **getId**()

Get PJSUA-LIB call ID or index associated with this call.

**Return**

Integer greater than or equal to zero.

bool **hasMedia**()

Check if call has an active media session.

**Return**

True if yes.

*Media* * **getMedia**(unsigned med_idx)

Get media for the specified media index.

**Return**

The media or NULL if invalid or inactive.

**Parameters**

- `med_idx` - *Media* index.

pjsip_dialog_cap_status **remoteHasCap**(int htype, const string & hname, const string & token)

Check if remote peer support the specified capability.

**Return**

PJSIP_DIALOG_CAP_SUPPORTED if the specified capability is explicitly supported, see pjsip_dialog_cap_status for more info.

**Parameters**

- `htype` - The header type (pjsip_hdr_e) to be checked, which value may be:
- `hname` - If htype specifies PJSIP_H_OTHER, then the header name must be supplied in this argument. Otherwise the value must be set to empty string (“”).
- `token` - The capability token to check. For example, if *htype* is PJSIP_H_ALLOW, then *token* specifies the method names; if *htype* is PJSIP_H_SUPPORTED, then *token* specifies the extension names such as “100rel”.

void **setUserData**(*Token* user_data)

Attach application specific data to the call.

Application can then inspect this data by calling *getUserData()*.

**Parameters**

- `user_data` - Arbitrary data to be attached to the call.

*Token* **getUserData**()

Get user data attached to the call, which has been previously set with *setUserData()*.

**Return**

The user data.

pj_stun_nat_type **getRemNatType**()

Get the NAT type of remote’s endpoint.

This is a proprietary feature of PJSUA-LIB which sends its NAT type in the SDP when *natTypeInSdp* is set in *UaConfig*.

This function can only be called after SDP has been received from remote, which means for incoming call, this function can be called as soon as call is received as long as incoming call contains SDP, and for outgoing call, this function can be called only after SDP is received (normally in 200/OK response to INVITE). As a general case, application should call this function after or in onCallMediaState() callback.

**Return**

The NAT type.

**See**

*Endpoint::natGetType()*, natTypeInSdp

void **makeCall**(const string & dst_uri, const *CallOpParam* & prm)

Make outgoing call to the specified URI.

**Parameters**

- `dst_uri` - URI to be put in the To header (normally is the same as the target URI).
- `prm.opt` - Optional call setting.
- `prm.txOption` - Optional headers etc to be added to outgoing INVITE request.

void **answer**(const *CallOpParam* & prm)

Send response to incoming INVITE request with call setting param.

Depending on the status code specified as parameter, this function may send provisional response, establish the call, or terminate the call. Notes about call setting:

**Parameters**

- `prm.opt` - Optional call setting.
- `prm.statusCode` - Status code, (100-699).
- `prm.reason` - Optional reason phrase. If empty, default text will be used.
- `prm.txOption` - Optional list of headers etc to be added to outgoing response message. Note that this message data will be persistent in all next answers/responses for this INVITE request.

void **hangup**(const *CallOpParam* & prm)

Hangup call by using method that is appropriate according to the call state.

This function is different than answering the call with 3xx-6xx response (with *answer()*), in that this function will hangup the call regardless of the state and role of the call, while *answer()* only works with incoming calls on EARLY state.

**Parameters**

- `prm.statusCode` - Optional status code to be sent when we're rejecting incoming call. If the value is zero, "603/Decline" will be sent.
- `prm.reason` - Optional reason phrase to be sent when we're rejecting incoming call. If empty, default text will be used.

- `prm.txOption` - Optional list of headers etc to be added to outgoing request/response message.

void **setHold**(const *CallOpParam* & prm)

Put the specified call on hold.

This will send re-INVITE with the appropriate SDP to inform remote that the call is being put on hold. The final status of the request itself will be reported on the onCallMediaState() callback, which inform the application that the media state of the call has changed.

**Parameters**

- `prm.options` - Bitmask of pjsua_call_flag constants. Currently, only the flag PJSUA_CALL_UPDATE_CONTACT can be used.
- `prm.txOption` - Optional message components to be sent with the request.

void **reinvite**(const *CallOpParam* & prm)

Send re-INVITE to release hold.

The final status of the request itself will be reported on the onCallMediaState() callback, which inform the application that the media state of the call has changed.

**Parameters**

- `prm.opt` - Optional call setting, if empty, the current call setting will remain unchanged.
- `prm.txOption` - Optional message components to be sent with the request.

void **update**(const *CallOpParam* & prm)

Send UPDATE request.

**Parameters**

- `prm.opt` - Optional call setting, if empty, the current call setting will remain unchanged.
- `prm.txOption` - Optional message components to be sent with the request.

void **xfer**(const string & dest, const *CallOpParam* & prm)

Initiate call transfer to the specified address.

This function will send REFER request to instruct remote call party to initiate a new INVITE session to the specified destination/target.

If application is interested to monitor the successfulness and the progress of the transfer request, it can implement onCallTransferStatus() callback which will report the progress of the call transfer request.

**Parameters**

- `dest` - URI of new target to be contacted. The URI may be in name address or addr-spec format.
- `prm.txOption` - Optional message components to be sent with the request.

void **xferReplaces**(const *Call* & dest_call, const *CallOpParam* & prm)

Initiate attended call transfer.

This function will send REFER request to instruct remote call party to initiate new INVITE session to the URL of *destCall*. The party at *dest_call* then should "replace" the call with us with the new call from the REFER recipient.

**Parameters**

- `dest_call` - The call to be replaced.
- `prm.options` - Application may specify PJSUA_XFER_NO_REQUIRE_REPLACES to suppress the inclusion of "Require: replaces" in the outgoing INVITE request created by the REFER request.
- `prm.txOption` - Optional message components to be sent with the request.

void **processRedirect**(pjsip_redirect_op cmd)

Accept or reject redirection response.

Application MUST call this function after it signaled PJSIP_REDIRECT_PENDING in the onCallRedirected() callback, to notify the call whether to accept or reject the redirection to the current target. Application can use the combination of PJSIP_REDIRECT_PENDING command in onCallRedirected() callback and this function to ask for user permission before redirecting the call.

Note that if the application chooses to reject or stop redirection (by using PJSIP_REDIRECT_REJECT or PJSIP_REDIRECT_STOP respectively), the call disconnection callback will be called before this function returns. And if the application rejects the target, the onCallRedirected() callback may also be called before this function returns if there is another target to try.

**Parameters**

- `cmd` - Redirection operation to be applied to the current target. The semantic of this argument is similar to the description in the onCallRedirected() callback, except that the PJSIP_REDIRECT_PENDING is not accepted here.

void **dialDtmf**(const string & digits)

Send DTMF digits to remote using RFC 2833 payload formats.

**Parameters**

- `digits` - DTMF string digits to be sent.

void **sendInstantMessage**(const *SendInstantMessageParam* & prm)

Send instant messaging inside INVITE session.

**Parameters**

- `prm.contentType` - MIME type.
- `prm.content` - The message content.
- `prm.txOption` - Optional list of headers etc to be included in outgoing request. The body descriptor in the txOption is ignored.
- `prm.userData` - Optional user data, which will be given back when the IM callback is called.

void **sendTypingIndication**(const *SendTypingIndicationParam* & prm)

Send IM typing indication inside INVITE session.

**Parameters**

- `prm.isTyping` - True to indicate to remote that local person is currently typing an IM.
- `prm.txOption` - Optional list of headers etc to be included in outgoing request.

void **sendRequest**(const *CallSendRequestParam* & prm)

Send arbitrary request with the call.

This is useful for example to send INFO request. Note that application should not use this function to send requests which would change the invite session's state, such as re-INVITE, UPDATE, PRACK, and BYE.

**Parameters**

- `prm.method` - SIP method of the request.
- `prm.txOption` - Optional message body and/or list of headers to be included in outgoing request.

string **dump**(bool with_media, const string indent)

Dump call and media statistics to string.

**Return**

*Call* dump and media statistics string.

**Parameters**

- `with_media` - True to include media information too.
- `indent` - Spaces for left indentation.

int **vidGetStreamIdx**()

Get the media stream index of the default video stream in the call.

Typically this will just retrieve the stream index of the first activated video stream in the call. If none is active, it will return the first inactive video stream.

**Return**

The media stream index or -1 if no video stream is present in the call.

bool **vidStreamIsRunning**(int med_idx, pjmedia_dir dir)

Determine if video stream for the specified call is currently running (i.e.

has been created, started, and not being paused) for the specified direction.

**Return**

True if stream is currently running for the specified direction.

**Parameters**

- `med_idx` - *Media* stream index, or -1 to specify default video media.
- `dir` - The direction to be checked.

void **vidSetStream**(pjsua_call_vid_strm_op op, const *CallVidSetStreamParam* & param)

Add, remove, modify, and/or manipulate video media stream for the specified call.

This may trigger a re-INVITE or UPDATE to be sent for the call.

**Parameters**

- `op` - The video stream operation to be performed, possible values are pjsua_call_vid_strm_op.
- `param` - The parameters for the video stream operation (see *CallVidSetStreamParam*).

*StreamInfo* **getStreamInfo**(unsigned med_idx)

Get media stream info for the specified media index.

**Return**

The stream info.

**Parameters**

- `med_idx` - *Media* stream index.

*StreamStat* **getStreamStat**(unsigned med_idx)

Get media stream statistic for the specified media index.

**Return**

The stream statistic.

**Parameters**

- `med_idx` - *Media* stream index.

*MediaTransportInfo* **getMedTransportInfo**(unsigned med_idx)

Get media transport info for the specified media index.

**Return**

The transport info.

**Parameters**

- `med_idx` - *Media* stream index.

void **processMediaUpdate**(*OnCallMediaStateParam* & prm)

Internal function (callled by *Endpoint*( to process update to call medias when call media state changes.

void **processStateChange**(*OnCallStateParam* & prm)

Internal function (called by *Endpoint*) to process call state change.

void **onCallState**(*OnCallStateParam* & prm)

Notify application when call state has changed.

Application may then query the call info to get the detail call states by calling *getInfo()* function.

**Parameters**

- `prm` - Callback parameter.

void **onCallTsxState**(*OnCallTsxStateParam* & prm)

This is a general notification callback which is called whenever a transaction within the call has changed state.

Application can implement this callback for example to monitor the state of outgoing requests, or to answer unhandled incoming requests (such as INFO) with a final response.

**Parameters**

- `prm` - Callback parameter.

void **onCallMediaState**(*OnCallMediaStateParam* & prm)

Notify application when media state in the call has changed.

Normal application would need to implement this callback, e.g. to connect the call's media to sound device. When ICE is used, this callback will also be called to report ICE negotiation failure.

**Parameters**

- `prm` - Callback parameter.

void **onCallSdpCreated**(*OnCallSdpCreatedParam* & prm)

Notify application when a call has just created a local SDP (for initial or subsequent SDP offer/answer).

Application can implement this callback to modify the SDP, before it is being sent and/or negotiated with remote SDP, for example to apply per account/call basis codecs priority or to add custom/proprietary SDP attributes.

**Parameters**

- `prm` - Callback parameter.

void **onStreamCreated**(*OnStreamCreatedParam* & prm)

Notify application when media session is created and before it is registered to the conference bridge.

Application may return different media port if it has added media processing port to the stream. This media port then will be added to the conference bridge instead.

**Parameters**

- `prm` - Callback parameter.

void **onStreamDestroyed**(*OnStreamDestroyedParam* & prm)

Notify application when media session has been unregistered from the conference bridge and about to be destroyed.

**Parameters**

- `prm` - Callback parameter.

void **onDtmfDigit**(*OnDtmfDigitParam* & prm)

Notify application upon incoming DTMF digits.

**Parameters**

- `prm` - Callback parameter.

void **onCallTransferRequest**(*OnCallTransferRequestParam* & prm)

Notify application on call being transferred (i.e.

REFER is received). Application can decide to accept/reject transfer request by setting the code (default is 202). When this callback is not implemented, the default behavior is to accept the transfer.

**Parameters**

- `prm` - Callback parameter.

void **onCallTransferStatus**(*OnCallTransferStatusParam* & prm)

Notify application of the status of previously sent call transfer request.

Application can monitor the status of the call transfer request, for example to decide whether to terminate existing call.

**Parameters**

- `prm` - Callback parameter.

void **onCallReplaceRequest**(*OnCallReplaceRequestParam* & prm)

Notify application about incoming INVITE with Replaces header.

Application may reject the request by setting non-2xx code.

**Parameters**

- `prm` - Callback parameter.

void **onCallReplaced**(*OnCallReplacedParam* & prm)

Notify application that an existing call has been replaced with a new call.

This happens when PJSUA-API receives incoming INVITE request with Replaces header.

After this callback is called, normally PJSUA-API will disconnect this call and establish a new call *newCallId*.

**Parameters**

- `prm` - Callback parameter.

void **onCallRxOffer**(*OnCallRxOfferParam* & prm)

Notify application when call has received new offer from remote (i.e.

re-INVITE/UPDATE with SDP is received). Application can decide to accept/reject the offer by setting the code (default is 200). If the offer is accepted, application can update the call setting to be applied in the answer. When this callback is not implemented, the default behavior is to accept the offer using current call setting.

**Parameters**

- `prm` - Callback parameter.

void **onInstantMessage**(*OnInstantMessageParam* & prm)

Notify application on incoming MESSAGE request.

**Parameters**

- `prm` - Callback parameter.

void **onInstantMessageStatus**(*OnInstantMessageStatusParam* & prm)

Notify application about the delivery status of outgoing MESSAGE request.

**Parameters**

- `prm` - Callback parameter.

void **onTypingIndication**(*OnTypingIndicationParam* & prm)

Notify application about typing indication.

**Parameters**

- `prm` - Callback parameter.

pjsip_redirect_op **onCallRedirected**(*OnCallRedirectedParam* & prm)

This callback is called when the call is about to resend the INVITE request to the specified target, following the previously received redirection response.

Application may accept the redirection to the specified target, reject this target only and make the session continue to try the next target in the list if such target exists, stop the whole redirection process altogether and cause the session to be disconnected, or defer the decision to ask for user confirmation.

This callback is optional, the default behavior is to NOT follow the redirection response.

**Return**

Action to be performed for the target. Set this parameter to one of the value below:

**Parameters**

- `prm` - Callback parameter.

void **onCallMediaTransportState**(*OnCallMediaTransportStateParam* & prm)

This callback is called when media transport state is changed.

**Parameters**

- `prm` - Callback parameter.

void **onCallMediaEvent**(*OnCallMediaEventParam* & prm)

Notification about media events such as video notifications.

This callback will most likely be called from media threads, thus application must not perform heavy processing in this callback. Especially, application must not destroy the call or media in this callback. If application needs to perform more complex tasks to handle the event, it should post the task to another thread.

**Parameters**

- `prm` - Callback parameter.

void **onCreateMediaTransport**(*OnCreateMediaTransportParam* & prm)

This callback can be used by application to implement custom media transport adapter for the call, or to replace the media transport with something completely new altogether.

This callback is called when a new call is created. The library has created a media transport for the call, and it is provided as the *mediaTp* argument of this callback. The callback may change it with the instance of media transport to be used by the call.

**Parameters**

- `prm` - Callback parameter.

*Public Static Functions*

*Call* * **lookup**(int call_id)

Get the *Call* class for the specified call Id.

**Return**

The *Call* instance or NULL if not found.

**Parameters**

- `call_id` - The call ID to lookup

*Private Members*

*Account* & **acc**

pjsua_call_id **id**

*Token* **userData**

std::vector< *Media* * > **medias**

## 12.5 presence.hpp

PJSUA2 Presence Operations.

*namespace* **pj**

PJSUA2 API is inside pj namespace.

*Typedefs*

typedef std::vector< *Buddy* * > **BuddyVector**

Array of buddies.

*class* **PresenceStatus**

This describes presence status.

*Public Functions*

**PresenceStatus**()

Constructor.

*Public Members*

pjsua_buddy_status **status**

*Buddy*'s online status.

string **statusText**

Text to describe buddy's online status.

pjrpid_activity **activity**

Activity type.

string **note**

Optional text describing the person/element.

string **rpidId**

Optional RPID ID string.

*class* **BuddyConfig**

This structure describes buddy configuration when adding a buddy to the buddy list with *Buddy::create()*.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

string **uri**

*Buddy* URL or name address.

bool **subscribe**

Specify whether presence subscription should start immediately.

*class* **BuddyInfo**

This structure describes buddy info, which can be retrieved by via *Buddy::getInfo()*.

*Public Functions*

void **fromPj**(const pjsua_buddy_info & pbi)

Import from pjsip structure.

*Public Members*

string **uri**

The full URI of the buddy, as specified in the configuration.

string **contact**

*Buddy*'s Contact, only available when presence subscription has been established to the buddy.

bool **presMonitorEnabled**

Flag to indicate that we should monitor the presence information for this buddy (normally yes, unless explicitly disabled).

pjsip_evsub_state **subState**

If *presMonitorEnabled* is true, this specifies the last state of the presence subscription.

If presence subscription session is currently active, the value will be PJSIP_EVSUB_STATE_ACTIVE. If presence subscription request has been rejected, the value will be PJSIP_EVSUB_STATE_TERMINATED, and the termination reason will be specified in *subTermReason*.

string **subStateName**

String representation of subscription state.

pjsip_status_code **subTermCode**

Specifies the last presence subscription termination code.

This would return the last status of the SUBSCRIBE request. If the subscription is terminated with NOTIFY by the server, this value will be set to 200, and subscription termination reason will be given in the *subTermReason* field.

string **subTermReason**

Specifies the last presence subscription termination reason.

If presence subscription is currently active, the value will be empty.

*PresenceStatus* **presStatus**

Presence status.

*class* **Buddy**

*Buddy*.

*Public Functions*

**Buddy**()

Constructor.

**~Buddy**()

Destructor.

Note that if the *Buddy* instance is deleted, it will also delete the corresponding buddy in the PJSUA-LIB.

void **create**(*Account* & acc, const *BuddyConfig* & cfg)

Create buddy and register the buddy to PJSUA-LIB.

**Parameters**

- `acc` - The account for this buddy.
- `cfg` - The buddy config.

bool **isValid**()

Check if this buddy is valid.

**Return**

True if it is.

*BuddyInfo* **getInfo**()

Get detailed buddy info.

**Return**

*Buddy* info.

void **subscribePresence**(bool subscribe)

Enable/disable buddy's presence monitoring.

Once buddy's presence is subscribed, application will be informed about buddy's presence status changed via onBuddyState() callback.

**Parameters**

- `subscribe` - Specify true to activate presence subscription.

void **updatePresence**(void)

Update the presence information for the buddy.

Although the library periodically refreshes the presence subscription for all buddies, some application may want to refresh the buddy's presence subscription immediately, and in this case it can use this function to accomplish this.

Note that the buddy's presence subscription will only be initiated if presence monitoring is enabled for the buddy. See *subscribePresence()* for more info. Also if presence subscription for the buddy is already active, this function will not do anything.

Once the presence subscription is activated successfully for the buddy, application will be notified about the buddy's presence status in the onBuddyState() callback.

void **sendInstantMessage**(const *SendInstantMessageParam* & prm)

Send instant messaging outside dialog, using this buddy's specified account for route set and authentication.

**Parameters**

- `prm` - Sending instant message parameter.

void **sendTypingIndication**(const *SendTypingIndicationParam* & prm)

Send typing indication outside dialog.

**Parameters**

- `prm` - Sending instant message parameter.

void **onBuddyState**()

Notify application when the buddy state has changed.

Application may then query the buddy info to get the details.

*Private Members*

pjsua_buddy_id **id**

*Buddy* ID.

*Account* * **acc**

*Account*.

## 12.6 persistent.hpp

PJSUA2 Persistent Services.

*namespace* **pj**

PJSUA2 API is inside pj namespace.

*class* **PersistentObject**

This is the abstract base class of objects that can be serialized to/from persistent document.

*Public Functions*

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*class* **PersistentDocument**

This a the abstract base class for a persistent document.

A document is created either by loading from a string or a file, or by constructing it manually when writing data to it. The document then can be saved to either string or to a file. A document contains one root *ContainerNode* where all data are stored under.

Document is read and written serially, hence the order of reading must be the same as the order of writing. The *PersistentDocument* class provides API to read and write to the root node, but for more flexible operations application can use the *ContainerNode* methods instead. Indeed the read and write API in *PersistentDocument* is just a shorthand which calls the relevant methods in the *ContainerNode*. As a tip, normally application only uses the *readObject()* and *writeObject()* methods declared here to read/write top level objects, and use the macros that are explained in *ContainerNode* documentation to read/write more detailed data.

*Public Functions*

**~PersistentDocument**()

Virtual destructor.

void **loadFile**(const string & filename)

Load this document from a file.

**Parameters**

- `filename` - The file name.

void **loadString**(const string & input)

Load this document from string.

**Parameters**

- `input` - The string.

void **saveFile**(const string & filename)

Write this document to a file.

**Parameters**

- `filename` - The file name.

string **saveString**()

Write this document to string.

**Return**

The string document.

*ContainerNode* & **getRootContainer**()

Get the root container node for this document.

**Return**

The root node.

bool **hasUnread**()

Determine if there is unread element.

If yes, then app can use one of the readXxx() functions to read it.

**Return**

True if there is.

string **unreadName**()

Get the name of the next unread element.

It will throw *Error* if there is no more element to read.

**Return**

The name of the next element .

int **readInt**(const string & name = "")

Read an integer value from the document and return the value.

This will throw *Error* if the current element is not a number. The read position will be advanced to the next element.

**Return**

The value.

**Parameters**

- `name` - If specified, then the function will check if the name of the next element matches the specified name and throw *Error* if it doesn't match.

float **readNumber**(const string & name = "")

Read a float value from the document and return the value.

This will throw *Error* if the current element is not a number. The read position will be advanced to the next element.

**Return**

The value.

**Parameters**

- `name` - If specified, then the function will check if the name of the next element matches the specified name and throw *Error* if it doesn't match.

bool **readBool**(const string & name = "")

Read a boolean value from the container and return the value.

This will throw *Error* if the current element is not a boolean. The read position will be advanced to the next element.

**Return**

The value.

**Parameters**

- `name` - If specified, then the function will check if the name of the next element matches the specified name and throw *Error* if it doesn't match.

string **readString**(const string & name = "")

Read a string value from the container and return the value.

This will throw *Error* if the current element is not a string. The read position will be advanced to the next element.

**Return**

The value.

**Parameters**

- `name` - If specified, then the function will check if the name of the next element matches the specified name and throw *Error* if it doesn't match.

*StringVector* **readStringVector**(const string & name = "")

Read a string array from the container.

This will throw *Error* if the current element is not a string array. The read position will be advanced to the next element.

**Return**

The value.

**Parameters**

- `name` - If specified, then the function will check if the name of the next element matches the specified name and throw *Error* if it doesn't match.

void **readObject**(*PersistentObject* & obj)

Read the specified object from the container.

This is equal to calling PersistentObject.readObject(ContainerNode);

**Parameters**

- `obj` - The object to read.

*ContainerNode* **readContainer**(const string & name = "")

Read a container from the container.

This will throw *Error* if the current element is not an object. The read position will be advanced to the next element.

**Return**

Container object.

**Parameters**

- `name` - If specified, then the function will check if the name of the next element matches the specified name and throw *Error* if it doesn't match.

*ContainerNode* **readArray**(const string & name = "")

Read array container from the container.

This will throw *Error* if the current element is not an array. The read position will be advanced to the next element.

**Return**

Container object.

**Parameters**

- `name` - If specified, then the function will check if the name of the next element matches the specified name and throw *Error* if it doesn't match.

void **writeNumber**(const string & name, float num)

Write a number value to the container.

**Parameters**

- `name` - The name for the value in the container.
- `num` - The value to be written.

void **writeInt**(const string & name, int num)

Write a number value to the container.

**Parameters**

- `name` - The name for the value in the container.
- `num` - The value to be written.

void **writeBool**(const string & name, bool value)

Write a boolean value to the container.

**Parameters**

- `name` - The name for the value in the container.

- `value` - The value to be written.

void **writeString**(const string & name, const string & value)

Write a string value to the container.

**Parameters**

- `name` - The name for the value in the container.
- `value` - The value to be written.

void **writeStringVector**(const string & name, const *StringVector* & arr)

Write string vector to the container.

**Parameters**

- `name` - The name for the value in the container.
- `arr` - The vector to be written.

void **writeObject**(const *PersistentObject* & obj)

Write an object to the container.

This is equal to calling PersistentObject.writeObject(ContainerNode);

**Parameters**

- `obj` - The object to be written

*ContainerNode* **writeNewContainer**(const string & name)

Create and write an empty Object node that can be used as parent for subsequent write operations.

**Return**

A sub-container.

**Parameters**

- `name` - The name for the new container in the container.

*ContainerNode* **writeNewArray**(const string & name)

Create and write an empty array node that can be used as parent for subsequent write operations.

**Return**

A sub-container.

**Parameters**

- `name` - The name for the array.

*class* **container_node_internal_data**

Internal data for *ContainerNode*.

See *ContainerNode* implementation notes for more info.

*Public Members*

void * **doc**

The document.

void * **data1**

Internal data 1.

void * **data2**

Internal data 2.

*class* **ContainerNode**

A container node is a placeholder for storing other data elements, which could be boolean, number, string, array of strings, or another container.

Each data in the container is basically a name/value pair, with a type internally associated with it so that written data can be read in the correct type. Data is read and written serially, hence the order of reading must be the same as the order of writing.

Application can read data from it by using the various read methods, and write data to it using the various write methods. Alternatively, it may be more convenient to use the provided macros below to read and write the data, because these macros set the name automatically:

Implementation notes:

The *ContainerNode* class is subclass-able, but not in the usual C++ way. With the usual C++ inheritance, some methods will be made pure virtual and must be implemented by the actual class. However, doing so will require dynamic instantiation of the *ContainerNode* class, which means we will need to pass around the class as pointer, for example as the return value of *readContainer()* and *writeNewContainer()* methods. Then we will need to establish who needs or how to delete these objects, or use shared pointer mechanism, each of which is considered too inconvenient or complicated for the purpose.

So hence we use C style "inheritance", where the methods are declared in container_node_op and the data in *container_node_internal_data* structures. An implementation of *ContainerNode* class will need to set up these members with values that makes sense to itself. The methods in container_node_op contains the pointer to the actual implementation of the operation, which would be specific according to the format of the document. The methods in this *ContainerNode* class are just thin wrappers which call the implementation in the container_node_op structure.

*Public Functions*

bool **hasUnread**()

Determine if there is unread element.

If yes, then app can use one of the readXxx() functions to read it.

string **unreadName**()

Get the name of the next unread element.

int **readInt**(const string & name = "")

Read an integer value from the document and return the value.

This will throw *Error* if the current element is not a number. The read position will be advanced to the next element.

**Return**

The value.

**Parameters**

- `name` - If specified, then the function will check if the name of the next element matches the specified name and throw *Error* if it doesn't match.

float **readNumber**(const string & name = "")

Read a number value from the document and return the value.

This will throw *Error* if the current element is not a number. The read position will be advanced to the next element.

**Return**

The value.

**Parameters**

- `name` - If specified, then the function will check if the name of the next element matches the specified name and throw *Error* if it doesn't match.

bool **readBool**(const string & name = "")

Read a boolean value from the container and return the value.

This will throw *Error* if the current element is not a boolean. The read position will be advanced to the next element.

**Return**

The value.

**Parameters**

- `name` - If specified, then the function will check if the name of the next element matches the specified name and throw *Error* if it doesn't match.

string **readString**(const string & name = "")

Read a string value from the container and return the value.

This will throw *Error* if the current element is not a string. The read position will be advanced to the next element.

**Return**

The value.

**Parameters**

- `name` - If specified, then the function will check if the name of the next element matches the specified name and throw *Error* if it doesn't match.

*StringVector* **readStringVector**(const string & name = "")

Read a string array from the container.

This will throw *Error* if the current element is not a string array. The read position will be advanced to the next element.

**Return**

The value.

**Parameters**

- `name` - If specified, then the function will check if the name of the next element matches the specified name and throw *Error* if it doesn't match.

void **readObject**(*PersistentObject* & obj)

Read the specified object from the container.

This is equal to calling PersistentObject.readObject(ContainerNode);

**Parameters**

- `obj` - The object to read.

*ContainerNode* **readContainer**(const string & name = "")

Read a container from the container.

This will throw *Error* if the current element is not a container. The read position will be advanced to the next element.

**Return**

Container object.

**Parameters**

- `name` - If specified, then the function will check if the name of the next element matches the specified name and throw *Error* if it doesn't match.

*ContainerNode* **readArray**(const string & name = "")

Read array container from the container.

This will throw *Error* if the current element is not an array. The read position will be advanced to the next element.

**Return**

Container object.

**Parameters**

- `name` - If specified, then the function will check if the name of the next element matches the specified name and throw *Error* if it doesn't match.

void **writeNumber**(const string & name, float num)

Write a number value to the container.

**Parameters**

- `name` - The name for the value in the container.
- `num` - The value to be written.

void **writeInt**(const string & name, int num)

Write a number value to the container.

**Parameters**

- `name` - The name for the value in the container.
- `num` - The value to be written.

void **writeBool**(const string & name, bool value)

Write a boolean value to the container.

**Parameters**

- `name` - The name for the value in the container.
- `value` - The value to be written.

void **writeString**(const string & name, const string & value)

Write a string value to the container.

**Parameters**

- `name` - The name for the value in the container.
- `value` - The value to be written.

void **writeStringVector**(const string & name, const *StringVector* & arr)

Write string vector to the container.

**Parameters**

- `name` - The name for the value in the container.
- `arr` - The vector to be written.

void **writeObject**(const *PersistentObject* & obj)

Write an object to the container.

This is equal to calling PersistentObject.writeObject(ContainerNode);

**Parameters**

- `obj` - The object to be written

*ContainerNode* **writeNewContainer**(const string & name)

Create and write an empty Object node that can be used as parent for subsequent write operations.

**Return**

A sub-container.

**Parameters**

- `name` - The name for the new container in the container.

*ContainerNode* **writeNewArray**(const string & name)

Create and write an empty array node that can be used as parent for subsequent write operations.

**Return**

A sub-container.

**Parameters**

- `name` - The name for the array.

*Public Members*

container_node_op * **op**

Method table.

*container_node_internal_data* **data**

Internal data.

## 12.7 json.hpp

*namespace* **pj**

PJSUA2 API is inside pj namespace.

*class* **JsonDocument**

Persistent document (file) with JSON format.

*Public Functions*

**JsonDocument**()

Default constructor.

**~JsonDocument**()

Destructor.

void **loadFile**(const string & filename)

Load this document from a file.

**Parameters**

- `filename` - The file name.

void **loadString**(const string & input)

Load this document from string.

**Parameters**

- `input` - The string.

void **saveFile**(const string & filename)

Write this document to a file.

**Parameters**

- `filename` - The file name.

string **saveString**()

Write this document to string.

*ContainerNode* & **getRootContainer**()

Get the root container node for this document.

pj_json_elem * **allocElement**()

An internal function to create JSON element.

pj_pool_t * **getPool**()

An internal function to get the pool.

*Private Functions*

void **initRoot**()

*Private Members*

pj_caching_pool **cp**

*ContainerNode* **rootNode**

pj_json_elem * **root**

pj_pool_t * **pool**

## 12.8 siptypes.hpp

*namespace* **pj**

PJSUA2 API is inside pj namespace.

*Typedefs*

typedef std::vector< *SipHeader* > **SipHeaderVector**

Array of strings.

typedef std::vector< *SipMultipartPart* > **SipMultipartPartVector**

Array of multipart parts.

*class* **AuthCredInfo**

Credential information.

Credential contains information to authenticate against a service.

*Public Functions*

**AuthCredInfo**()

Default constructor.

**AuthCredInfo**(const string & scheme, const string & realm, const string & user_name, const int data_type, const string data)

Construct a credential with the specified parameters.

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

string **scheme**

The authentication scheme (e.g.

"digest").

string **realm**

Realm on which this credential is to be used.

Use "*" to make a credential that can be used to authenticate against any challenges.

string **username**

Authentication user name.

int **dataType**

Type of data that is contained in the "data" field.

Use 0 if the data contains plain text password.

string **data**

The data, which can be a plain text password or a hashed digest.

string **akaK**

Permanent subscriber key.

string **akaOp**

Operator variant key.

string **akaAmf**

Authentication Management Field.

*class* **TlsConfig**

TLS transport settings, to be specified in *TransportConfig*.

*Public Functions*

**TlsConfig**()

Default constructor initialises with default values.

pjsip_tls_setting **toPj**()

Convert to pjsip.

void **fromPj**(const pjsip_tls_setting & prm)

Convert from pjsip.

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

string **CaListFile**

Certificate of Authority (CA) list file.

string **certFile**

Public endpoint certificate file, which will be used as client- side certificate for outgoing TLS connection, and server-side certificate for incoming TLS connection.

string **privKeyFile**

Optional private key of the endpoint certificate to be used.

string **password**

Password to open private key.

pjsip_ssl_method **method**

TLS protocol method from pjsip_ssl_method.

Default is PJSIP_SSL_UNSPECIFIED_METHOD (0), which in turn will use PJSIP_SSL_DEFAULT_METHOD, which default value is PJSIP_TLSV1_METHOD.

*IntVector* **ciphers**

Ciphers and order preference.

The *Endpoint::utilSslGetAvailableCiphers()* can be used to check the available ciphers supported by backend. If the array is empty, then default cipher list of the backend will be used.

bool **verifyServer**

Specifies TLS transport behavior on the server TLS certificate verification result:

In any cases, application can inspect pjsip_tls_state_info in the callback to see the verification detail.

Default value is false.

bool **verifyClient**

Specifies TLS transport behavior on the client TLS certificate verification result:

In any cases, application can inspect pjsip_tls_state_info in the callback to see the verification detail.

Default value is PJ_FALSE.

bool **requireClientCert**

When acting as server (incoming TLS connections), reject incoming connection if client doesn't supply a TLS certificate.

This setting corresponds to SSL_VERIFY_FAIL_IF_NO_PEER_CERT flag. Default value is PJ_FALSE.

unsigned **msecTimeout**

TLS negotiation timeout to be applied for both outgoing and incoming connection, in milliseconds.

If zero, the SSL negotiation doesn't have a timeout.

Default: zero

pj_qos_type **qosType**

QoS traffic type to be set on this transport.

When application wants to apply QoS tagging to the transport, it's preferable to set this field rather than *qosParam* fields since this is more portable.

Default value is PJ_QOS_TYPE_BEST_EFFORT.

pj_qos_params **qosParams**

Set the low level QoS parameters to the transport.

This is a lower level operation than setting the *qosType* field and may not be supported on all platforms.

By default all settings in this structure are disabled.

bool **qosIgnoreError**

Specify if the transport should ignore any errors when setting the QoS traffic type/parameters.

Default: PJ_TRUE

*class* **TransportConfig**

Parameters to create a transport instance.

*Public Functions*

**TransportConfig**()

Default constructor initialises with default values.

void **fromPj**(const pjsua_transport_config & prm)

Convert from pjsip.

pjsua_transport_config **toPj**()

Convert to pjsip.

void **readObject**(const *ContainerNode* & node)

Read this object from a container node.

**Parameters**

- `node` - Container to read values from.

void **writeObject**(*ContainerNode* & node)

Write this object to a container node.

**Parameters**

- `node` - Container to write values to.

*Public Members*

unsigned **port**

UDP port number to bind locally.

This setting MUST be specified even when default port is desired. If the value is zero, the transport will be bound to any available port, and application can query the port by querying the transport info.

unsigned **portRange**

Specify the port range for socket binding, relative to the start port number specified in *port*.

Note that this setting is only applicable when the start port number is non zero.

Default value is zero.

string **publicAddress**

Optional address to advertise as the address of this transport.

Application can specify any address or hostname for this field, for example it can point to one of the interface address in the system, or it can point to the public address of a NAT router where port mappings have been configured for the application.

Note: this option can be used for both UDP and TCP as well!

string **boundAddress**

Optional address where the socket should be bound to.

This option SHOULD only be used to selectively bind the socket to particular interface (instead of 0.0.0.0), and SHOULD NOT be used to set the published address of a transport (the public_addr field should be used for that purpose).

Note that unlike public_addr field, the address (or hostname) here MUST correspond to the actual interface address in the host, since this address will be specified as bind() argument.

*TlsConfig* **tlsConfig**

This specifies TLS settings for TLS transport.

It is only be used when this transport config is being used to create a SIP TLS transport.

pj_qos_type **qosType**

QoS traffic type to be set on this transport.

When application wants to apply QoS tagging to the transport, it's preferable to set this field rather than *qosParam* fields since this is more portable.

Default is QoS not set.

pj_qos_params **qosParams**

Set the low level QoS parameters to the transport.

This is a lower level operation than setting the *qosType* field and may not be supported on all platforms.

Default is QoS not set.

*class* **TransportInfo**

This structure describes transport information returned by *Endpoint::transportGetInfo()* function.

*Public Functions*

void **fromPj**(const pjsua_transport_info & info)

Construct from pjsua_transport_info.

*Public Members*

*TransportId* **id**

PJSUA transport identification.

pjsip_transport_type_e **type**

Transport type.

string **typeName**

Transport type name.

string **info**

Transport string info/description.

unsigned **flags**

Transport flags (see pjsip_transport_flags_e).

*SocketAddress* **localAddress**

Local/bound address.

*SocketAddress* **localName**

Published address (or transport address name).

unsigned **usageCount**

Current number of objects currently referencing this transport.

*class* **SipRxData**

This structure describes an incoming SIP message.

It corresponds to the pjsip_rx_data structure in PJSIP library.

*Public Functions*

**SipRxData**()

Default constructor.

void **fromPj**(pjsip_rx_data & rdata)

Construct from PJSIP's pjsip_rx_data.

*Public Members*

string **info**

A short info string describing the request, which normally contains the request method and its CSeq.

string **wholeMsg**

The whole message data as a string, containing both the header section and message body section.

*SocketAddress* **srcAddress**

Source address of the message.

void * **pjRxData**

Pointer to original pjsip_rx_data.

Only valid when the struct is constructed from PJSIP's pjsip_rx_data.

*class* **SipTxData**

This structure describes an outgoing SIP message.

It corresponds to the pjsip_tx_data structure in PJSIP library.

*Public Functions*

**SipTxData**()

Default constructor.

void **fromPj**(pjsip_tx_data & tdata)

Construct from PJSIP's pjsip_tx_data.

*Public Members*

string **info**

A short info string describing the request, which normally contains the request method and its CSeq.

string **wholeMsg**

The whole message data as a string, containing both the header section and message body section.

*SocketAddress* **dstAddress**

Destination address of the message.

void * **pjTxData**

Pointer to original pjsip_tx_data.

Only valid when the struct is constructed from PJSIP's pjsip_tx_data.

*class* **SipTransaction**

This structure describes SIP transaction object.

It corresponds to the pjsip_transaction structure in PJSIP library.

*Public Functions*

**SipTransaction**()

Default constructor.

void **fromPj**(pjsip_transaction & tsx)

Construct from PJSIP's pjsip_transaction.

*Public Members*

pjsip_role_e **role**

Role (UAS or UAC)

string **method**

The method.

int **statusCode**

Last status code seen.

string **statusText**

Last reason phrase.

pjsip_tsx_state_e **state**

State.

*SipTxData* **lastTx**

Msg kept for retrans.

void * **pjTransaction**

pjsip_transaction.

*class* **TimerEvent**

This structure describes timer event.

*Public Members*

*TimerEntry* **entry**

The timer entry.

*class* **TsxStateEvent**

This structure describes transaction state changed event.

*Public Members*

*SipRxData* **rdata**

The incoming message.

*SipTxData* **tdata**

The outgoing message.

*TimerEntry* **timer**

The timer.

pj_status_t **status**

Transport error status.

*GenericData* **data**

Generic data.

struct pj::TsxStateEvent::@2 **src**

Event source.

*SipTransaction* **tsx**

The transaction.

pjsip_tsx_state_e **prevState**

Previous state.

pjsip_event_id_e **type**

Type of event source:

*class* **TxMsgEvent**

This structure describes message transmission event.

*Public Members*

*SipTxData* **tdata**

The transmit data buffer.

*class* **TxErrorEvent**

This structure describes transmission error event.

*Public Members*

*SipTxData* **tdata**

The transmit data.

*SipTransaction* **tsx**

The transaction.

*class* **RxMsgEvent**

This structure describes message arrival event.

*Public Members*

*SipRxData* **rdata**

The receive data buffer.

*class* **UserEvent**

This structure describes user event.

*Public Members*

*GenericData* **user1**

User data 1.

*GenericData* **user2**

User data 2.

*GenericData* **user3**

User data 3.

*GenericData* **user4**

User data 4.

*class* **SipEvent**

This structure describe event descriptor to fully identify a SIP event.

It corresponds to the pjsip_event structure in PJSIP library.

*Public Functions*

**SipEvent**()

Default constructor.

void **fromPj**(const pjsip_event & ev)

Construct from PJSIP's pjsip_event.

*Public Members*

pjsip_event_id_e **type**

The event type, can be any value of **pjsip_event_id_e**.

*TimerEvent* **timer**

Timer event.

*TsxStateEvent* **tsxState**

Transaction state has changed event.

*TxMsgEvent* **txMsg**

Message transmission event.

*TxErrorEvent* **txError**

Transmission error event.

*RxMsgEvent* **rxMsg**

Message arrival event.

*UserEvent* **user**

User event.

struct pj::SipEvent::@3 **body**

The event body, which fields depends on the event type.

void * **pjEvent**

Pointer to its original pjsip_event.

Only valid when the struct is constructed from PJSIP's pjsip_event.

*class* **SipMediaType**

SIP media type containing type and subtype.

For example, for "application/sdp", the type is "application" and the subtype is "sdp".

*Public Functions*

void **fromPj**(const pjsip_media_type & prm)

Construct from PJSIP's pjsip_media_type.

pjsip_media_type **toPj**()

Convert to PJSIP's pjsip_media_type.

*Public Members*

string **type**

*Media* type.

string **subType**

*Media* subtype.

*class* **SipHeader**

Simple SIP header.

*Public Functions*

void **fromPj**(const pjsip_hdr *)

Initiaize from PJSIP header.

pjsip_generic_string_hdr & **toPj**()

Convert to PJSIP header.

*Public Members*

string **hName**

Header name.

string **hValue**

Header value.

*Private Members*

pjsip_generic_string_hdr **pjHdr**

Interal buffer for conversion to PJSIP header.

*class* **SipMultipartPart**

This describes each multipart part.

*Public Functions*

void **fromPj**(const pjsip_multipart_part & prm)

Initiaize from PJSIP's pjsip_multipart_part.

pjsip_multipart_part & **toPj**()

Convert to PJSIP's pjsip_multipart_part.

*Public Members*

*SipHeaderVector* **headers**

Optional headers to be put in this multipart part.

*SipMediaType* **contentType**

The MIME type of the body part of this multipart part.

string **body**

The body part of tthis multipart part.

*Private Members*

pjsip_multipart_part **pjMpp**

Interal buffer for conversion to PJSIP pjsip_multipart_part.

pjsip_msg_body **pjMsgBody**

*class* **SipTxOption**

Additional options when sending outgoing SIP message.

This corresponds to pjsua_msg_data structure in PJSIP library.

*Public Functions*

bool **isEmpty**()

Check if the options are empty.

If the options are set with empty values, there will be no additional information sent with outgoing SIP message.

**Return**

True if the options are empty.

void **fromPj**(const pjsua_msg_data & prm)

Initiaize from PJSUA's pjsua_msg_data.

void **toPj**(pjsua_msg_data & msg_data)

Convert to PJSUA's pjsua_msg_data.

*Public Members*

string **targetUri**

Optional remote target URI (i.e.

Target header). If empty (""), the target will be set to the remote URI (To header). At the moment this field is only used when sending initial INVITE and MESSAGE requests.

*SipHeaderVector* **headers**

Additional message headers to be included in the outgoing message.

string **contentType**

MIME type of the message body, if application specifies the messageBody in this structure.

string **msgBody**

Optional message body to be added to the message, only when the message doesn't have a body.

*SipMediaType* **multipartContentType**

Content type of the multipart body.

If application wants to send multipart message bodies, it puts the parts in multipartParts and set the content type in multipartContentType. If the message already contains a body, the body will be added to the multipart bodies.

*SipMultipartPartVector* **multipartParts**

Array of multipart parts.

If application wants to send multipart message bodies, it puts the parts in *parts* and set the content type in *multipart_ctype*. If the message already contains a body, the body will be added to the multipart bodies.

*class* **SendInstantMessageParam**

This structure contains parameters for sending instance message methods, e.g: *Buddy::sendInstantMessage()*, *Call*:sendInstantMessage().

*Public Functions*

**SendInstantMessageParam**()

Default constructor initializes with zero/empty values.

*Public Members*

string **contentType**

MIME type.

Default is "text/plain".

string **content**

The message content.

*SipTxOption* **txOption**

List of headers etc to be included in outgoing request.

*Token* **userData**

User data, which will be given back when the IM callback is called.

*class* **SendTypingIndicationParam**

This structure contains parameters for sending typing indication methods, e.g: *Buddy::sendTypingIndication()*, *Call*:sendTypingIndication().

*Public Functions*

**SendTypingIndicationParam**()

Default constructor initializes with zero/empty values.

*Public Members*

bool **isTyping**

True to indicate to remote that local person is currently typing an IM.

*SipTxOption* **txOption**

List of headers etc to be included in outgoing request.

## 12.9 types.hpp

PJSUA2 Base Types.

*Defines*

**PJSUA2_RAISE_ERROR**(status)

Raise Error exception.

**PJSUA2_RAISE_ERROR2**(status, op)

Raise Error exception.

**PJSUA2_RAISE_ERROR3**(status, op, txt)

Raise Error exception.

**PJSUA2_CHECK_RAISE_ERROR2**(status, op)

Raise Error exception if the expression fails.

**PJSUA2_CHECK_RAISE_ERROR**(status)

Raise Error exception if the status fails.

**PJSUA2_CHECK_EXPR**(expr)

Raise Error exception if the expression fails.

*namespace* **pj**

PJSUA2 API is inside pj namespace.

*Typedefs*

typedef std::vector< std::string > **StringVector**

Array of strings.

typedef std::vector< int > **IntVector**

Array of integers.

typedef void * **Token**

Type of token, i.e.

arbitrary application user data

typedef string **SocketAddress**

Socket address, encoded as string.

The socket address contains host and port number in "host[:port]" format. The host part may contain hostname, domain name, IPv4 or IPv6 address. For IPv6 address, the address will be enclosed with square brackets, e.g. "[::1]:5060".

typedef int **TransportId**

Transport ID is an integer.

typedef void * **TransportHandle**

Transport handle, corresponds to pjsip_transport instance.

typedef void * **TimerEntry**

Timer entry, corresponds to pj_timer_entry.

typedef void * **GenericData**

Generic data.

*Enums*

**Anonymous enum**

Constants.

*Values:*

- `INVALID_ID = -1` - Invalid ID, equal to PJSUA_INVALID_ID.
- `SUCCESS = 0` - Success, equal to PJ_SUCCESS.

*class* **Error**

This structure contains information about an error that is thrown as an exception.

*Public Functions*

string **info**(bool multi_line = false)

Build error string.

**Error**()

Default constructor.

**Error**(pj_status_t prm_status, const string & prm_title, const string & prm_reason, const string & prm_src_file, int prm_src_line)

Construct an *Error* instance from the specified parameters.

If *prm_reason* is empty, it will be filled with the error description for the status code.

*Public Members*

pj_status_t **status**

The error code.

string **title**

The PJSUA API operation that throws the error.

string **reason**

The error message.

string **srcFile**

The PJSUA source file that throws the error.

int **srcLine**

The line number of PJSUA source file that throws the error.

*class* **Version**

*Version* information.

*Public Members*

int **major**

Major number.

int **minor**

Minor number.

int **rev**

Additional revision number.

string **suffix**

*Version* suffix (e.g.

"-svn")

string **full**

The full version info (e.g.

"2.1.0-svn")

unsigned **numeric**

PJLIB version number as three bytes with the following format: 0xMMIIRR00, where MM: major number, II: minor number, RR: revision number, 00: always zero for now.

*class* **TimeVal**

Representation of time value.

*Public Functions*

void **fromPj**(const pj_time_val & prm)

Convert from pjsip.

*Public Members*

long **sec**

The seconds part of the time.

long **msec**

The miliseconds fraction of the time.

## 12.10 config.hpp

PJSUA2 Base Agent Operation.

*Defines*

**PJSUA2_ERROR_HAS_EXTRA_INFO**

Specify if the Error exception info should contain operation and source file information.

CHAPTER

THIRTEEN

# APPENDIX: GENERATING THIS DOCUMENTATION

## 13.1 Requirements

This documentation is created with Sphinx and Breathe. Here are the required tools:

1. Doxygen is required. Install it for your platform.
2. The easiest way to install all the tools is with Python Package Index (PyPI). Just run this and it will install Sphinx, Breathe, and all the required tools if they are not installed:

   ```
   $ sudo pip install breathe
   ```

3. Otherwise if PyPI is not available, consult Sphinx and Breathe sites for installation instructions and you may need to install these manually:

- Sphinx
- Breathe
- docutils
- Pygments

## 13.2 Rendering The Documentation

The main source of the documentation is currently the ‘‘Trac’’ pages at https://trac.pjsip.org/repos/wiki/pjsip-doc/index. The copies in SVN are just copies for backup.

To render the documentation as HTML in *_build/html* directory:

```
$ cd $PJDIR/doc/pjsip-book
$ python fetch_trac.py
$ make
```

To build PDF, run:

```
$ make latexpdf
```

## 13.3 How to Use Integrate Book with Doxygen

Quick sample:

```
will be rendered like this:
++++++++++++++++++++++++++

This is how to quote a code with syntax coloring:

.. code-block:: c++

     pj::AudioMediaPlayer *player = new AudioMediaPlayer;
     player->createPlayer("announcement.wav");

There are many ways to refer a symbol:

* A method: :cpp:func:`pj::AudioMediaPlayer::createPlayer()`
* A method with alternate display: :cpp:func:`a method <pj::AudioMediaPlayer::createPlayer()>`
* A class :cpp:class:`pj::AudioMediaPlayer`
* A class with alternate display: :cpp:class:`a class <pj::AudioMediaPlayer>`

For that links to work, we need to display the link target declaration (a class or method)
somewhere in the doc, like this:

.. doxygenclass:: pj::AudioMediaPlayer
      :path: xml
      :members:

Alternatively we can display a single method declaration like this:

.. doxygenfunction:: pj::AudioMediaPlayer::createPlayer()
      :path: xml
      :no-link:

We can also display class declaration with specific members.

For more info see `Breathe documentation <http://michaeljones.github.io/breathe/domains.html>`_
```

### 13.3.1 will be rendered like this:

This is how to quote a code with syntax coloring:

```
pj::AudioMediaPlayer *player = new AudioMediaPlayer;
player->createPlayer("announcement.wav");
```

There are many ways to refer a symbol:

- A method: `pj::AudioMediaPlayer::createPlayer()`
- A method with alternate display: `a method`
- A class `pj::AudioMediaPlayer`
- A class with alternate display: `a class`

For that links to work, we need to display the link target declaration (a class or method) somewhere in the doc, like this:

*class* **pj::AudioMediaPlayer**

Audio *Media* Player.

*Public Functions*

**AudioMediaPlayer**()

Constructor.

void **createPlayer**(const string & file_name, unsigned options = 0)

Create a file player, and automatically add this player to the conference bridge.

**Parameters**

- `file_name` - The filename to be played. Currently only WAV files are supported, and the WAV file MUST be formatted as 16bit PCM mono/single channel (any clock rate is supported).
- `options` - Optional option flag. Application may specify PJMEDIA_FILE_NO_LOOP to prevent playback loop.

void **createPlaylist**(const *StringVector* & file_names, const string & label = "", unsigned options = 0)

Create a file playlist media port, and automatically add the port to the conference bridge.

**Parameters**

- `file_names` - Array of file names to be added to the play list. Note that the files must have the same clock rate, number of channels, and number of bits per sample.
- `label` - Optional label to be set for the media port.
- `options` - Optional option flag. Application may specify PJMEDIA_FILE_NO_LOOP to prevent looping.

void **setPos**(pj_uint32_t samples)

Set playback position.

This operation is not valid for playlist.

**Parameters**

- `samples` - The desired playback position, in samples.

**~AudioMediaPlayer**()

Virtual destructor.

*Public Static Functions*

*AudioMediaPlayer* * **typecastFromAudioMedia**(*AudioMedia* * media)

Typecast from base class *AudioMedia*.

This is useful for application written in language that does not support down-casting such as Python.

**Return**

The object as *AudioMediaPlayer* instance

**Parameters**

- `media` - The object to be downcasted

Alternatively we can display a single method declaration like this:

void **createPlayer**(const string & file_name, unsigned options = 0)

Create a file player, and automatically add this player to the conference bridge.

**Parameters**

- `file_name` - The filename to be played. Currently only WAV files are supported, and the WAV file MUST be formatted as 16bit PCM mono/single channel (any clock rate is supported).
- `options` - Optional option flag. Application may specify PJMEDIA_FILE_NO_LOOP to prevent playback loop.

We can also display class declaration with specific members.

For more info see Breathe documentation

CHAPTER

FOURTEEN

# INDICES AND TABLES

- *genindex*
- *modindex*
- *search*