EICSS

Embedded Systems

SK336IPP User Guide

SK336IPP User Guide

© EICSS

Revision 1.3

Table of Contents

Introduction 3

Objectives 3

Document Structure 3

SK336IPP Architecture 4

SK336IPP Interfaces 5

API 5

Event interface 5

Network interface 5

Media interface 5

Writing New VoIP Application 6

Including header files 6

Initializing the callbacks and registry information 6

Initializing signaling 7

Adding supported codecs 7

Registering with servers 8

Starting signaling 8

Appendix A 9

Signaling API 9

ippInitMessaging 9

ippCloseMessaging 9

ippStartMessaging 10

ippStopMessaging 10

ippAddCodec 10

ippRegister 11

ippStartCall 11

Appendix B 13

Event handlers 13

Event handler callback implemented by application 13

Signaling events (to application) 13

Event handler callback implemented by signaling 14

Application events (to signaling) 15

Appendix C 16

Network callbacks 16

Appendix D 17

Media callbacks 17

Chapter

1

Introduction

SK336IPP is a signaling stack for smart VoIP clients. It comprises highly optimized implementations of two popular signaling protocols – SIP 2.0 and IAX 2.0. Integrated with SK336 framework, the implementation provides a fast route towards developing compact VoIP clients for smart devices.

Objectives

This document serves as a reference for developers using SK336IPP stack for developing VoIP or in general, MoIP applications. It provides detailed description of stack API along with application examples.

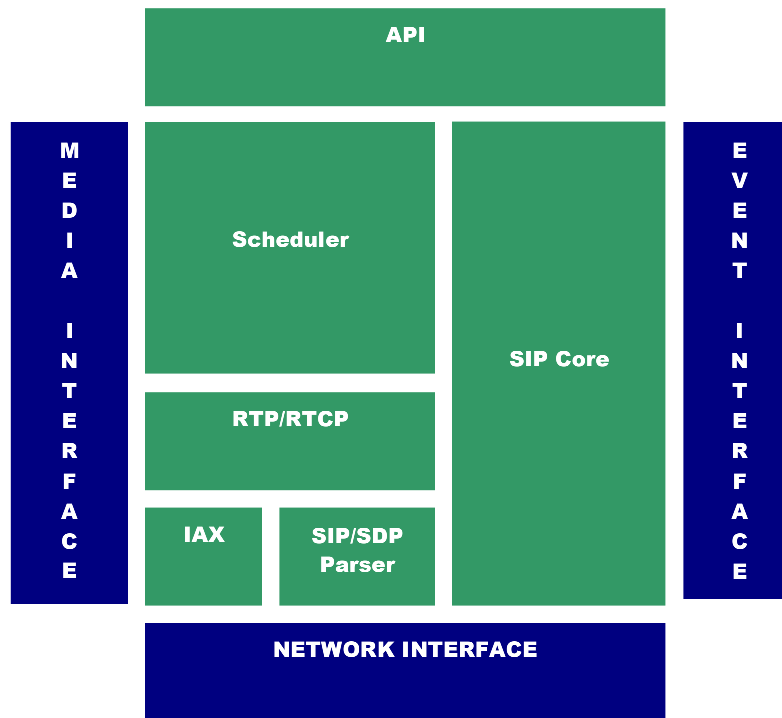
Document Structure

Chapter 1 provides an overview of SK336IPP architecture and interfaces. Chapter 2 describes the steps necessary to initialize and invoke the signaling module. Appendix A provides description of signaling API. Appendix B, C and D provide description user-exported event callback interfaces.

SK336IPP Architecture

As mentioned earlier, SK336IPP comprises optimized implementation of IAX and SIP signaling protocols. The fact that IAX can work easily behind firewalls in contrast to SIP makes the combination a very powerful one. In an advanced messaging client for example, SIP could be used for instant messaging while IAX could be used for voice.

SK336IPP has been designed with a different concept of portability – instead of providing an OS/RTOS abstraction layer to port it to different platforms, it has a built-in scheduler that eliminates need for full integration with third-party OS/RTOS. This essentially makes SK336IPP a standalone component that can be used to build complex VoIP clients using a set of API calls and callbacks.



Implemented in SK336IPP

To be implemented by application

SK336IPP Interfaces

As shown in the above figure, the stack provides three interfaces to be implemented in application:

- API
- Event interface (as callbacks, to be implemented by application)
- Network interface (as callbacks, to be implemented by application)
- Media interface (as callbacks, to be implemented by application)

These interfaces are described in the following sections.

API

The interface comprises function calls to initialize SK336IPP.

Event interface

This interface comprises two types of event handlers. The one implemented in application intercepts and responds to signaling event. The one implemented in signaling component is used by the application to trigger response events. Since signaling events can be triggered by either of the two signaling protocols (IAX and SIP), the originating context is always reported to the application so that response events can be triggered back to the originator of the event.

Network interface

Network interface is a user-supplied set of callbacks invoked by signaling module to create network endpoints and communicate information over these endpoints. Internally, a Berkley Sockets compatible API and data structures are assumed and user implementation should create compatible wrappers. The wrappers should interpret all interface data as Berkley Sockets compatible data.

Media interface

Media interface is a user-supplied set of callbacks invoked by signaling module to create media context and report/demand media packets. User implementation is responsible for encoding/decoding/capturing/rendering the data.

Chapter

2

Writing New VoIP Application

Writing new VoIP application using SK336IPP is straightforward due to availability of high-level API. This is in contrast to other commercial and open-source solutions that export low-level APIs and require a whole lot of signaling knowledge on behalf of application developer.

SK336IPP demo code can be used as template for writing new application for Windows and Linux platforms. For others, the same pattern can be followed. The following section describes the demo application step-by-step.

Including header files

Some basic header files have to be included in the main source file (and all other application level source files):

```
#include "common.h"
#include "ippapi.h"
```

Initializing the callbacks and registry information

Before initializing the signaling component, the configuration structure has to be populated with callbacks and registry information. The example code populates all three types of callback structures:

```
    /* Init callback interfaces and pass on to signaling */
    ippconf.sigCB.eventHandler             = uiEventCallback;

    ippconf.mediaCB.mediaContextCreate     = (MEDIA_CONTEXT_CREATE_CB)
mediaContextCreate;
    ippconf.mediaCB.mediaContextDestroy    = (MEDIA_CONTEXT_DESTROY_CB)
mediaContextDestroy;
    ippconf.mediaCB.mediaSetFormat         = (MEDIA_SET_FORMAT_CB)
mediaSetFormat;
    ippconf.mediaCB.mediaDecodeAndRender   =
(MEDIA_DECODE_AND_RENDER_CB) mediaDecodeAndRender;
    ippconf.mediaCB.mediaCaptureAndEncode  =
(MEDIA_CAPTURE_AND_ENCODE_CB) mediaCaptureAndEncode;

    ippconf.netCB.socketCreate = (SOCK_CREATE_CB) socket;
    ippconf.netCB.socketClose  = (SOCK_CLOSE_CB) closesocket;
```

```
    ippconf.netCB.socketBind   = (SOCK_BIND_CB) bind;
    ippconf.netCB.socketSendTo = (SOCK_SENDTO_CB) sendto;
    ippconf.netCB.socketReceiveFrom = (SOCK_RECEIVEFROM_CB) recvfrom;
    ippconf.netCB.socketIoctl  = (SOCK_IOCTL_CB) ioctlsocket;
    ippconf.netCB.inetPToN     = (INET_PTON_CB) inet_pton;
    ippconf.netCB.inetNToP     = (INET_NTOP_CB) inet_ntop;
    ippconf.netCB.netMatchSubnet    = (NET_MATCH_SUBNET_CB)
netMatchSubnet;
    ippconf.netCB.netGetLocalAddr   = (NET_GET_LOCAL_ADDR_CB)
netGetLocalAddr;
    ippconf.netCB.netGetMappedAddr  = (NET_GET_MAPPED_ADDR_CB)
netGetMappedAddr;
    ippconf.netCB.netResolveHost    = (NET_RESOLVE_HOST_CB)
netResolveHost;
```

Referring to the example, following code initializes two registries – one SIP registry and one IAX registry. As can be seen, virtually unlimited number of registry entries can be initialized and supplied to core signaling module:

```
    ippconf.sipPortUDP      = 5060;
    ippconf.iaxPortUDP      = 4569;
    ippconf.numRegistries  = 2;

    /* First registry - SIP */
    ippconf.registryInfo[0].protocol = SIG_PROTO_SIP;
    strcpy(ippconf.registryInfo[0].username, "101");
    strcpy(ippconf.registryInfo[0].password, "101");
    strcpy(ippconf.registryInfo[0].context, "");
    strcpy(ippconf.registryInfo[0].servername, "192.168.1.100");
    ippconf.registryInfo[0].serverport = ntohs(5060);
    ippconf.registryInfo[0].regRefreshTime = 0;
    ippconf.registryInfo[0].status = REG_PENDING;

    /* Second registry - IAX */
    ippconf.registryInfo[1].protocol = SIG_PROTO_IAX;
    strcpy(ippconf.registryInfo[1].username, "100");
    strcpy(ippconf.registryInfo[1].password, "100");
    strcpy(ippconf.registryInfo[1].context, "");
    strcpy(ippconf.registryInfo[1].servername, "192.168.1.100");
    ippconf.registryInfo[1].serverport = ntohs(4569);
    ippconf.registryInfo[1].regRefreshTime = 0;
    ippconf.registryInfo[1].status = REG_PENDING;
```

Initializing signaling

Signaling module can now be initializing by invoking API call with initialized configuration structure as first parameter and required signaling protocols bitmap as second parameter:

```
    /* Initialize signaling */
    err = ippInitMessaging(&ippconf, SIG_PROTO_IAX | SIG_PROTO_SIP);
```

Adding supported codecs

Supported media codecs should be added in order of preference. In the example code, ALAW is added first and thus is the preferred codec when negotiating codecs:

```
/* Add supported codecs in order of preference */
ippAddCodec(MT_AUDIO, PT_ALAW, 8000, 1, EN_ALAW);
ippAddCodec(MT_AUDIO, PT_ULAW, 8000, 1, EN_ULAW);
```

Registering with servers

Following API call is necessary to specify preferred registry refresh interval (in seconds). This is a blocking call and will return only when all registrations have completed with failure/success:

```
printf("Registering with server...");
ippRegister(3600);
```

Starting signaling

After all the initialization steps, signaling can be started. Following call starts signaling. This call never returns and thus should be called from a separate dedicated thread in a multi-threaded environment or should be the last call in OS-less environment:

```
printf("Starting signalling...\r\n");
ippStartMessaging();
```

Appendix A

Signaling API

ippInitMessaging

Initializing signaling module to known state.

Include

```
ippapi.h
```

Prototype

```
UInt32 ippInitMessaging(pIPPConfig ipconf, UInt16 proto);
```

Parameters

ipconf	I	Pointer to configuration information which also contains pointer to callback implemented in signaling to respond to application events
proto	I	Bitmap specifying signaling protocols to initialize

Returns

ERR_OK

OS_ERR_NOTENOUGHMEM

IPP_ERR_BAD_PROTO

IPP_ERR_BAD_CONFIG

IPP_ERR_NETWORK_RESOURCE

ippCloseMessaging

Initializing signaling module to known state.

Include

```
ippapi.h
```

Prototype

```
void ippCloseMessaging(void);
```

Parameters

None

Returns

None

ippStartMessaging

Start signaling module.

Include

ippapi.h

Prototype

```
UInt32 ippStartMessaging(void)
```

Parameters

None

Returns

ERR_OK

OS_ERR_INVALIDTASKID

ippStopMessaging

Start signaling module.

Include

ippapi.h

Prototype

```
void ippStopMessaging(void)
```

Parameters

None

Returns

None

ippAddCodec

Add supported media codec info. The order of addition is order of priority.

Include

ippapi.h

Prototype

```
void ippAddCodec(UInt8 mediaType, UInt8 payloadType, UInt32 clockRate, UInt8 numChan, char *encodingName)
```

Parameters

mediaType	I	One of MT_AUDIO/MT_VIDEO
payloadType	I	RTP payload type
clockRate	I	Actual clock rate corresponding to payload type
numChan	I	Number of channels associated with payload type
encodingName	I	RTP specific encoding name associated with payload

Returns

None

ippRegister

Force registration process with all configured servers.

Include

ippapi.h

Prototype

```
void ippRegister(UInt32 refreshTime)
```

Parameters

refreshTime	I	Refresh time in seconds. Use 0 to de-register

Returns

None

ippStartCall

Start a new outgoing call.

Include

ippapi.h

Prototype

```
UInt32 ippStartCall(UInt16 proto, UInt32 *context, char *ext)
```

Parameters

proto	I	Bitmap specifying signaling protocol to use If more than one protocols are defined, they will be attempted in order – first IAX, then SIP
context	O	Call context for future call reference
ext	I	Called number or URI (without sip, sips etc.)

Returns

ERR_OK

OS_ERR_NOTENOUGHMEM

IPP_ERR_BAD_CONFIG

IPP_ERR_NO_SESSION

IPP_ERR_BAD_CALLEE_ADDRESS

IPP_ERR_NO_RTP_SESSION

Appendix B

Event handlers

Signaling and application implement event handler callbacks to process events from each other. Application implemented callback is provided to signaling in configuration structure passed to ippInitMessaging().

Please note that an outgoing call should be initiated using functional interface i.e. ippStartCall(). All subsequent events shall be reported to application by calling application callback. All reported events are accompanied by call context making it possible to maintain states of multiple calls.

Event handler callback implemented by application

This callback should be implemented by application to listen to asynchronous events from signaling module.

Include

```
ippapi.h
```

Prototype

```
typedef void (*EVENT_HANDLER_CB)(void *iface, UInt32 event, UInt32 context,
void *args)
```

Parameters

```
iface        I     Pointer to structure containing pointer to callbacks to
                   process application events. Should be type casted to
                   pIPPConfig to report events to signaling callback
event        I     Event to report (see below for event details)
context      I     Call context the event should be processed in
args         I     Event specific parameters
```

Returns

```
None
```

Signaling events (to application)

Event type	Arguments	Description
SIG2UI_EV_CALL_ESTABLISHED	None	Call is established

SIG2UI_EV_INCOMING_CALL	Caller ID	New incoming call
SIG2UI_EV_ERROR	None	Error in response to a previous request
SIG2UI_EV_BUSY	None	Called number is busy
SIG2UI_EV_RINGBACK	None	Call in progress
SIG2UI_EV_HANGUP	None	Remote party hung up
SIG2UI_EV_REGISTERED	None	Registered with registrar
SIG2UI_EV_DEREGISTERED	None	De-registered from registrar
SIG2UI_EV_END_SESSION	None	Call session is no longer valid
SIG2UI_EV_IM	Pointer to message string	Incoming instant message

Event handler callback implemented by signaling

The callback is implemented by signaling and passed on to application in first parameter to application callback invocation which is data structure of following type:

```
typedef struct ipp_interface_t {
    UInt32 (*eventHandler)(UInt32 event, UInt32 *context, void *args);
} IPPInterface, *pIPPInterface;
```

Include

```
ippapi.h
```

Prototype

```
UInt32 (*eventHandler)(UInt32 event, UInt32 *context, void *args)
```

Parameters

```
event        I     Event to report (see below for event details)

context      I     Call context the event should be processed in

args         I     Event specific parameters
```

Returns

```
None
```

Application events (to signaling)

Event type	Arguments	Description/Comments
UI2SIG_EV_NEW	Called number/URI	ippStartCall() should instead be used
UI2SIG_EV_HANGUP	None	Hang up call on specified context
UI2SIG_EV_SEND_RINGING	None	Indicate call progress
UI2SIG_EV_ANSWER	None	Incoming call has been accepted
UI2SIG_EV_REJECT	None	Incoming call has been rejected
UI2SIG_EV_HOLD	None	Put specified call on hold (media streaming will pause)
UI2SIG_EV_UNHOLD	None	Take a call out of hold
UI2SIG_EV_START_VOICE	None	Start media streaming for a newly accepted call
UI2SIG_EV_SEND_DTMF	DTMF digit	Send DTMF digit
UI2SIG_EV_SEND_IM	Pointer to message string	Send instant message

Appendix C

Network callbacks

Following network callbacks are to be implemented by application. The callbacks use BSD Sockets API compliant interface and use data structures compliant to the same API.

Callback	Description
SOCK_CREATE_CB	socket()
SOCK_CLOSE_CB	close()
SOCK_BIND_CB	bind()
SOCK_SENDTO_CB	sento()
SOCK_RECEIVEFROM_CB	recvfrom()
SOCK_IOCTL_CB	ioctl() or ioctlsocket()
INET_PTON_CB	inet_pton()
INET_NTOP_CB	inet_ntop()
NET_MATCH_SUBNET_CB	This is a purely user implementation called by signaling to check whether two addresses are on the same subnet.
NET_GET_LOCAL_ADDR_CB	This is a purely user implementation called by signaling to get IPv4/IPv6 address of local interface. See demo for example.
NET_GET_MAPPED_ADDR_CB	This is a purely user implementation called by signaling to get mapped public address corresponding to given interface address (assuming client is behind NAT). Any NAT traversing algorithm may be implemented to get mapped address.
NET_RESOLVE_HOST_CB	This is a purely user implementation called by signaling to resolve FQDN to IPv4/IPv6 address. See demo for example.

Appendix D

Media callbacks

The set of callbacks are called by media transport part of SIP/IAX signaling to send/receive compressed media streams to/from application layer:

Callback	Description
MEDIA_CONTEXT_CREATE_CB	Create a media context for full-duplex streaming
MEDIA_CONTEXT_DESTROY_CB	Destroy a previously created media context
MEDIA_SET_FORMAT_CB	Set media stream format over specified context
MEDIA_DECODE_AND_RENDER_CB	Called on receiving compressed media. Should decompress media frame and render it immediately or queue it for later rendering
MEDIA_CAPTURE_AND_ENCODE_CB	Called repeatedly when media transmission is enabled. Should capture media frame, compress it and make it available to the caller (i.e. media transport part of signaling)