# VOIP Project: Final Report

*Sarfraz Nawaz*
*Mark Niebur*
*Scott Schuff*
*Athar Shiraz Siddiqui*

## *Overview*

The purpose of this document is to describe in detail the final design and implementation of our CSEE 4840 semester project: a voice-over-IP (VOIP) soft-phone. The document is divided into 3 main parts. Part I covers the design and implementation of the industry standard network protocols and application logic that we will use to implement VOIP. Part II covers the system software for the phone, and finally part III covers the hardware implementation.

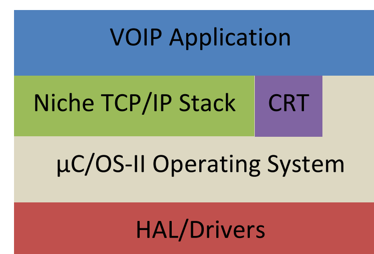
Before we dive into the details of each of these sections, we should first talk about the high level system requirements. The requirements for a soft-phone device are something like the following:

- A way to "pick up" the phone, dial a number, and see who you are dialing
- A way for the phone to ring
- A way to speak into the phone
- A way to transmit/receive voice data
- A way to hear the person on the other end
- A way to hang up

In our system, these requirements are correspondingly implemented by:

- A PS2 keyboard and the 2-line LCD display
- A speaker connected to the DE2 audio out, and the LCD
- A microphone connected to the DE2 audio in
- An Ethernet connection
- The same speaker used for ring
- The PS2 keyboard

To implement the system, we used a Nios II processor on the Altera FPGA, with a software "stack" that looks roughly as shown in the figure below. In the top layer, we have the VOIP application, including the network protocols and application logic. This is built on the system software layer, which provides a C environment, a BSD sockets API for the network communication (via the TCP/IP stack), a small operating system for threading and synchronization, and drivers that abstract away the hardware details. At the lowest level, we have hardware peripherals, implemented on the FPGA, to drive the various human interface and communications components listed above.




With the high level picture now in place, we dive into the details of the system design and implementation.

# Part I: VOIP Application Layer

The application layer consists of 2 main components: a state machine implementation of the phone behavior and the industry standard VOIP networking protocols.

## Application Framework: Phone State Machine

At the highest level, the application software is a state machine that enforces the behavior of a traditional phone. User input is taken from the keyboard, with the space bar representing the traditional phone 'hook', and the number keys used for dialing. User feedback is given via the LCD. The state machine follows the diagrams shown in figures 1 and 2. These could be shown as a single state diagram, but for the sake of clarity, we have divided it up into two diagrams: the state diagram for the caller (figure 1), and the state diagram followed by the callee (figure 2).

In the figures, the states are shown as green ellipses, and the transitions as blue boxes. Each of the transitions is numbered for a more detailed explanation in its corresponding transition table. The system spends nearly all its time in the ready state, waiting for a call to arrive or the local user to pick up the phone and start dialing. Voice data is exchanged only in the In Call state, the primary active state of the phone. Some small details are left out of the diagram for clarity (such as the handling of various error conditions).

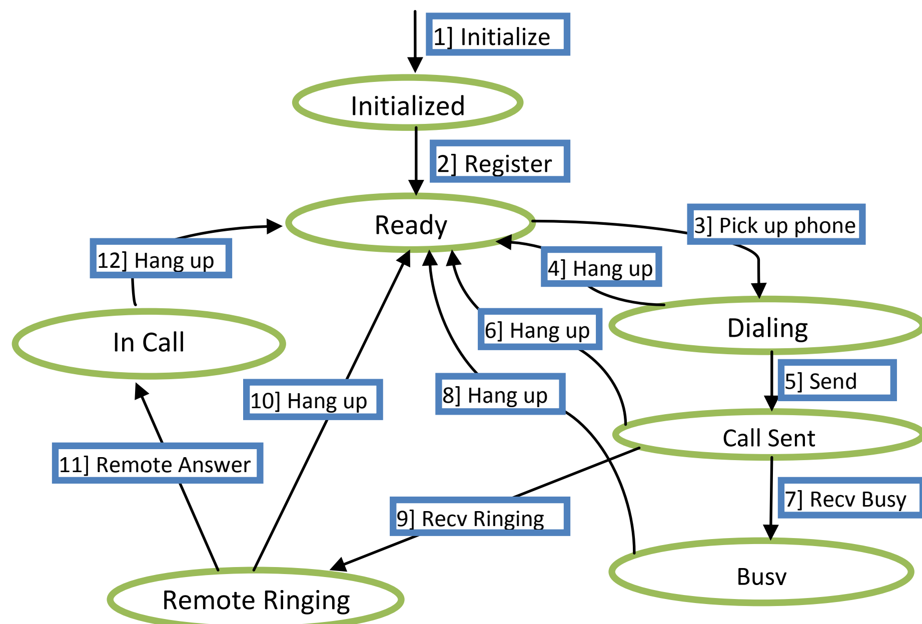



Figure 1 **State transition diagram for a caller**

| # | Description | Keyboard Input | SIP message |
|---|---|---|---|
| 1 | User enters mac address, IP address, and extension number for phone. | Enter numbers | |
| 2 | The phone is registered with the Asterisk registrar | | REGISTER |
| 3 | Local user picks up the phone | Space Bar | |
| 4 | Local user decides not to call, hangs up | Space Bar | |

| 5 | User sends call | Enter | INVITE |
|---|---|---|---|
| 6 | User gives up waiting for response, hangs up | Space Bar | |
| 7 | User gets a busy signal back from registrar | | BUSY |
| 8 | User identifies the busy tone, hangs up | Space Bar | |
| 9 | Remote extension available, we receive a message indicating it is ringing | | RINGING or TRYING |
| 10 | No one picks up at the remote side, local user hangs up | Space Bar | CANCEL |
| 11 | Remote user picks up his extension | Space Bar | OK |
| 12 | End of conversation, one side hangs up | Space Bar | BYE |

The callee state diagram is significantly simpler. A dataflow style diagram is shown in figure 3, and is how the application looks from the point of view of how data moves through it. This diagram shows the 2 main threads in the application, the state machine thread and the SIP thread. The blue arrows between these threads indicate message queues for keeping the SIP and application states in sync. The black arrows indicate network communications. The green arrows indicate data queues containing buffers of sound data moving between the sound interrupt service routine and the main thread (where it received and sent via RTP). Finally the red arrow indicates data reads/writes over the Avalon bus.

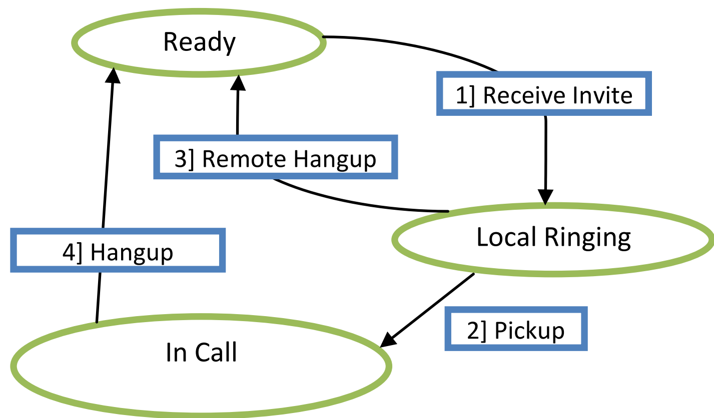



**Figure 2** State transition diagram for a caller

| # | Description | Keyboard Input | SIP Messages |
|---|---|---|---|
| 1 | We receive a call (an invite message from the registrar) | | >> INVITE |
| 2 | The local user picks up the phone, OK sent to registrar | Space Bar | OK >> |
| 3 | The remote user hangs up the phone | Space Bar | >> CANCEL |
| 4 | Conversation over, one side hangs up | Space Bar | BYE |

## Session Initiation Protocol (SIP)

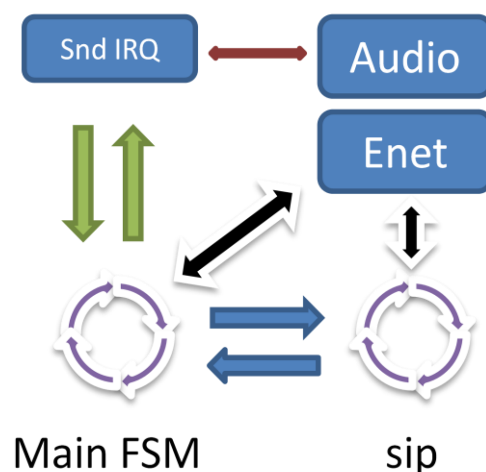



**Figure 3** Data flow diagram of application software

SIP is a signaling protocol that is commonly used in order to set up and tear down internet multimedia streams such as voice and video conferencing. It uses a text based request and response mechanism similar to HTTP called a dialog. The first line of each request in a dialog is called the request line. This contains the request or response, the recipient of the request and the protocol used. There are five standard requests, however others are supported by various SIP implementations. The five standard requests are REGISTER, INVITE, ACK, CANCEL, and OPTIONS. REGISTER is the command used to register with a SIP registrar, INVITE is used to invite a peer to a multimedia session, ACK is used to acknowledge status responses and is often the last message sent in a dialog, CANCEL is used to cancel a SIP dialog, and OPTIONS is used to find out other supported request types. After the request line, there is a message header. In the message header are tags that define properties of the dialog. There are many different tags, but in every message five specific ones are required. These are Via, From, To, Call-ID, and Cseq. The Via tag is used to denote the path of the session, From is used to denote the sender, To is used to denote the recipient of the message, Call-ID is a random string of characters used to signify a specific SIP dialog, and Cseq is used in order to show the sequence of messages being sent or received. CSeq is used because many SIP implementations use UDP as the transport mechanism, so the SIP server needs to be able to differentiate between resends and new messages. Along with the data mentioned here, several of the lines contain unique randomly generated tags. These tags are used in order to differentiate different SIP dialogs being routed through the registrar.

Along with the five required tags, other tags are also necessary depending on the message being sent. The ones that we used in our project were Contact, Content-type, and Content-length. Contact contains the registrar identifier and the location and port of the host machine sending the message. This is used to find the exact location of the SIP client, so the registrar knows where to forward messages to reach the client, and so a peer knows where to find another, should it try and bypass the registrar. Content-type is used to denote the type of content being carried in the message payload. In our project we used SDP, so the contents of the Content-type tag were statically assigned to application/sdp. Finally, Content-length gives the size in bytes of the message payload. All lines in the SIP dialog are terminated with a line

feed and carriage return. The end of the header is shown by two subsequent line feed carriage return sequences.

Replies are similar to requests. On the first line there is a status-line. This gives a number and text description of the status being returned to a response. Aside from this all other tags are kept the same with the exception of the unique tag for the recipient, as well as the contact tag, which is updated with the recipient's identifier and location. All of the status response messages fall into certain types, which are denoted by the first digit of the status number. These are as follows:

100s – Informational, gives status updates on the message that do not require acknowledgment. These are updates such as ringing and call is being forwarded.

200s – OK Only the 200 is used in this category.

300s – Redirection. Used to signify a need to change the location that messages are being sent.

400s – Client side error. This is used to denote an error that the client had interpreting or handling the message. This is used with messages such as the call was cancelled, or the client was busy.

500s – Server side error. This is used to denote an error that the server had interpreting or handling the message. This is used with messages such as the server could not locate the peer the message was being sent to.

600s – Global messaging error. This is used to signify a message with the whole SIP system. One such example is that the network is busy everywhere.

The typical flow of messages in a SIP dialog is first a request, then next a status response, then finally an acknowledgment. An example is shown below of just the client registrar messages being sent in an INVITE dialog.

Client request:

```
INVITE sip:user2@SIPregistrar.com SIP/2.0
Via: SIP/2.0/UDP user1.local.address:portnum; branch=z9hG4bK1234
From: username <user1@SIPregistrar.com>; tag=randTag
To: username2 <user2@SIPregistrar.com>
Call-ID: xyzrandomgen
Cseq 1234 INVITE
Contact: <sip:user1@user1.local.address:portnum>
Content-type: application/sdp
Content-length 263
```

This message shows the typical request from a client to the sip registrar. In the branch tag, the first seven characters must be z9hG4bK. This is unique to the SIP 2.0 protocol, and is used to identify SIP 2.0 messages. Also, the To tag is left empty to be filled in by the peer being reached. Finally, there is an SDP payload that is not shown. This is detailed later.

Registrar response:

```
SIP/2.0 100 Trying
Via: SIP/2.0/UDP user1.local.address:portnum; branch=z9hG4bK1234
From: username <user1@SIPregistrar.com>; tag=randTag
To: username2 <user2@SIPregistrar.com>
Call-ID: xyzrandomgen
Cseq 1234 INVITE
Content-length 0
```

This message shows the immediate response from the registrar. This is sent back because the protocol is UDP and so such a message is required for the client to know that its request successfully reached the registrar. All non-relevant tags have been removed, and the rest remains the same.

Register response:

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP user1.local.address:portnum; branch=z9hG4bK1234
From: username <user1@SIPregistrar.com>; tag=randTag
To: username2 <user2@SIPregistrar.com>; tag=randTag2
Call-ID: xyzrandomgen
Cseq 1234 INVITE
Contact: <sip:user2@user2.local.address:portnum>
Content-type: application/sdp
Content-length 183
```

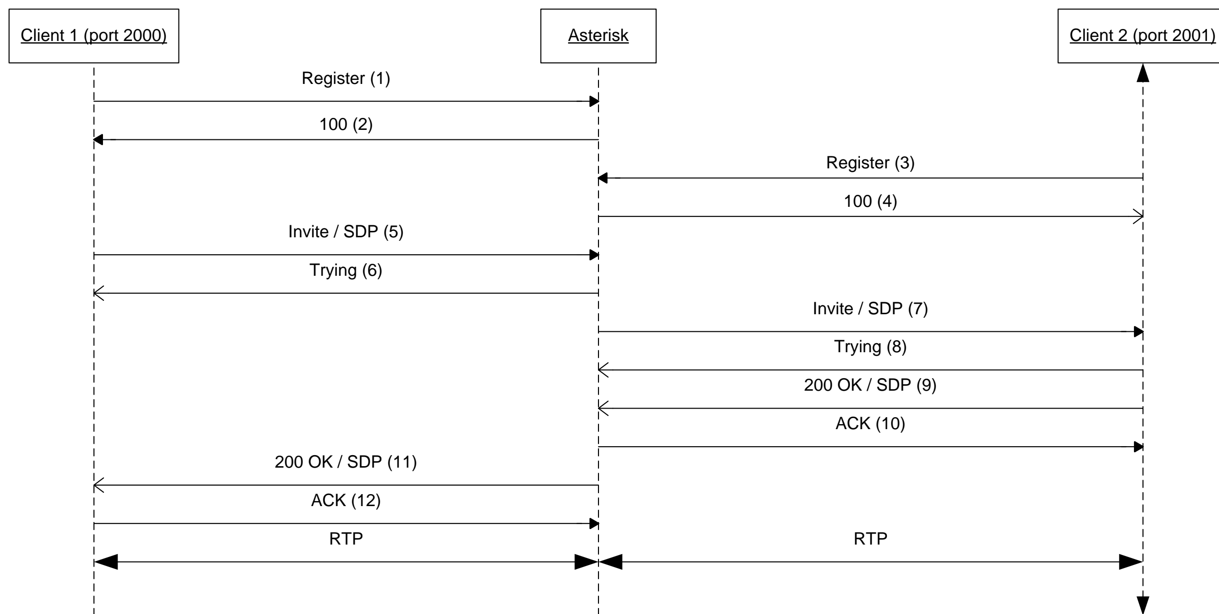
After the client being reached answers, the message is routed through the register back to the original client. In this message, the remote tag is filled in by the remote user, and the contact information is changed to match the remote peer. Likewise, the SDP payload has been changed to reflect the stats of the remote peer. After this, the local user stars an RTP session with the remote user using the statistics in the SDP payload.

Client Response:

```
ACK sip:user2@SIPregistrar.com SIP/2.0
```

Via: SIP/2.0/UDP user1.local.address:portnum; branch=z9hG4bK1234
From: username <user1@SIPregistrar.com>; tag=randTag
To: username2 <user2@SIPregistrar.com>
Call-ID: xyzrandomgen
Cseq 1234 ACK
Content-length 0

After this acknowledgment, the dialog is over. A diagram of this type of exchange is shown in the figure below.




## The Asterisk Registrar

As our VOIP registrar, we used the open source Asterisk program[1]. Asterisk by default tries to redirect the RTP media stream (audio) to go directly from the caller to the callee. In this default mode, Asterisk has 'canreinvite=yes' when the server sends re-invites to both the remote clients to ensure a direct RTP media session between the two clients. Some devices do not support this (especially if one of them is behind a NAT). The Asterisk server when configured with **'canreinvite=no'** stops the sending of the (re)INVITEs once the call is established. The RTP media stream between the caller and the callee then takes place through the registrar. We have configured Asterisk with 'canreinvite=no' in our project. Configuration of Asterisk is accomplished through a variety of '.conf' files. The modifications we have made to these are given below.

[1] Information on Asterisk is available at www.asterisk.org.

*Extensions.conf*

[others] [my-phones] exten => 2000,1,Dial(SIP/2000) exten => 2001,1,Dial(SIP/2001)

*SIP.conf*

[general] bindport=5060 bindaddr=0.0.0.0 context = my-phones canreinvite = no

[2000] type=friend context=my-phones;secret=1234 host=dynamic disallow=all allow=ulaw

[2001] type=friend context=my-phones ;secret=1234 host=dynamic disallow=all allow=ulaw

## Session Description Protocol (SDP)

SDP is used in conjunction with SIP in order to set up media sessions. The SDP message is included as the SIP payload. The protocol is deliberately kept as simple as possible. The sequence of tags must be in a specific order with only a single letter followed by an equals sign to denote each tag. The tags used in this project are as follows:

| Tag | Description |
|---|---|
| v= | This is the version tag. It gives the version number of the SDP protocol being used. No point version numbers are allowed. |
| o= | This is the origin tag. It gives the name of the client or - for name. After this the session number is given, followed by the session version. The session version starts at the same number as the session number, but is incremented for each change being given. Often times, the network time protocol is used to generate the numbers, however in our project we just used a random number generator. After the session number and version, the network type is given. IN is used to denote internet. Following that is the address type IP4 or IP6 can be used for IPv4 and IPv6 internet addresses respectively. Finally the address of the peer is given. |
| s= | This is the session name. Any string can be given, but often times, the implementation name is used. In our project, we use the string teamVOIP. ☺ |
| c= | This is the connection data. It gives the network type, address protocol and address of the peer already expanded in the origin tag. |
| t= | This gives the start time and end time of each SDP session. This will start and end the session at the given times. 0 can be used to denote any time. |
| m= | This gives the media session specifics. It includes the media type, the ports, and the protocol. Following this is a variable length list of formats that the session may be. |

|  | These are expanded with the following a= tags. |
|---|---|
| a= | These are media attributes of the media session. There are many different values that it may hold. It usually describes the accepted types of communication, the values of the media type and the timeframe of each sample. |

Just like the SIP protocol, each line is terminated with a line feed and carriage return and the end of the packet is terminated with two line feed carriage return sequences in succession. One sample SDP packet is shown below.

Sample SDP:

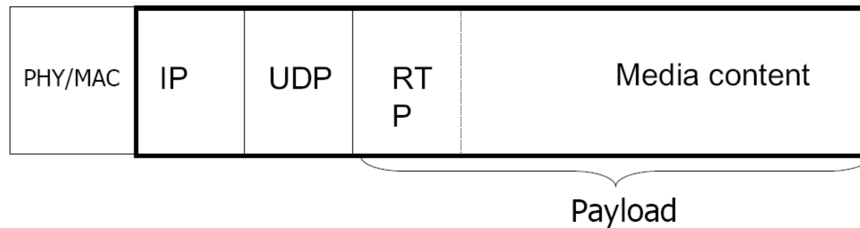
```
v=0
o=- 112938749238  112938749239 IN IP4 192.168.1.1
s=teamVoip
c=IN IP4 192.168.1.1
t=0 0
m=audio 15000 RTP/AVP 0 101
a=rtpmap:0 PCMU/8000
a=sendrecv
a=fmtp:101 0-15
a=ptime:20
```

In the above message, the version is 0, the packet is in its second version denoted by the version being one number greater than the session number, the peer address is 192.168.1.1, session name is teamVoip, it is an audio session with the RTP port being 15000, the sound encoding is PCM at 8000 hertz, the user can send and receive data, there samples are 16 bits, and the length of each packet is 20 ms.

## Real Time Transport Protocol (RTP)

RTP is a real-time end-to-end transport protocol. RTP is best viewed as a framework that applications can use to implement a new single multi-media protocol. RTP doesn't guarantee timely delivery of packets, nor does it keep the packets in sequence. RTP gives the responsibility for recovering lost segments and re-sequencing packets to the application layer. RTP provides the features - Payload type identification, Source identification, Sequence numbering and Time stamping, which are required by most multimedia applications.

Following illustrates the placement of this protocol relative to the Ethernet packet header:




RTP header illustrated below provides information for:

- media content type
- talk spurts
- sender identification
- synchronization
- loss detection
- segmentation and reassembly
- security (encryption)

The RTP packet format (Table 1) is in detail reviewed in the table below:

| V | P | X | CC | M | PT | Sequence Number |
|---|---|---|---|---|---|---|
| Timestamp | | | | | | |
| Synchronization source (SSRC) identifier | | | | | | |
| Contributing source (SSRC_1) identifier | | | | | | |
| … | | | | | | |
| Contributing source (SSRC_1) identifier | | | | | | |
| P A Y L O A D | | | | | | |

## Application Layer Pitfalls

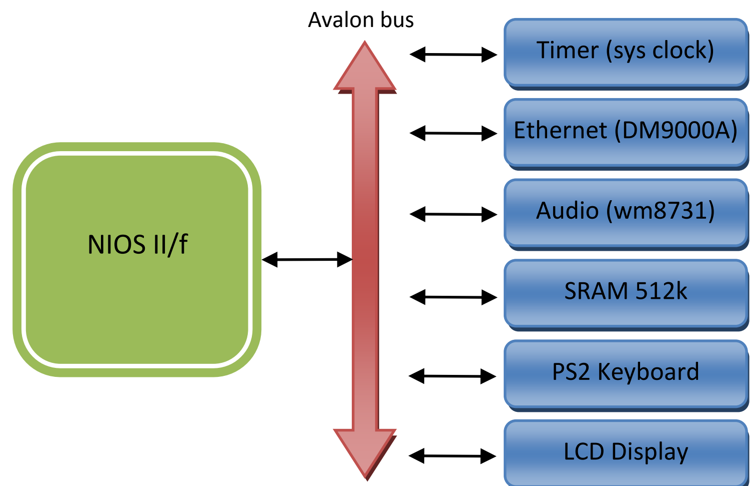
One of the biggest pitfalls of the project is that we did not immediately start examining the sip protocol, rather favoring to use an existing implementation. This was a big mistake because the application chosen was extremely difficult to port, and the effort was abandoned after several weeks of effort.

Another pitfall we ran into was that because the SIP RFC does not make it immediately obvious that SDP is needed to create a media session, we did not find out about its necessity until we were well into the project. However, once development started on a SIP library from scratch, progress moved slowly and steadily until the end.

# Part II: System Software Design

## Overview

The primary purpose of the system software (and hardware) is to provide a comfortable abstraction to the VOIP application layer, so that it may focus on the application logic and communications protocols.  This section describes the design and implementation of the system software layer and its various elements, and the considerations that went into the design process.  For reference, a high level block diagram of the system looks as shown at right.




## CPU

We use the fastest available Nios II processor, with no MMU or MPU, and a small data cache.  The OS we plan to use, µC/OS-II, does not support paging anyway, so an MMU would be a waste.  This CPU fit easily on the Cyclone II FPGA along with additional hardware peripherals and controllers.  We used less than 20% of the LEs on the FPGA.

## Memory

We use the 512k SRAM chip on the DE2 board for both code and data.  Note that some peripherals have additional buffer memory, and the Nios II on the FPGA includes cache memory, and was outfitted with a small amount of TCM (tightly coupled memory) for performance sensitive code and data.

## OS

We use the µC/OS-II operating system provided by the Altera Nios II board support package (BSP) builder.  This provides multi-threading to the TCP/IP stack, system software, as well as the VOIP application.  Threads are needed for the network stack, and allow the system software to keep ISRs very short (offloading their main processing to system threads).  This provides good (read: small) interrupt latency, which keeps everything running smoothly.

## TCP/IP Stack

We use the INiche stack, also provided by Altera via the BSP builder.  Note that this package requires the µC/OS-II operating system.  This stack provides a standard sockets (sometimes called BSD sockets) abstraction to the application for network communication.  This greatly simplified out protocol development.

## CRT

The C runtime library is provided by the Altera BSP builder via their newlib library, and provides the C environment for the system software and application layer software.

## Peripheral Drivers

Peripheral drivers for most peripherals are trivial or already provided. A keyboard driver was given in lab2, and was perfectly suitable. The SRAM needs a simple controller (provided by Quartus SOPC builder), but no driver software. The timer is very simple (just set the rate, hook the interrupt, and increment an integer). The LCD comes with a built-in driver from Altera that works via a file descriptor style interface ('open("/dev/lcd")' and 'write(fdLcd, "hello")').

The audio driver is fairly simple, but not trivial. The hardware provides an interrupt, and some memory mapped registers to control it. The audio peripheral allows us to start/stop sampling, enable and disable interrupts, and so on via a status register. Audio data is read and written to/from the peripheral via additional memory mapped regions. When the peripheral FIFO buffer is half full, the CPU is interrupted. The interrupt handler reads 512 samples out of the peripheral, and queues them to the main thread for sending via RTP.

## Ethernet Driver

When we began the project, we were a bit concerned with the Ethernet hardware on the Altera DE2 (a Davicom chip) because it turns out Altera doesn't provide a proper driver for it. Some basic packet read/write routines were provided in lab 2, but these alone were not robust enough for our application. Using the lab2 code, the documentation for the Davicom chip, an ethernet driver implementation provided for another chip (smsc), and section 7-14 of the Nios II SW developer's guide, we wrote an Altera HAL driver for the DM9000A on the DE2. The driver is provided in the attached file dm9000a.c. As a HAL driver, it is added automatically to the BSP when our ethernet hardware component (dm9000a.vhd/.tcl) is added to a Nios system in SOPC builder, and it is automatically initialized before main is called.

While developing the driver, there were 2 primary issues that we worked through. First, when an interrupt arrives signaling that a packet is ready to be read out, we found that it is important to read out **all** packets that have been received: there may be more than one. The second, much more minor performance issue we found is that the read write delay to the chip is only required to be 200 ns. We generated this delay (actually we used 240 nsecs) using inline assembly "nops", and this worked fine. The original terasic code from lab 2 had used a large 40 usec delay, which actually significantly reduces the throughput of the interface (to less than 500 kb/sec).

## Using Nios II EDS Command Line Tools for Project Development

There are two distinct paths for developing software for the nios2 processor. Nominally, the default method seems to be use the Nios II IDE. In this approach you create your projects with the IDE's wizards using a nice point and click interface. You start by creating what the IDE calls a "system library" project. This contains a C library, and other options that you can add such as the UCOS/II operating system or the

iniche TCP/IP stack[2]. The plan is then that you can build a number of applications against this system library, using an app wizard to generate those as well. You then write code, debug, and repeat as necessary, all from the IDE, until the project is complete. Unfortunately, this method did not work for us ... the Eclipse wizard died with a Java exception.

The other way to work, as we soon discovered, is to use Altera command line tools to generate your system library and applications, and interact with the DE2. The Altera documentation refers to this collection of command line tools as the "software build tools". The use of these tools is discussed in detail in the Nios II Software developer's handbook (Section 1, Chapter 3). This set of tools also comes with a slightly different parlance, using the name "BSP" for what the Nios II IDE calls a "system library". In the model here you generate a BSP with a command line tool called nios2-bsp. In particular, the bsp for our project can be generated with the following invocation:

```
nios2-bsp ucosii ../hw enable_sw_package altera_iniche –set altera_iniche.iniche_default_if eth0
```

where '../hw' is the path to your Quartus hardware project that contains your Nios processor. This creates a BSP that includes uCOS/II, iniche, plus a C library and any drivers it can find for hardware peripherals included in your SOPC builder project. The end product of this is a big pile of code for the options you've selected, a standard makefile to build it all, and a linker description file that tells the linker where (i.e., in memory) to put code, data and heap for applications built for this system. To create an application with this BSP, you run a second tool called nios2-app-generate-makefile as follows:

```
nios2-app-generate-makefile --bsp-dir ../bsp --elf-name myapp.elf --set OBJDUMP_INCLUDE_SOURCE 1 --src-files main.c
```

where '../bsp' is the directory containing the BSP generated in the previous step. This creates a makefile that has the right include directories for the BSP, and adds a new source file containing a main() routine. The Nios2 SDK comes with a complete gnu toolchain including gcc, ar, and so on. So users compile code in this model by just typing make in the BSP or application directory. To download the compiled application, a tool called nios2-download is used, along with nios2-terminal to capture the device's stdout. These 2 would form a nice basis for an automated testing framework. For a pretty graphical debugging environment, users can import the makefile into the Nios II IDE, and set breakpoints, step through the program, examine variables, etc. This can make quick work of many small issues.

Overall, the software build tool approach was straight-forward, and worked very well for our project. One final note, to work with these tools, users must put some additional directories on their path. On the machines in the embedded systems lab these are:

/opt/e4840/altera7.2/nios2eds/sdk2/bin
/opt/e4840/altera7.2/nios2eds/bin
/opt/e4840/altera7.2/nios2eds/bin/nios2-gnutools/H-i686-pc-linux-gnu/bin

# Part III: System Hardware Design

## The LCD Screen

The on-board LCD screen is the display device showing call status and connection information. The interface to the LCD screen is either 4 bit or 8 bit parallel with 4 status pins. On the DE2 board, it is set

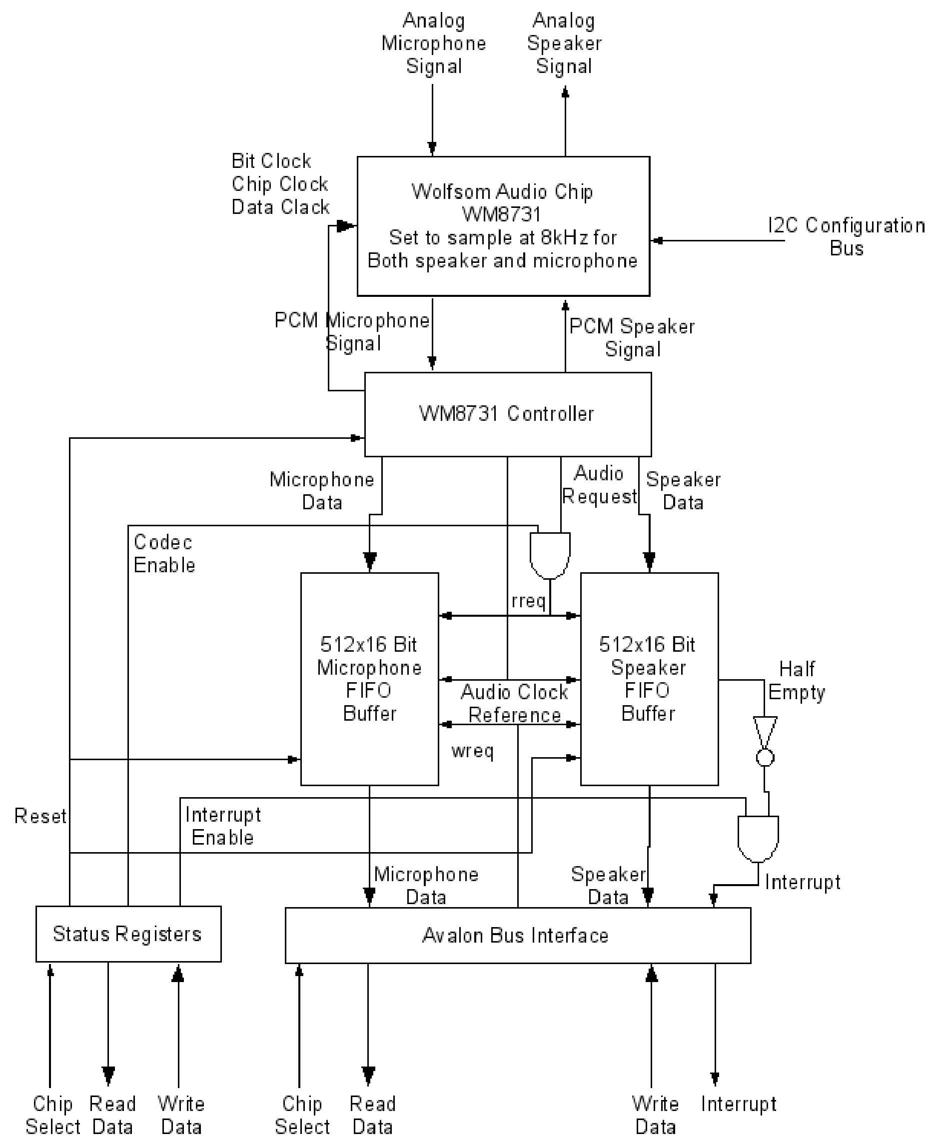
2 But it turns out that if you add both of these, the wizard will crash with some silly Java exception before it generates the code for your project.

up in the 8 bit configuration. SOPC builder contains a stock hardware block for controlling the LCD screen, as well as a stock driver that works with the provided controller.

## The Audio Stack

A diagram of the audio hardware is shown in the diagram below. The audio stack is divided into 6 components. These are the Wolfson audio chip, the Wolfson audio controller, two FIFO buffers, an Avalon bus interface to audio data, and an Avalon bus interface to internal audio stack status registers. All of these components are written in VHDL with the exception of the Wolfson audio chip which is an ASIC processor that is shipped with the Altera DE2 development board.

In the configuration used for this project, the Wolfson chip samples data at 8KHz on both the microphone and speaker input. The way that the audio stack works is that the Wolfson chip samples audio data and encodes it with a DAC for input to the speaker and an ADC for input from the microphone. The Wolfson chip takes 4 clock inputs, a chip clock, a bit clock, an ADC clock, and a DAC clock. The chip clock is 18 MHz, but for this project, we feed it with 12.5 MHz. The bit clock coincides with the synchronization of the PCM audio bits streaming in and out of the chip. Lastly, the ADC and DAC clocks run at 8 KHz and each clock high makes the audio chip take a new sample of audio data. These clocks along with the input and output PCM signals are handled through the Wolfson audio controller.




The Wolfson audio controller takes the 12.5MHz chip clock and divides it down to both the bit and ADC/DAC clocks. Then it shifts out speaker data to form a PCM output for the speaker input for the

audio controller. Likewise, the controller takes PCM microphone data input from the audio chip and shifts it in. The controller does both of these steps in parallel. When a whole sample has been shifted out, the controller sends an audio request to the FIFO buffers in order to retrieve the next sample.

The two FIFO buffers are made from the megafunction wizard included in the Altera Quartus IDE. These functions are built in the same fashion, with the exception that the speaker buffer has a status output that shows the number of samples currently stored in the FIFO that is used for a processor interrupt. These FIFO buffers take two different clocks for the output signal and for the input signal. This is because the audio stack works at a much lower frequency then the board, so a separate clock needs to be provided in order to ensure that the audio gets only one sample per input request.

The Avalon bus interface is the standard one used throughout the project. It provides the signals readdata, writedata, read, write, chipselect, reset, clock, and irq. Also, the status register interface provides all of these signals with the exception of the interrupt. All of the data presented from the readdata and writedata lines are 16 bit samples of audio data. The registers in the status bytes are divided up to signal needed to be set by the processor. The signals exposed are codec_en, irq_en, test_mode, and buffer half full. The codec_en signal enables audio samples to reach the Wolfson audio chip. When this is set to low, all output samples are suspended which effectively stops sound output. The irq_en masks the interrupt output, so that interrupt signals can be turned off. This is primarily used in the interrupt service routine of the application for the receiving and sending of sound data to the chip. The test mode signal turns on the internal test mode of the audio controller. This just reads out a preset list of values to the audio controller. This was primarily used for debugging purposes so that the correct operation of the lower level of the audio stack could be ensured. Likewise, the buffer half full status line was used for debugging purposes in order to see when the interrupt was going off and if it needed to be triggered.

## Hardware pitfalls

The main pitfall regarding the sound stack is that because the audio chip and the audio chip controller run at different clocks, synchronizing the results so that only one sample per *audio_request* high was given to the controller was extremely difficult. In our first revision of working code, there were no checks on this, and it is quite likely that when the *audio_request* signal was given, many audio samples were flooded, resulting in the last one being captured. This means that the audio stack interrupted more often then it should, as well as the intended pitch of the data being read out was much lower than the actual pitch that was heard. This was finally fixed by adding FIFO buffers that were synchronized to two different clocks. The audio chip clock was given for the output and input to the controller, and the board clock was given for the output and input to the NIOS read and write requests.

# Contributions

***Sarfraz Nawaz***

- Master of tools (Asterisk and PJSIP)
- Protocol support, investigation of PJSIP and RTP library for use in system
- Elements of design and final documents, and final presentation

***Mark Niebur***

- Audio stack hardware
- Complete, from scratch SIP/SDP implementation
- Elements of design and final documentation

***Scott Schuff***

- System Software (BSP generation, OS & TCP/IP stack integration, ethernet driver)
- Application Framework (data flow elements (queues, etc), main state machine)
- Elements of design and final documents, and final presentation
- Primary write-up of: design, presentation, and final documents.

***Shiraz Siddiqui***

- Investigation into the use of PJSIP
- Supporting documentation

# Conclusion

We have presented here the layered design of our VOIP system: Application Layer, System Layer, and Hardware Layer. Overall, the final implementation is very close to the original system design, with the only major difference being the audio hardware.

# Lessons Learned

We learned the following lessons:

- To avoid assuming that third party components will be shrunk down to size in order to work on an embedded system (even if the third party software claims to be portable, and have a small footprint)

- SDP played and we discovered that three weeks before the deadline. We were able to implement SDP but it would have been better if we had known of this earlier.

- TCP / IP works nicely with small embedded systems. We had a fairly complete sockets implementation along with an OS and an application in less than 250k of code.

- Test your audio thoroughly. There are things like analog bypass (which copy mic directly to speaker), and single tones can be generated by bugs in the hardware. Check with multiple tones. Then re-check.

# References


1. Nios II Software Developer's Handbook (http://www.altera.com/literature/hb/nios2/n2sw_nii5v2.pdf)
2. NicheStack IPv4 Datasheet (http://www.iniche.com/pdf/nichestackipv4_ds.pdf)
3. µC/OS-II (http://www.micrium.com/products/rtos/kernel/benefits.html)

4. RFC 3261 (SIP: Session Initiation Protocol) (http://www.ietf.org/html/rfc3261#page8)
5. RFC 3261 (SIP: Session Initiation Protocol) (http://www.ietf.org/html/rfc3261#section-18)
6. RFC 1889 (RTP) (http://tools.ietf.org/html/rfc1889#page-3)
7. HD44780U LCD Display Datasheet (http://www.sparkfun.com/datasheets/LCD/HD44780.pdf)
8. LCD interface timing diagram (http://home.iae.nl/users/pouweha/lcd/lcd0.shtml#_8bit-transfer)
9. LCD interface commands (http://www.geocities.com/dinceraydin/lcd/commands.htm)
10. Asterisk project (http://www.asterisk.org/)

# Appendix A: Hardware VDHL code

## Aud_stack.vhd

```
library ieee;

use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity aud_stack is
  port(
    --Avalon Bus Connections For Stack--
    avs_snd_clk         : in std_logic;
    avs_snd_reset_n     : in std_logic;
    avs_snd_read        : in std_logic;
    avs_snd_write       : in std_logic;
    avs_snd_chipselect  : in std_logic;
    avs_snd_readdata    : out std_logic_vector(15 downto 0);
    avs_snd_writedata   : in std_logic_vector(15 downto 0);
    avs_snd_irq         : out std_logic;

    --Avalon Bus Connections For Stat Control--
    avs_snd_stat_read        : in std_logic;
    avs_snd_stat_write       : in std_logic;
    avs_snd_stat_chipselect  : in std_logic;
    avs_snd_stat_readdata    : out std_logic_vector(15 downto 0);
    avs_snd_stat_writedata   : in std_logic_vector(15 downto 0);

    --exported signals--
    audio_request : in std_logic;      --    Audio controller request new data
    mic_dat : in std_logic_vector(15 downto 0);
    speak_dat : out std_logic_vector(15 downto 0)
  );
end aud_stack;

architecture rtl of aud_stack is
  component fifo IS
        PORT
        (
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
                rdreq           : IN STD_LOGIC ;
                sclr            : IN STD_LOGIC ;
                wrreq           : IN STD_LOGIC ;
                usedw           : OUT STD_LOGIC_VECTOR (8 downto 0) ;
                q               : OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
        );
  END component;
  --signals to connect to the audio buffer--
```

```
  signal proc_we    : std_logic;
  signal proc_rd    : std_logic;
  signal codec_we   : std_logic;
  signal codec_rd   : std_logic;
  signal half_full  : std_logic_vector(8 downto 0);
  signal half_full2  : std_logic_vector(8 downto 0);
  signal proc_dat_in: std_logic_vector(15 downto 0);
  signal proc_dat_out: std_logic_vector(15 downto 0);
  --signal codec_dat_in: std_logic_vector(15 downto 0);
  --signal codec_dat_out: std_logic_vector(15 downto 0);
  signal clock_div : unsigned(1 downto 0);

  --register for the audio stack status
  --stat_reg is: en intrs    | en codec    | not used    | not used
  --             not used    | not used    | not used    | not used
  --             not used    | not used    | not used    | not used
  --             not used    | not used    | not used    | not used
  signal stat_reg: std_logic_vector(15 downto 0) := x"0000";

begin
  mic_fifo: fifo port map(
    clock            => clock_div(1),
    data             => mic_dat,
    rdreq            => proc_rd,
    sclr             => '0',
    wrreq            => codec_we,
    usedw            => half_full,
    q                => proc_dat_out
  );
  speak_fifo: fifo port map(
    clock            => clock_div(1),
    data             => proc_dat_in,
    rdreq            => codec_rd,
    sclr             => '0',
    wrreq            => proc_we,
    usedw            => half_full2,
    q                => speak_dat
  );
  avs_snd_irq <= not half_full2(8) and stat_reg(15);
  --codec_dat_in <= speak_dat;
  --mic_dat <= codec_dat_out;
  --codec_dat_in <= mic_dat;
  --speak_dat <= codec_dat_out;
  process(avs_snd_clk)
  begin
    if rising_edge(avs_snd_clk) then
      clock_div <= clock_div + 1;
      codec_we <= '0';
      codec_rd <= '0';
      proc_we <= '0';
      proc_rd <= '0';
      if avs_snd_reset_n = '0' then
        --zero out everything--
        proc_dat_in <= x"0000";
        stat_reg <= x"0000";
      else
        if avs_snd_chipselect = '1' then
          if avs_snd_read = '1' then
              avs_snd_readdata <= proc_dat_out;
              proc_rd <= '1';
          elsif avs_snd_write = '1' then
              proc_dat_in <= avs_snd_writedata;
```

```
              proc_we <= '1';
          end if;
        elsif avs_snd_stat_chipselect = '1' then
          if avs_snd_stat_read = '1' then
            avs_snd_stat_readdata <= stat_reg;
          elsif avs_snd_stat_write = '1' then
            stat_reg <= avs_snd_stat_writedata;
          end if;
        else
          if audio_request = '1' and stat_reg(14) = '1' then
            codec_we <= '1';
            codec_rd <= '1';
          end if;
        end if;
      end if;
    end if;
  end process;
end architecture;
```

dm9000a.vhd:

```
library ieee;
use ieee.std_logic_1164.all;


entity dm9000a is
  port(
       signal iCMD,iRD_N,iWR_N,
              iCS_N,iRST_N: in std_logic;

        signal iDATA: in std_logic_vector(15 downto 0);
        signal oDATA: out std_logic_vector(15 downto 0);
        signal oINT: out std_logic;

        --  DM9000A Side
        signal ENET_DATA: inout std_logic_vector(15 downto 0);

        signal ENET_CMD,
               ENET_RD_N,ENET_WR_N,
               ENET_CS_N,ENET_RST_N: out std_logic;

        signal ENET_INT: in std_logic
       );
end dm9000a;

architecture behavior of dm9000a is
begin

       ENET_DATA  <=  iDATA when iWR_N='0' else (others => 'Z');
       oDATA      <=  ENET_DATA;

       ENET_CMD   <=  iCMD;
       ENET_RD_N  <=  iRD_N;
       ENET_WR_N       <=    iWR_N;
       ENET_CS_N       <=    iCS_N;
       ENET_RST_N <=  iRST_N;
       oINT       <=  ENET_INT;

end behavior;
```

# Appendix B: Source code listing

## cv-voip.h

```
#ifndef __voip_h_5f9ac83e_da06_4dca_96b0_0832d06221f3
#define __voip_h_5f9ac83e_da06_4dca_96b0_0832d06221f3

#include "cv-queue.h"
#include "cv-bpool.h"
#include "cv-sound.h"
#include "cv-rtp.h"
#include "cv-lcd.h"
#include "cv-kbd.h"
#include "cv-mbox.h"
#include "cv-sip.h"

typedef enum cv_voipState
{

  eInitialized,
  eReady,
  eDialing,
  eCallSent,
  eRinging,
  eBusy,
  eRemoteRinging,
  eInCall,
  eUnknown
} cv_voipState;


typedef struct cv_voip
{
  cv_sound     snd;
  cv_buffpool  bpool;
  cv_queue     outq;
  cv_queue     inq;
  cv_rtp       rtp;
  cv_lcd       lcd;
  cv_kbd       kbd;
  cv_mbox      mbx;
  cv_sip       sip;

  cv_voipState state;

  SOCKET       sockMain;
  cv_mbox*     smbx;

  uint32       maxDat;
  uint32       mcount;
} cv_voip;

cv_status cv_voip_construct(cv_voip* pv);
cv_status cv_voip_run(cv_voip* pv);
cv_status cv_voip_destruct(cv_voip* pv);



#endif /* __voip_h_5f9ac83e_da06_4dca_96b0_0832d06221f3 */
```

## cv-voip.c

```
#include <stdlib.h>
#include <stdio.h>

#include "includes.h"
#include <alt_iniche_dev.h>
#include <ipport.h>
```

```
#include <osport.h>
#include <tcpport.h>
#include "cv-voip.h"
#include "cv-msg.h"



#if 0
// tone, 440 Hz, 256 samples
const unsigned short a_buff[256] = {
  0, 1413, 2816, 4199, 5549, 6859, 8117, 9315, 10443, 11493, 12458, 13330,
14102, 14769, 15326, 15768,
  16093, 16298, 16381, 16343, 16182, 15900, 15500, 14984, 14357, 13622,
12786, 11854, 10834, 9734, 8560,
  7323, 6031, 4694, 3322, 1925, 514, -900, -2308, -3699, -5062, -6388, -7666,
-8887, -10041, -11121,
  -12118, -13024, -13833, -14539, -15136, -15621, -15989, -16238, -16365, -
16371, -16254, -16016, -15659, -15185, -14598,
  -13901, -13102, -12204, -11215, -10143, -8995, -7780, -6506, -5185, -3824,
-2435, -1028, 386, 1797, 3196,
  4570, 5911, 7207, 8450, 9630, 10738, 11765, 12705, 13550, 14294, 14932,
15458, 15869, 16161, 16333,
  16383, 16311, 16117, 15803, 15371, 14824, 14167, 13404, 12541, 11585,
10542, 9420, 8229, 6975, 5670,
  4323, 2943, 1541, 128, -1285, -2690, -4074, -5428, -6742, -8005, -9209, -
10344, -11401, -12374, -13254,
  -14036, -14713, -15280, -15733, -16069, -16285, -16379, -16351, -16201, -
15931, -15541, -15036, -14418, -13693, -12866,
  -11943, -10931, -9837, -8670, -7438, -6150, -4817, -3448, -2053, -643, 771,
2181, 3574, 4940, 6269,
  7552, 8778, 9939, 11026, 12031, 12945, 13764, 14479, 15087, 15582, 15960,
16220, 16359, 16375, 16270,
  16043, 15697, 15233, 14656, 13969, 13178, 12289, 11309, 10243, 9102, 7893,
6624, 5307, 3949, 2563,
  1157, -257, -1669, -3070, -4447, -5791, -7092, -8340, -9525, -10640, -
11675, -12624, -13478, -14231, -14879,
  -15415, -15836, -16140, -16322, -16384, -16322, -16140, -15836, -15415, -
14879, -14231, -13478, -12624, -11675, -10640,
  -9525, -8340, -7092, -5791, -4447, -3070, -1669, -257, 1157, 2563, 3949,
5307, 6624, 7893, 9102,
  10243, 11309, 12289, 13178, 13969, 14656, 15233, 15697, 16043, 16270,
16375, 16359, 16220, 15960, 15582,
  15087, 14479, 13764, 12945, 12031, 11026, 9939, 8778, 7552, 6269, 4940,
3574, 2181, 771, -643
};

// tone, 220 Hz, 256 samples
unsigned short b_buff[256] = {
  0, 176, 353, 529, 704, 877, 1049, 1219, 1387, 1552, 1714, 1873, 2029, 2181,
2328, 2472,
  2610, 2744, 2873, 2996, 3114, 3226, 3332, 3432, 3525, 3612, 3692, 3765,
3831, 3890, 3942,
  3986, 4023, 4052, 4074, 4088, 4095, 4094, 4085, 4069, 4045, 4014, 3975,
3928, 3875, 3814,
  3746, 3671, 3589, 3500, 3405, 3304, 3196, 3083, 2963, 2838, 2708, 2573,
2433, 2288, 2140,
```

```
  1987, 1830, 1670, 1507, 1341, 1173, 1003, 830, 656, 481, 305, 128, -48, -
225, -401,
  -577, -751, -924, -1096, -1265, -1432, -1597, -1758, -1916, -2071, -2221, -
2368, -2510, -2647, -2780,
  -2907, -3029, -3145, -3256, -3360, -3458, -3549, -3634, -3712, -3784, -
3848, -3905, -3955, -3997, -4032,
  -4059, -4079, -4091, -4095, -4092, -4082, -4063, -4037, -4004, -3963, -
3914, -3859, -3796, -3726, -3649,
  -3565, -3475, -3378, -3275, -3166, -3051, -2930, -2803, -2672, -2535, -
2394, -2248, -2098, -1945, -1787,
  -1626, -1462, -1296, -1127, -956, -783, -608, -433, -257, -80, 96, 273,
449, 624, 799,
  971, 1142, 1311, 1477, 1641, 1801, 1959, 2112, 2262, 2407, 2548, 2684,
2815, 2941, 3061,
  3176, 3285, 3387, 3483, 3573, 3656, 3733, 3802, 3864, 3919, 3967, 4007,
4040, 4065, 4083,
  4093, 4095, 4090, 4077, 4057, 4029, 3993, 3950, 3900, 3842, 3778, 3706,
3627, 3541, 3449,
  3351, 3246, 3135, 3018, 2896, 2768, 2635, 2497, 2355, 2208, 2057, 1902,
1743, 1582, 1417,
  1250, 1080, 909, 735, 561, 385, 209, 32, -144, -321, -497, -672, -846, -
1018, -1189,
  -1357, -1522, -1685, -1845, -2001, -2153, -2302, -2446, -2586, -2720, -
2850, -2974, -3093, -3206, -3313,
  -3414, -3509, -3597, -3678, -3752, -3820, -3880, -3933, -3979, -4017, -
4048, -4071, -4086, -4094, -4095
};

const unsigned short z_buff[256] = {0};

#endif


static cv_status cv_voip_getNextState(cv_voip* pv, cv_voipState* ns);
static cv_status cv_voip_transition(cv_voip* pv, cv_voipState nextState);

/* GetNextState  */
static cv_status cv_voip_initializedGetNextState(cv_voip* pv, cv_voipState*
ns);
static cv_status cv_voip_readyGetNextState(cv_voip* pv, cv_voipState* ns);
static cv_status cv_voip_inCallGetNextState(cv_voip* pv, cv_voipState* ns);
static cv_status cv_voip_ringingGetNextState(cv_voip* pv, cv_voipState* ns);
static cv_status cv_voip_remoteRingingGetNextState(cv_voip* pv, cv_voipState*
ns);
static cv_status cv_voip_dialingGetNextState(cv_voip* pv, cv_voipState* ns);
static cv_status cv_voip_busyGetNextState(cv_voip* pv, cv_voipState* ns);
static cv_status cv_voip_callSentGetNextState(cv_voip* pv, cv_voipState* ns);

/* state transition functions */
static cv_status cv_voip_readyFromInitialized(cv_voip* pv);
static cv_status cv_voip_remoteRingingFromCallSent(cv_voip* pv);
static cv_status cv_voip_readyFromDialing(cv_voip* pv);
static cv_status cv_voip_readyFromBusy(cv_voip* pv);
static cv_status cv_voip_busyFromCallSent(cv_voip* pv);
static cv_status cv_voip_callSentFromDialing(cv_voip* pv);

static cv_status cv_voip_readyFromRinging(cv_voip* pv);
```

```c
static cv_status cv_voip_readyFromRemoteRinging(cv_voip* pv);
static cv_status cv_voip_readyFromInCall(cv_voip* pv);
static cv_status cv_voip_readyFromCallSent(cv_voip* pv);
static cv_status cv_voip_inCallFromRinging(cv_voip* pv);
static cv_status cv_voip_inCallFromRemoteRinging(cv_voip* pv);
static cv_status cv_voip_ringingFromReady(cv_voip* pv);
static cv_status cv_voip_dialingFromReady(cv_voip* pv);

static cv_status cv_voip_monitor(cv_voip* pv, uint8* buffer, uint32 sz);

#define CONTROL 1
#define RECV    2

#define BAD_STATE_TRANSITION(s) { debugBreak(); printf("bad state
transition\n"); }

static const char* SIP_REGISTRAR = "192.168.1.2";
#define SIP_PORT 15908
#define RTP_PORT 12908

/* debugging routines */
void debugBreak()
{
  printf("debug break\n");
  fflush(stdout);
}

#if 0
/* dtrap() - function to trap to debugger */
void dtrap(void)
{
  printf("dtrap - needs breakpoint\n");
  fflush(stdout);
}
#endif

cv_status errHook(cv_status status)
{
  if(status != cv_status_success)
    debugBreak();
  return status;
}

cv_status cv_voip_construct(cv_voip* pv)
{
  cv_status status = cv_status_success;
  int idx;

  pv->maxDat = 0;

  status = cv_buffpool_construct(&pv->bpool, sizeof(cv_queueNode) + 640, 16,
64);
  cv_status_returnIfFailed(status);
  status = cv_queue_construct(&pv->outq);
  cv_status_returnIfFailed(status);
  status = cv_queue_construct(&pv->inq);
  cv_status_returnIfFailed(status);
```

```
  status = cv_sound_construct(&pv->snd, &pv->bpool, &pv->inq, &pv->outq);
  cv_status_returnIfFailed(status);
  status = cv_rtp_construct(&pv->rtp,cv_getlocalIPAddress(),"RTP is fun");
  cv_status_returnIfFailed(status);
  status = cv_lcd_construct(&pv->lcd);
  cv_status_returnIfFailed(status);
  status = cv_kbd_construct(&pv->kbd);
  cv_status_returnIfFailed(status);
  status = cv_mbox_construct(&pv->mbx, 5);
  cv_status_returnIfFailed(status);
  status = cv_sip_construct(&pv->sip, cv_getlocalIPAddress(),
                            SIP_REGISTRAR, SIP_PORT,
                            RTP_PORT, &pv->mbx);
  cv_status_returnIfFailed(status);

  pv->state = eInitialized;
  pv->smbx = &pv->sip.mbox;

  cv_status_return(status);
}

cv_status voip_net(cv_voip* pv)
{
  cv_status status = cv_status_success;
  cv_voipState state;

  while(1) {
    status = cv_voip_getNextState(pv, &state);
    cv_status_returnIfFailed(status);
    status = cv_voip_transition(pv, state);
    cv_status_returnIfFailed(status);
  }

  cv_status_return(status);
}

void voip_net_entry(void* arg)
{
  cv_status status;
  cv_voip* pv = (cv_voip*)arg;
  status = voip_net(pv);
}

cv_status cv_voip_transition(cv_voip* pv, cv_voipState nextState)
{
  cv_status status = cv_status_success;

  switch(pv->state)
  {
    case eInitialized:
      if(nextState == eReady) {
        status = cv_voip_readyFromInitialized(pv);
        cv_status_returnIfFailed(status);
      } else
        BAD_STATE_TRANSITION(status);
      break;
    case eReady:
```

```
    if(nextState == eDialing) {
      status = cv_voip_dialingFromReady(pv);
      cv_status_returnIfFailed(status);
    } else if (nextState == eRinging) {
      status = cv_voip_ringingFromReady(pv);
      cv_status_returnIfFailed(status);
    } else
      BAD_STATE_TRANSITION(status);
    break;
  case eDialing:
    if(nextState == eCallSent) {
      status = cv_voip_callSentFromDialing(pv);
      cv_status_returnIfFailed(status);
    } else if(nextState == eReady) {
      status = cv_voip_readyFromDialing(pv);
      cv_status_returnIfFailed(status);
    }
    break;
  case eCallSent:
    if(nextState == eRemoteRinging) {
      status = cv_voip_remoteRingingFromCallSent(pv);
      cv_status_returnIfFailed(status);
    } else if(nextState == eBusy) {
      status = cv_voip_busyFromCallSent(pv);
      cv_status_returnIfFailed(status);
    } else if(nextState == eReady) {
      status = cv_voip_readyFromCallSent(pv);
      cv_status_returnIfFailed(status);
    } else
      BAD_STATE_TRANSITION(status);
    break;
  case eRinging:
    if(nextState == eInCall) {
      status = cv_voip_inCallFromRinging(pv);
      cv_status_returnIfFailed(status);
    } else if(nextState == eReady) {
      status = cv_voip_readyFromRinging(pv);
      cv_status_returnIfFailed(status);
    } else
      BAD_STATE_TRANSITION(status);
    break;
  case eRemoteRinging:
    if(nextState == eInCall) {
      status = cv_voip_inCallFromRemoteRinging(pv);
      cv_status_returnIfFailed(status);
    } else if (nextState == eReady) {
      status = cv_voip_readyFromRemoteRinging(pv);
      cv_status_returnIfFailed(status);
    } else
      BAD_STATE_TRANSITION(status);
    break;
  case eBusy:
    if(nextState == eReady) {
      status = cv_voip_readyFromBusy(pv);
      cv_status_returnIfFailed(status);
    } else
      BAD_STATE_TRANSITION(status);
```

```
      break;
    case eInCall:
      if(nextState == eReady) {
        status = cv_voip_readyFromInCall(pv);
        cv_status_returnIfFailed(status);
      } else
        BAD_STATE_TRANSITION(status);
      break;
    default:
      break;
  };

  pv->state = nextState;

  cv_status_return(status);
}


cv_status cv_voip_testRTP(cv_voip* pv)
{
  cv_status status = cv_status_success;
  cv_queueNode* pn;
  uint8* buffer = NULL;

  sprintf(pv->rtp.remoteAddr, "192.168.1.2");
  pv->rtp.remotePort = 5060;

  status = cv_rtp_start(&pv->rtp);
  cv_status_returnIfFailed(status);

  status = cv_sound_start(&pv->snd);
  cv_status_returnIfFailed(status);

  while(1)
  {
    status = cv_queue_pop(&pv->outq, &pn);
    cv_status_returnIfFailed(status);
    buffer = (uint8*)(pn+1);

    status = cv_rtp_send(&pv->rtp, buffer, 512);
    cv_status_returnIfFailed(status);

    status = cv_buffpool_free(&pv->bpool, (uint8*)pn);
    cv_status_returnIfFailed(status);
  }

  cv_status_return(status);
}

cv_status cv_voip_getNextState(cv_voip* pv, cv_voipState* ns)
{
  cv_status status = cv_status_success;

  switch(pv->state)
  {
    case eInitialized:
      status = cv_voip_initializedGetNextState(pv, ns);
```

```c
      cv_status_returnIfFailed(status);
      break;
    case eReady:
      status = cv_voip_readyGetNextState(pv, ns);
      cv_status_returnIfFailed(status);
      break;
    case eDialing:
      status = cv_voip_dialingGetNextState(pv, ns);
      cv_status_returnIfFailed(status);
      break;
    case eCallSent:
      status = cv_voip_callSentGetNextState(pv, ns);
      cv_status_returnIfFailed(status);
      break;
    case eRinging:
      status = cv_voip_ringingGetNextState(pv, ns);
      cv_status_returnIfFailed(status);
      break;
    case eBusy:
      status = cv_voip_busyGetNextState(pv, ns);
      cv_status_returnIfFailed(status);
      break;
    case eRemoteRinging:
      status = cv_voip_remoteRingingGetNextState(pv, ns);
      cv_status_returnIfFailed(status);
      break;
    case eInCall:
      status = cv_voip_inCallGetNextState(pv, ns);
      cv_status_returnIfFailed(status);
      break;
    default:
      break;
  }

  /* TODO: testing hack!!!
   *  *ns = pv->state;
   */

  cv_status_return(status);
}

cv_status cv_voip_readyGetNextState(cv_voip* pv, cv_voipState* ns)
{
  cv_status status = cv_status_success;
  cv_voipState nextState = eUnknown;
  int inp;
  cv_msg msg;

  /* in this case we wait only for the sip socket
   * to get an invite, or the local user to initiate
   * an invite via local UI (keyboard) */
  printf("entering ready\n");

  while(nextState == eUnknown)
  {
    status = cv_mbox_read(&pv->mbx, &msg);
    cv_status_returnIfFailed(status);
```

```
    if(msg.cmd == eMsg_none) {
      /* check the keyboard */
      inp = cv_kbd_pollChar(&pv->kbd);
      if( (inp != kbdTimeout) && (isAscii(inp)) ) {
        uint8 ch = mkAscii(inp);
        switch( ch ) {
          case ' ':
            nextState = eDialing;
            break;
          default:
            printf("ignoring key: %c\n", ch);
            break;
        }
      }
    } else if(msg.cmd == eMsg_ringing ||
              msg.cmd == eMsg_invite) {
      printf("invite recvd, ringing\n");
      nextState = eRinging;
    } else {
      printf("ignoring unknown ready msg: %d\n", msg.cmd);
      OSTimeDlyHMSM(0,0,0,1);
    }
  }

  *ns = nextState;

  cv_status_return(status);
}

cv_status cv_voip_start(cv_voip* pv)
{
  cv_status status = cv_status_success;

  voip_net(pv);
  printf("exiting\n");

  cv_status_return(status);
}

cv_status cv_voip_destruct(cv_voip* pv)
{
  cv_status status = cv_status_success;

  status = cv_buffpool_destruct(&pv->bpool);
  cv_status_returnIfFailed(status);
  status = cv_queue_destruct(&pv->outq);
  cv_status_returnIfFailed(status);
  status = cv_queue_destruct(&pv->inq);
  cv_status_returnIfFailed(status);
  status = cv_sound_destruct(&pv->snd);
  cv_status_returnIfFailed(status);
  status = cv_rtp_destruct(&pv->rtp);
  cv_status_returnIfFailed(status);
  status = cv_sip_destruct(&pv->sip);
  cv_status_returnIfFailed(status);
```

```
  cv_status_return(status);
}

cv_status cv_voip_dialingGetNextState(cv_voip* pv, cv_voipState* ns)
{
  cv_status status = cv_status_success;
  cv_voipState nextState = eUnknown;
  cv_msg msg;
  int inp, pos = 0;
  char number[16] = {0};

  status = cv_msg_init(&msg, eMsg_invite, 0);
  cv_status_returnIfFailed(status);

  // in this state, we are waiting for a 'space' key
  // or an invite to arrive via sip
  while(nextState == eUnknown)
  { // check the keyboard
    inp = cv_kbd_pollChar(&pv->kbd);
    if(inp != kbdTimeout) {
      if(isAscii(inp)) {
        uint8 ch = mkAscii(inp);
        switch( ch )
        {
          case '\r':
          case '\n':
            msg.arg = atoi(number);
            status = cv_mbox_write(pv->smbx, &msg);
            cv_status_returnIfFailed(status);
            nextState = eCallSent;
            break;
          case ' ':
            nextState = eReady;
            break;
          default:
            cv_lcd_appendChar(&pv->lcd, ch);
            number[pos++] = ch;
            break;
        }
      } else {
        if( (inp == kbdBS) && (pos > 0) ) {
          number[--pos] = 0;
          status = cv_lcd_backspace(&pv->lcd);
          cv_status_returnIfFailed(status);
        }
      }
    }
  }

  *ns = nextState;

  cv_status_return(status);
}

cv_status cv_voip_callSentGetNextState(cv_voip* pv, cv_voipState* ns)
{
  cv_status status = cv_status_success;
```

```
cv_msg msg;
int inp;
cv_voipState nextState = eUnknown;

status = cv_msg_init(&msg, eMsg_hup, 0);
cv_status_returnIfFailed(status);

while (nextState == eUnknown)
{
  status = cv_mbox_read(&pv->mbx, &msg);
  cv_status_returnIfFailed(status);

  if(msg.cmd == eMsg_ringing) {
    nextState = eRemoteRinging;
    break;
  }
  if(msg.cmd == eMsg_busy) {
    nextState = eBusy;
    break;
  }
  if(msg.cmd == eMsg_decline) {
    status = cv_lcd_clear(&pv->lcd);
    cv_status_returnIfFailed(status);
    status = cv_lcd_print(&pv->lcd, "**** ERROR: \n");
    cv_status_returnIfFailed(status);
    status = cv_lcd_print(&pv->lcd, "call DECLINED");
    cv_status_returnIfFailed(status);
    OSTimeDlyHMSM(0,0,4,0);
    nextState = eReady;
    break;
  }
  if(msg.cmd == eMsg_unavailable) {
    status = cv_lcd_clear(&pv->lcd);
    cv_status_returnIfFailed(status);
    status = cv_lcd_print(&pv->lcd, "**** ERROR: \n");
    cv_status_returnIfFailed(status);
    status = cv_lcd_print(&pv->lcd, "USER UNAVAILABLE");
    cv_status_returnIfFailed(status);
    OSTimeDlyHMSM(0,0,4,0);
    nextState = eReady;
    break;
  }

  inp = cv_kbd_pollChar(&pv->kbd);
  if(inp != kbdTimeout) {
    if(isAscii(inp)) {
      if(mkAscii(inp) == ' ') {
        status = cv_mbox_write(pv->smbx, &msg);
        cv_status_returnIfFailed(status);
        nextState = eReady;
      }
    }
  }
}

*ns = nextState;
```

```
  cv_status_return(status);
}

cv_status cv_voip_ringingGetNextState(cv_voip* pv, cv_voipState* ns)
{
  cv_status status = cv_status_success;
  cv_msg ans, msg;
  cv_voipState nextState = eUnknown;

  status = cv_msg_init(&ans, eMsg_answer, 0);
  cv_status_returnIfFailed(status);

  /* if the local user answers, the next state is inCall, otherwise
   * we go back to ready */

  while(nextState == eUnknown)
  {
    int res = cv_kbd_pollChar(&pv->kbd);
    if(isAscii(res))
      res = mkAscii(res);
    if(res == ' ') {
      status = cv_mbox_write(pv->smbx, &ans);
      cv_status_returnIfFailed(status);
      nextState = eInCall;
      break;
    }

    status = cv_mbox_read(&pv->mbx, &msg);
    cv_status_returnIfFailed(status);

    if(msg.cmd == eMsg_hup)
      nextState = eReady;
  }
  *ns = nextState;

  cv_status_return(status);
}

cv_status cv_voip_remoteRingingGetNextState(cv_voip* pv, cv_voipState* ns)
{
  cv_status status = cv_status_success;
  cv_msg hup, msg;
  cv_voipState nextState = eUnknown;

  status = cv_msg_init(&hup, eMsg_hup, 0);
  cv_status_returnIfFailed(status);

  while(nextState == eUnknown)
  {
    status = cv_mbox_read(&pv->mbx, &msg);
    cv_status_returnIfFailed(status);

    if(msg.cmd == eMsg_none) {
      int res = cv_kbd_pollChar(&pv->kbd);
      if(isAscii(res)) {
        if(mkAscii(res) == ' ') {
          status = cv_mbox_write(pv->smbx, &hup);
```

```
          cv_status_returnIfFailed(status);
          nextState = eReady;
          break;
        }
      }
    } else if (msg.cmd == eMsg_answer) {
      nextState = eInCall;
    }
  }

  *ns = nextState;

  cv_status_return(status);
}

cv_status cv_voip_initializedGetNextState(cv_voip* pv, cv_voipState* ns)
{
  cv_status status = cv_status_success;
  char number[16] = {0};
  int inp, pos = 0;
  cv_msg num;
  cv_voipState nextState = eUnknown;

  status = cv_msg_init(&num, eMsg_number, 0);
  cv_status_returnIfFailed(status);

  status = cv_lcd_clear(&pv->lcd);
  cv_status_returnIfFailed(status);
  status = cv_lcd_print(&pv->lcd, "enter extention:");
  cv_status_returnIfFailed(status);
  status = cv_lcd_line2(&pv->lcd);
  cv_status_returnIfFailed(status);

  while(nextState == eUnknown)
  {
    inp = cv_kbd_pollChar(&pv->kbd);
    if(inp != kbdTimeout) {
      if(isAscii(inp)) {
        char ch = mkAscii(inp);
        switch(ch) {
          case '\r':
          case '\n':
            num.arg = atoi(number);
            status = cv_mbox_write(pv->smbx, &num);
            cv_status_returnIfFailed(status);
            nextState = eReady;
            break;
          default:
            if(pos < 16) {
              number[pos++] = ch;
              status = cv_lcd_appendChar(&pv->lcd, ch);
              cv_status_returnIfFailed(status);
            } else
              printf("ignoring character, > 16\n");
            break;
        }
      } else {
```

```
        if( (inp == kbdBS) && (pos > 0) ) {
          number[--pos] = '0';
          status = cv_lcd_backspace(&pv->lcd);
          cv_status_returnIfFailed(status);
        }
      }
    }
  }

  *ns = nextState;
  cv_status_return(status);
}

cv_status cv_voip_busyGetNextState(cv_voip* pv, cv_voipState* ns)
{
  cv_status status = cv_status_success;
  cv_msg hup;
  cv_voipState nextState = eUnknown;

  status = cv_msg_init(&hup, eMsg_hup, 0);
  cv_status_returnIfFailed(status);

  while(nextState == eUnknown)
  {
    int res = cv_kbd_pollChar(&pv->kbd);
    if(isAscii(res))
      res = mkAscii(res);
    if(res == ' ') {
      status = cv_mbox_write(pv->smbx, &hup);
      cv_status_returnIfFailed(status);
      nextState = eReady;
    }
  }

  *ns = nextState;

  cv_status_return(status);
}

cv_status cv_voip_monitor(cv_voip* pv, uint8* buffer, uint32 sz)
{
  cv_status status = cv_status_success;

  int idx = 0;
  for(idx=0; idx < sz; ++idx)
  {
    if(buffer[idx] > pv->maxDat) {
      pv->maxDat = buffer[idx];
    }
  }

  if(++pv->mcount == 10)
  {
    char buf[16] = {0};
    sprintf(buf, "%x", pv->maxDat);

    status = cv_lcd_clear(&pv->lcd);
```

```
    cv_status_returnIfFailed(status);
    status = cv_lcd_print(&pv->lcd, buf);
    pv->maxDat = 0;
    pv->mcount = 0;
  }

  cv_status_return(status);
}

cv_status cv_voip_inCallGetNextState(cv_voip* pv, cv_voipState* ns)
{
  cv_status status = cv_status_success;
  cv_voipState nextState = eUnknown;
  cv_queueNode* pn;
  uint8* buffer = NULL;
  cv_msg hup, msg;
  uint32 len;

  status = cv_msg_init(&hup, eMsg_hup, 0);
  cv_status_returnIfFailed(status);

  status = cv_sound_start(&pv->snd);
  cv_status_returnIfFailed(status);

  while(nextState == eUnknown)
  {
    /* check if local user hung up */
    int res = cv_kbd_pollChar(&pv->kbd);
    if(isAscii(res))
      res = mkAscii(res);
    if(res == ' ') {
      status = cv_mbox_write(pv->smbx, &hup);
      cv_status_returnIfFailed(status);
      nextState = eReady;
      break;
    }

    /* check if remote user hung up */
    status = cv_mbox_read(&pv->mbx, &msg);
    cv_status_returnIfFailed(status);
    if(msg.cmd == eMsg_hup) {
      nextState = eReady;
      break;
    }

    /* check for incoming voice data on the rtp socket */
    if(cv_rtp_waitReadable(&pv->rtp, 1)) {
      // grab a buffer from pool
      status = cv_buffpool_alloc(&pv->bpool, (uint8**)&pn);
      cv_status_returnIfFailed(status);
      buffer = (uint8*)(pn+1);
      // recv it into buffer
      status = cv_rtp_recv(&pv->rtp, buffer, 512, &len);
      cv_status_returnIfFailed(status);
      // queue to sound hardware
      status = cv_queue_push(&pv->inq, pn);
      cv_status_returnIfFailed(status);
```

```
    }

    /* shovel voice data from mic into rtp socket */
    status = cv_queue_pop(&pv->outq, &pn);
    cv_status_returnIfFailed(status);
    buffer = (uint8*)(pn+1);

    status = cv_rtp_send(&pv->rtp, buffer, 512);
    cv_status_returnIfFailed(status);

    status = cv_buffpool_free(&pv->bpool, (uint8*)pn);
    cv_status_returnIfFailed(status);
   }

  status = cv_sound_stop(&pv->snd);
  cv_status_returnIfFailed(status);

  *ns = nextState;

  cv_status_return(status);
}

/*
 ****   State transition functions
 */
cv_status cv_voip_dialingFromReady(cv_voip* pv)
{
  cv_status status = cv_status_success;

  printf("cv_voip_dialingFromReady\n");

  status = cv_lcd_clear(&pv->lcd);
  cv_status_returnIfFailed(status);
  status = cv_lcd_print(&pv->lcd, "*** dialing: ");
  cv_status_returnIfFailed(status);
  status = cv_lcd_line2(&pv->lcd);
  cv_status_returnIfFailed(status);

#if 0
  /* debugging stuff  */
  cv_sound_playTone(&pv->snd, z_buff, 256);
  cv_sound_start(&pv->snd);

  while(1)
  {
    cv_sound_playTone(&pv->snd, a_buff, 256);
    usleep(1000*1000);
    cv_sound_playTone(&pv->snd, z_buff, 256);
    usleep(1000*1000);
    cv_sound_playTone(&pv->snd, b_buff, 256);
    usleep(1000*1000);
    cv_sound_playTone(&pv->snd, z_buff, 256);
    usleep(1000*1000);
  }
#endif

  // TODO: print the number here
```

```
  cv_status_return(status);
}

cv_status cv_voip_ringingFromReady(cv_voip* pv)
{
  cv_status status = cv_status_success;

  printf("cv_voip_ringingFromReady\n");

  status = cv_lcd_clear(&pv->lcd);
  cv_status_returnIfFailed(status);
  status = cv_lcd_print(&pv->lcd, "*** incoming ***");
  cv_status_returnIfFailed(status);
  status = cv_lcd_line2(&pv->lcd);
  cv_status_returnIfFailed(status);
  status = cv_lcd_print(&pv->lcd, "1234567");
  cv_status_returnIfFailed(status);

  // TODO:

  // 1) write invite info out to LCD
  //
  // 2) start a ringing sound on speaker
  //

  cv_status_return(status);
}

cv_status cv_voip_inCallFromRemoteRinging(cv_voip* pv)
{
  cv_status status = cv_status_success;

  sprintf(pv->rtp.remoteAddr, pv->sip.incomingSDP.addr);
  pv->rtp.remotePort = pv->sip.incomingSDP.port;

  printf("starting RTP for remote caller: %s:%d\n", pv->rtp.remoteAddr,
        pv->rtp.remotePort);

  /* start rtp session */
  status = cv_rtp_start(&pv->rtp);
  cv_status_returnIfFailed(status);

  status = cv_lcd_clear(&pv->lcd);
  cv_status_returnIfFailed(status);
  status = cv_lcd_print(&pv->lcd, "*** in call ***");
  cv_status_returnIfFailed(status);

  cv_status_return(status);
}
cv_status cv_voip_inCallFromRinging(cv_voip* pv)
{
  cv_status status = cv_status_success;

  sprintf(pv->rtp.remoteAddr, pv->sip.incomingSDP.addr);
  pv->rtp.remotePort = pv->sip.incomingSDP.port;
```

```
  /* start rtp session */
  status = cv_rtp_start(&pv->rtp);
  cv_status_returnIfFailed(status);

  status = cv_lcd_clear(&pv->lcd);
  cv_status_returnIfFailed(status);
  status = cv_lcd_print(&pv->lcd, "*** in call ***");
  cv_status_returnIfFailed(status);

  cv_status_return(status);
}
static cv_status printReady(cv_lcd* lcd)
{
  cv_status status = cv_status_success;

  status = cv_lcd_clear(lcd);
  cv_status_returnIfFailed(status);
  status = cv_lcd_print(lcd, "**** ready ****");
  cv_status_returnIfFailed(status);

  cv_status_return(status);
}

cv_status cv_voip_readyFromRemoteRinging(cv_voip* pv)
{
  return printReady(&pv->lcd);
}
cv_status cv_voip_readyFromRinging(cv_voip* pv)
{
  return printReady(&pv->lcd);
}
cv_status cv_voip_readyFromBusy(cv_voip* pv)
{
  return printReady(&pv->lcd);
}
cv_status cv_voip_readyFromDialing(cv_voip* pv)
{
  return printReady(&pv->lcd);
}
cv_status cv_voip_readyFromCallSent(cv_voip* pv)
{
  return printReady(&pv->lcd);
}

cv_status cv_voip_remoteRingingFromCallSent(cv_voip* pv)
{
  cv_status status = cv_status_success;

  status = cv_lcd_clear(&pv->lcd);
  cv_status_returnIfFailed(status);
  status = cv_lcd_print(&pv->lcd, "*** ringing ***");
  cv_status_returnIfFailed(status);

  cv_status_return(status);
}

cv_status cv_voip_readyFromInitialized(cv_voip* pv)
```

```
{
  cv_msg msg;
  cv_status status = cv_status_success;

  printReady(&pv->lcd);

  /* tell sip to register with asterisk */
  status = cv_msg_init(&msg, eMsg_register, 0);
  cv_status_returnIfFailed(status);

  status = cv_mbox_write(pv->smbx, &msg);
  cv_status_returnIfFailed(status);

  cv_status_return(status);
}
cv_status cv_voip_readyFromInCall(cv_voip* pv)
{
  cv_status status = cv_status_success;

  status = cv_lcd_clear(&pv->lcd);
  cv_status_returnIfFailed(status);
  status = cv_lcd_print(&pv->lcd, "**** ready ****");
  cv_status_returnIfFailed(status);

  cv_status_return(status);
}
cv_status cv_voip_busyFromCallSent(cv_voip* pv)
{
  cv_status status = cv_status_success;

  status = cv_lcd_clear(&pv->lcd);
  cv_status_returnIfFailed(status);
  status = cv_lcd_print(&pv->lcd, "****  busy  ****");
  cv_status_returnIfFailed(status);

  cv_status_return(status);
}
cv_status cv_voip_callSentFromDialing(cv_voip* pv)
{
  cv_status status = cv_status_success;

  status = cv_lcd_clear(&pv->lcd);
  cv_status_returnIfFailed(status);
  status = cv_lcd_print(&pv->lcd, "*** calling ***");
  cv_status_returnIfFailed(status);

  cv_status_return(status);
}
```

>> cv-bpool.h

```
#ifndef _buffpool_h__fa285c6c_2966_42d6_9add_ba364fcff348
#define _buffpool_h__fa285c6c_2966_42d6_9add_ba364fcff348


typedef struct buffNode buffNode;
typedef struct buffTrack buffTrack;
```

```c
typedef struct cv_buffpool
{
  int32 buffsz_;
  int32 mcap_;
  int32 ccap_;
  int32 usage_;

  buffNode* head_;
  buffTrack* buffers_;

} cv_buffpool;

cv_status cv_buffpool_construct(cv_buffpool* pbp, int32 buffsz,
                       int32 initialCapacity, int32 maxCapacity);
cv_status cv_buffpool_destruct(cv_buffpool* pbp);

cv_status cv_buffpool_alloc(cv_buffpool* pbp, uint8** ppbuff);
cv_status cv_buffpool_allocIsr(cv_buffpool* pbp, uint8** ppbuff);

cv_status cv_buffpool_free(cv_buffpool* pbp, uint8* ppbuff);
cv_status cv_buffpool_freeIsr(cv_buffpool* pbp, uint8* ppbuff);



#endif /* _buffpool_h__fa285c6c_2966_42d6_9add_ba364fcff348 */
```

>> cv-bpool.c

```c
#include <stdlib.h>
#include "defs.h"
#include "cv-bpool.h"
#include "basic_io.h"


static alt_irq_context cpu_statusreg;
#define cli cpu_statusreg = alt_irq_disable_all()
#define sti alt_irq_enable_all(cpu_statusreg)

static cv_status cv_buffpool_grow(cv_buffpool* pbp, int32 newSize);

struct buffNode
{
  buffNode* next;
};
struct buffTrack
{
  buffTrack* next;
};

cv_status cv_buffpool_construct(cv_buffpool* pbp, int32 buffsz,
                                int32 initialCapacity, int32 maxCapacity)
{
  cv_status status = cv_status_success;
  pbp->buffsz_  = buffsz;
  pbp->mcap_    = maxCapacity;
  pbp->buffers_ = NULL;
  pbp->usage_   = 0;
  pbp->ccap_    = 0;
  pbp->head_    = NULL;

  if(initialCapacity > 0)
    return cv_buffpool_grow(pbp, initialCapacity);

  cv_status_return(status);
}


cv_status cv_buffpool_destruct(cv_buffpool* pbp)
```

```
{
  cv_status status = cv_status_success;
  buffTrack* bt = NULL, *next;
  /* assert pbp->usage_ == 0 */

  cli;
  bt = pbp->buffers_;
  pbp->buffers_ = NULL;
  sti;

  while(bt) {
    next = bt->next;
    free(bt);
    bt = next;
  }

  cv_status_return(status);
}

cv_status cv_buffpool_alloc(cv_buffpool* pbp, uint8** ppbuff)
{
  cv_status status = cv_status_success;

  cli;
  if(pbp->head_) {
    *ppbuff = (uint8*) pbp->head_;
    pbp->head_ = pbp->head_->next;
    pbp->usage_++;
  } else status = cv_status_failure;
  sti;

  if(status != cv_status_success) {
    status = cv_buffpool_grow(pbp, pbp->ccap_);
    if(cv_status_succeeded(status))
      return cv_buffpool_alloc(pbp, ppbuff);
  }
  cv_status_return(status);
}

cv_status cv_buffpool_allocIsr(cv_buffpool* pbp, uint8** ppbuff)
{
  cv_status status = cv_status_success;
  /* note: CANNOT grow here ... we are in isr context! */

  if(pbp->head_) {
    *ppbuff = (uint8*) pbp->head_;
    pbp->head_ = pbp->head_->next;
    pbp->usage_++;
  } else status = cv_status_failure;

  cv_status_return(status);
}

cv_status cv_buffpool_free(cv_buffpool* pbp, uint8* pbuff)
{
  cv_status status = cv_status_success;

  cli;
  status = cv_buffpool_freeIsr(pbp, pbuff);
  sti;

  cv_status_return(status);
}

cv_status cv_buffpool_freeIsr(cv_buffpool* pbp, uint8* pbuff)
{
  cv_status status = cv_status_success;

  buffNode* bn = (buffNode*)pbuff;
  bn->next = pbp->head_;
  pbp->head_ = bn;
```

```
  pbp->usage_--;

  cv_status_return(status);
}

static cv_status cv_buffpool_grow(cv_buffpool* pbp, int32 newSize)
{
  cv_status status = cv_status_success;
  buffTrack* bt;
  buffNode* bn, *first, *prev = NULL;
  uint32 sz = newSize;

  /* one alloc for all */
  bt = (buffTrack*) malloc(sizeof(buffTrack) + newSize * pbp->buffsz_);
  first = bn = (buffNode*)(bt+1);

  while(sz--)
  {
    bn->next = (buffNode*)(((uint8*)bn) + pbp->buffsz_);
    prev = bn;
    bn = bn->next;
  }

  cli;

  bt->next = pbp->buffers_;
  pbp->buffers_ = bt;

  prev->next = pbp->head_;
  pbp->head_ = first;
  pbp->ccap_ += newSize;

  sti;

  printf("pool grow: %d, head:%p\n", pbp->ccap_, pbp->head_);

  cv_status_return(status);
}
```

>> cv-queue.h

```
#ifndef __cv_queue_h__67440bf0_f380_4f92_bb64_a4547bdf9d09
#define __cv_queue_h__67440bf0_f380_4f92_bb64_a4547bdf9d09

#include <ucos_ii.h>
#include "defs.h"

typedef struct cv_queueNode cv_queueNode;

struct cv_queueNode {
  cv_queueNode* next;
  cv_queueNode* prev;
};

typedef struct cv_queue
{
  uint32 size;
  cv_queueNode* head;
  cv_queueNode* tail;

  OS_EVENT*     rdy;

} cv_queue;


cv_status cv_queue_construct(cv_queue* pq);
cv_status cv_queue_push(cv_queue* pq, cv_queueNode* pn);
cv_status cv_queue_pushIsr(cv_queue* pq, cv_queueNode* pn);
cv_status cv_queue_pop(cv_queue* pq, cv_queueNode** ppn);
cv_status cv_queue_popIsr(cv_queue* pq, cv_queueNode** ppn);
```

```
cv_status cv_queue_destruct(cv_queue* pq);


#endif /* __cv_queue_h__67440bf0_f380_4f92_bb64_a4547bdf9d09 */
```

>> cv-queue.c

```
#include <stdlib.h>
#include "cv-queue.h"
#include "basic_io.h"

static alt_irq_context cpu_statusreg;
#define cli cpu_statusreg = alt_irq_disable_all()
#define sti alt_irq_enable_all(cpu_statusreg)

cv_status cv_queue_construct(cv_queue* pq)
{
  cv_status status = cv_status_success;

  pq->head = NULL;
  pq->tail = NULL;
  pq->size = 0;
  pq->rdy = OSSemCreate(0);

  cv_status_return(status);
}

cv_status cv_queue_push(cv_queue* pq, cv_queueNode* pn)
{
  cv_status status = cv_status_success;

  cli;

  pn->prev = NULL;
  pn->next = pq->head;
  if(pq->head)
    pq->head->prev = pn;
  else {
    pq->tail = pn;
  }
  pq->head = pn;
  pq->size++;

  sti;

  OSSemPost(pq->rdy);
  cv_status_return(status);
}

cv_status cv_queue_pushIsr(cv_queue* pq, cv_queueNode* pn)
{
  cv_status status = cv_status_success;

  pn->prev = NULL;
  pn->next = pq->head;
  if(pq->head)
    pq->head->prev = pn;
  else {
    pq->tail = pn;
  }
  pq->head = pn;
  pq->size++;
  OSSemPost(pq->rdy);

  cv_status_return(status);
}

cv_status cv_queue_pop(cv_queue* pq, cv_queueNode** ppn)
```

```
{
  cv_status status = cv_status_success;
  uint8 rv;

  OSSemPend(pq->rdy, 0, &rv);

  cli;
  status = cv_queue_popIsr(pq, ppn);
  sti;

  cv_status_return(status);
}

cv_status cv_queue_popIsr(cv_queue* pq, cv_queueNode** ppn)
{
  cv_status status = cv_status_success;

  *ppn = pq->tail;
  if(pq->tail) {
    if(pq->tail->prev)
      pq->tail->prev->next = NULL;
    else
      pq->head = NULL;
    pq->tail = pq->tail->prev;
    --pq->size;
  }
  cv_status_return(status);
}


cv_status cv_queue_destruct(cv_queue* pq)
{
  cv_status status = cv_status_success;
  cv_status_return(status);
}
```

>> cv-mbox.h

```
#ifndef __cv_mbox_h__f6630654_9e9c_4ff2_90d3_6157b79039df
#define __cv_mbox_h__f6630654_9e9c_4ff2_90d3_6157b79039df

#include "defs.h"

typedef struct cv_msg cv_msg;

struct cv_msg {
  cv_msg* next;
  cv_msg* prev;

  uint32 cmd;
  uint32 arg;
};

typedef struct cv_mbox {
  void*   pool;
  cv_msg* free;

  cv_msg* head;
  cv_msg* tail;

  OS_EVENT* sem;

} cv_mbox;


cv_status cv_mbox_construct(cv_mbox* mbox, int poolSz);
cv_status cv_mbox_destruct(cv_mbox* mbox);

cv_status cv_mbox_read(cv_mbox* mbox, cv_msg* ppmsg);
cv_status cv_mbox_write(cv_mbox* mbox, cv_msg* pmsg);
```

```
cv_status cv_msg_init(cv_msg* pmsg, uint32 cmd, uint32 arg);

#endif /* __cv_mbox_h__f6630654_9e9c_4ff2_90d3_6157b79039df */
```

>> cv-mbox.c

```
#include <stdlib.h>
#include <stdio.h>
#include "includes.h"
#include "cv-mbox.h"


static alt_irq_context cpu_statusreg;
#define cli cpu_statusreg = alt_irq_disable_all()
#define sti alt_irq_enable_all(cpu_statusreg)

cv_status cv_mbox_construct(cv_mbox* mbox, int poolSz)
{
  cv_status status = cv_status_success;
  cv_msg* pmsg;

  mbox->head = NULL;
  mbox->tail = NULL;

  mbox->pool = malloc(sizeof(cv_msg) * poolSz);
  if(mbox->pool == NULL) {
    printf("out of memory\n");
    return cv_status_failure;
  }

  mbox->sem = OSSemCreate(0);

  pmsg = (cv_msg*) mbox->pool;
  mbox->free = NULL;

  while(poolSz--)
  {
    if(mbox->free)
      mbox->free->prev = pmsg;
    pmsg->prev = NULL;
    pmsg->next = mbox->free;
    mbox->free = pmsg;
    pmsg++;
  }

  cv_status_return(status);
}

cv_status cv_mbox_destruct(cv_mbox* mbox)
{
  cv_status status = cv_status_success;

  free(mbox->pool);

  cv_status_return(status);
}
cv_status cv_mbox_read(cv_mbox* mbox, cv_msg* pmsg)
{
  cv_status status = cv_status_success;
  cv_msg* m = NULL;
  uint8 rv;

  OSSemPend(mbox->sem, 1, &rv);
  if(rv == OS_TIMEOUT) {
    pmsg->cmd = 0;
    pmsg->arg = 0;
    cv_status_return(status);
  }
```

```
  // pop a message from the tail of the queue
  cli;
  if( (m = mbox->tail) ) {
    if(m->prev)
      m->prev->next = NULL;
    else
      mbox->head = NULL;
    mbox->tail = m->prev;
  }
  sti;

  // copy from m to pmsg
  if(m) {
    pmsg->cmd = m->cmd;
    pmsg->arg = m->arg;
  } else {
    pmsg->cmd = 0;
    pmsg->arg = 0;
  }

  /* back into the free queue with m */
  cli;
  m->next = mbox->free;
  mbox->free = m;
  sti;

  cv_status_return(status);
}

cv_status cv_mbox_write(cv_mbox* mbox, cv_msg* pmsg)
{
  cv_status status = cv_status_success;
  cv_msg* m = NULL;

  // get a free msg buffer from the free queue
  cli;
  if( (m = mbox->free) )
    mbox->free = m->next;
  sti;

  // copy from pmsg to m
  if(m) {
    m->cmd = pmsg->cmd;
    m->arg = pmsg->arg;
  } else {
    printf("out of msg buffers\n");
    return cv_status_failure;
  }

  // queue m into mbox
  cli;

  m->next = mbox->head;
  m->prev = NULL;
  if(mbox->head)
    mbox->head->prev = m;
  else
    mbox->tail = m;
  mbox->head = m;

  sti;
  OSSemPost(mbox->sem);

  cv_status_return(status);
}

cv_status cv_msg_init(cv_msg* pmsg, uint32 cmd, uint32 arg)
{
  pmsg->cmd = cmd;
  pmsg->arg = arg;
```

```
  return cv_status_success;
}
```

>> cv-sound.h

```
#ifndef __cv_sound_h__f581191d_39fb_4bf1_b340_e2405331452d
#define __cv_sound_h__f581191d_39fb_4bf1_b340_e2405331452d

#include "defs.h"
#include "cv-bpool.h"
#include "cv-queue.h"

typedef struct cv_sound
{
  int32 intrCount;

  uint32 base;
  uint32 statBase;

  cv_buffpool* bp;

  cv_queue* inq;
  cv_queue* outq;

  int queueInput;
  int toneOutput;

  const uint16* toneBuff;
  int32 tbidx;
  int32 tbsz;

} cv_sound;


cv_status cv_sound_construct(cv_sound* snd, cv_buffpool* pbp,
                             cv_queue* pinq, cv_queue* poutq);

cv_status cv_sound_destruct(cv_sound* snd);

cv_status cv_sound_start(cv_sound* snd);
cv_status cv_sound_stop(cv_sound* snd);

cv_status cv_sound_playTone(cv_sound* snd, const uint16* buff, int bufsz);


#endif /* __cv_sound_h__f581191d_39fb_4bf1_b340_e2405331452d */
```

>> cv-sound.c

```
#include <stdlib.h>
#include <stdio.h>

#include <alt_types.h>
#include <sys/alt_irq.h>

#include <system.h>
#include <io.h>

#include "cv-sound.h"


#define CODEC_EN 0x2000
#define INTR_EN 0x4000
#define INTR_FU 0x8000

static void sound_isr(void* context, alt_u32 id);

cv_status cv_sound_construct(cv_sound* snd, cv_buffpool* pbp,
```

```
                             cv_queue* pinq, cv_queue* poutq)
{
  int stat;
  cv_status status = cv_status_success;

  snd->intrCount = 0;
  snd->base = AUD_STACK_INST_SND_BASE;
  snd->statBase = AUD_STACK_INST_SND_STAT_BASE;

  snd->inq = pinq;
  snd->outq = poutq;
  snd->bp = pbp;
  snd->queueInput = 1;
  snd->toneOutput = 0;

  IOWR_16DIRECT(snd->statBase, 0, 0);
  stat = IORD_16DIRECT(snd->statBase, 0);
  if(stat != 0)
    printf("inconsistent read on snd_stat: %d\n", stat);

  alt_irq_register(AUD_STACK_INST_SND_IRQ, (void*)snd, sound_isr);

  return status;
}

cv_status cv_sound_playTone(cv_sound* snd, const uint16* tb, int tbsz)
{
  cv_status status = cv_status_success;

  snd->toneBuff = tb;
  snd->tbsz = tbsz;
  snd->tbidx = 0;
  snd->toneOutput = 1;
  snd->queueInput = 0;

  return cv_status_success;
}

cv_status cv_sound_destruct(cv_sound* snd)
{
  cv_sound_stop(snd);
  return cv_status_success;
}

// Normal mode, 128fs BOSR,

static void sound_isr(void* context, alt_u32 id)
{
  cv_status status;
  int idx;
  cv_queueNode* pbuff = NULL;
  uint16* buff = NULL;
  volatile cv_sound* snd = (cv_sound*) context;

  /* turn ints off briefly */
  // IOWR_16DIRECT(snd->statBase, 0, INTR_FU);
  IOWR_16DIRECT(snd->statBase, 0, CODEC_EN);

  if(snd->queueInput) {
    /* read the incoming sound data from mic */
    status = cv_buffpool_allocIsr(snd->bp, (uint8**)&pbuff);
    if(status != 0) {
      //printf("OOB: %d\n", snd->intrCount);
      IOWR_16DIRECT(snd->statBase, 0, INTR_EN|CODEC_EN);
      return;
    }

    if(pbuff != NULL) {
      buff = (uint16*)(pbuff+1);
      buff += 6;
```

```
      for(idx=0; idx < 256; ++idx) {
        buff[idx] = IORD_16DIRECT(snd->base, idx);
      }
      /* queue the outgoing sound data to outq */
      status = cv_queue_pushIsr(snd->outq, pbuff);
    }
  }

  /* grab any incoming sound data, & write to speaker */
  if(snd->toneOutput) {
    for(idx = 0; idx < 256; ++idx) {
      if(snd->tbidx == snd->tbsz)
        snd->tbidx = 0;
      IOWR_16DIRECT(snd->base, idx, snd->toneBuff[snd->tbidx++]);
    }
  } else {
    while(snd->inq->size) {
      status = cv_queue_popIsr(snd->inq, &pbuff);
      buff = (uint16*)(pbuff+1);

      for(idx=0; idx < 256; ++idx) {
        IOWR_16DIRECT(snd->base, idx, buff[idx]);
      }

      status = cv_buffpool_freeIsr(&snd->bp, (uint8*)pbuff);
      cv_status_returnIfFailed(status);
    }
  }
  /* re-enable interrupts */
  ++snd->intrCount;
  IOWR_16DIRECT(snd->statBase, 0, (INTR_EN | CODEC_EN));
}

cv_status cv_sound_start(cv_sound* snd)
{
  int stat;
  IOWR_16DIRECT(snd->statBase, 0, (INTR_EN | CODEC_EN));
  stat = IORD_16DIRECT(snd->statBase, 0);
  if(stat != (INTR_EN | CODEC_EN))
    printf("inconsistent read on snd_stat: %d\n", stat);

  return cv_status_success;
}

cv_status cv_sound_stop(cv_sound* snd)
{
  /* peripheral should flush sample buffers on disable... */
  IOWR_16DIRECT(snd->statBase, 0, 0);
  return cv_status_success;
}
```

>> cv-lcd.h

```
#ifndef __cv_lcd_h__9ef40604_7ad1_454f_b1a3_565c869c31c7
#define __cv_lcd_h__9ef40604_7ad1_454f_b1a3_565c869c31c7


#include "defs.h"

typedef struct cv_lcd {

  unsigned int base;
  int fd;

} cv_lcd;

cv_status cv_lcd_construct(cv_lcd* lcd);

cv_status cv_lcd_clear(cv_lcd* lcd);
```

```c
cv_status cv_lcd_print(cv_lcd* lcd, const char* txt);
cv_status cv_lcd_appendChar(cv_lcd* lcd, char ch);
cv_status cv_lcd_backspace(cv_lcd* lcd);
cv_status cv_lcd_line2(cv_lcd* lcd);

cv_status cv_lcd_destruct(cv_lcd* lcd);


#endif /* __cv_lcd_h__9ef40604_7ad1_454f_b1a3_565c869c31c7 */
```

>> cv-lcd.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <io.h>
#include "system.h"
#include "cv-lcd.h"

#define lcd_write_cmd(base, data)                     IOWR(base, 0, data)
#define lcd_read_cmd(base)                            IORD(base, 1)
#define lcd_write_data(base, data)                    IOWR(base, 2, data)
#define lcd_read_data(base)                           IORD(base, 3)

cv_status cv_lcd_construct(cv_lcd* lcd)
{
  cv_status status = cv_status_success;

  lcd->fd = open(LCD_NAME, O_WRONLY, 0);

  if(lcd->fd == -1) {
    printf("unable to open %s\n", LCD_NAME);
  }

  cv_status_return(status);
}

cv_status cv_lcd_print(cv_lcd* lcd, const char* txt)
{
  cv_status status = cv_status_success;
  int slen;

  slen = strlen(txt);
  write(lcd->fd, txt, slen);

  cv_status_return(status);
}

cv_status cv_lcd_appendChar(cv_lcd* lcd, char ch)
{
  cv_status status = cv_status_success;

  write(lcd->fd, &ch, 1);

  cv_status_return(status);
}

cv_status cv_lcd_clear(cv_lcd* lcd)
{
  char buf[4] = { 27, '[', '2', 'J' };

  write(lcd->fd, buf, 4);

  return cv_status_success;
}
```

```
cv_status cv_lcd_line2(cv_lcd* lcd)
{
  cv_status status = cv_status_success;
  char ch = '\n';

  write(lcd->fd, &ch, 1);

  cv_status_return(status);
}

cv_status cv_lcd_destruct(cv_lcd* lcd)
{
  cv_status status = cv_status_success;

  // no interrupts, nothing to do.
  cv_lcd_print(lcd, "bye, bye");

  cv_status_return(status);
}

cv_status cv_lcd_backspace(cv_lcd* lcd)
{
  cv_status status = cv_status_success;
  char chb[3] = {'\b', ' ', '\b' };

  write(lcd->fd, chb, sizeof(chb));

  cv_status_return(status);
}
```

>> cv-kbd.h

```
#ifndef __cv_kbd_h_dd568b6f_17b7_4714_b248_9ec0038400df
#define __cv_kbd_h_dd568b6f_17b7_4714_b248_9ec0038400df

#include "defs.h"

typedef enum kbdConstants {
  kbdUp    = 0x1075,
  kbdDown  = 0x1072,
  kbdLeft  = 0x106b,
  kbdRight = 0x1074,
  kbdBS    = 0x1066,
  kbdDel   = 0x1071,
  kbdHome  = 0x106c,
  kbdEnd   = 0x1069,
  kbdTimeout = 0x11ff
} kbdConstants;

#define isAscii(code) ( (code >> 8) == 0)
#define mkAscii(code) ((uint8)(code & 0xFF))

typedef struct cv_kbd {

  uint32 keyFlags;

} cv_kbd;


cv_status cv_kbd_construct(cv_kbd* kbd);
cv_status cv_kbd_destruct(cv_kbd* kbd);
int cv_kbd_getInput(cv_kbd* pea);
int cv_kbd_pollChar(cv_kbd* kbd);


#endif /* __cv_kbd_h_dd568b6f_17b7_4714_b248_9ec0038400df */
```

>> cv-kbd.c

```
#include "cv-kbd.h"
#include "alt_up_ps2_port.h"

#define NUM_SCAN_CODES  104
#define SHFT_MASK  1
#define ALT_MASK   2
#define CTRL_MASK  4

#define ctrlHeld(flags) (flags & CTRL_MASK)
#define shftHeld(flags) (flags & SHFT_MASK)
#define altHeld(flags)  (flags & ALT_MASK)


/////////////////////////////////////////////////////////////////////
// Table of scan code, make code and their corresponding values
// These data are useful for developing more features for the keyboard
//
alt_u8 *key_table[NUM_SCAN_CODES] = {
  "A", "B", "C", "D", "E", "F", "G", "H",
  "I", "J", "K", "L", "M", "N", "O", "P",
  "Q", "R", "S", "T", "U", "V", "W", "X",
  "Y", "Z", "0", "1", "2", "3", "4", "5",
  "6", "7", "8", "9", "`", "-", "=", "\\",
  "BKSP", "SPACE", "TAB", "CAPS", "L SHFT", "L CTRL", "L GUI", "L ALT",
  "R SHFT", "R CTRL", "R GUI", "R ALT", "APPS", "ENTER", "ESC", "F1",
  "F2", "F3", "F4", "F5", "F6", "F7", "F8", "F9",
  "F10", "F11", "F12", "SCROLL", "[", "INSERT", "HOME", "PG UP",
  "DELETE", "END", "PG DN", "U ARROW", "L ARROW", "D ARROW", "R ARROW", "NUM",
  "KP /", "KP *", "KP -", "KP +", "KP ENTER", "KP .", "KP 0", "KP 1",
  "KP 2", "KP 3", "KP 4", "KP 5", "KP 6", "KP 7", "KP 8", "KP 9",
  "]", ";", "'", ",", ".", "/","|","^"
};

alt_u8 ascii_codes[NUM_SCAN_CODES] = {
  'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
  'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',
  'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
  'Y', 'Z', '0', '1', '2', '3', '4', '5',
  '6', '7', '8', '9', '`', '-', '=', '\\',
  0x08, 0, 0x09, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0x0A, 0x1B,
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, '[', 0, 0,
  0, 0x7F, 0, 0, 0, 0, 0, 0,
  0, '/', '*', '-', '+', 0x0A, '.', '0', '1',
  '2', '3', '4', '5', '6', '7', '8', '9',
  ']', ';', '\'', ',', '.', '/', '|','^'
};

alt_u8 single_byte_make_code[NUM_SCAN_CODES] = {
  0x1C, 0x32, 0x21, 0x23, 0x24, 0x2B, 0x34, 0x33,
  0x43, 0x3B, 0x42, 0x4B, 0x3A, 0x31, 0x44, 0x4D,
  0x15, 0x2D, 0x1B, 0x2C, 0x3C, 0x2A, 0x1D, 0x22,
  0x35, 0x1A, 0x45, 0x16, 0x1E, 0x26, 0x25, 0x2E,
  0x36, 0x3D, 0x3E, 0x46, 0x0E, 0x4E, 0x55, 0x5D,
  0x66, 0x29, 0x0D, 0x58, 0x12, 0x14,    0, 0x11,
  0x59,    0,    0,    0,    0, 0x5A, 0x76, 0x05,
  0x06, 0x04, 0x0C, 0x03, 0x0B, 0x83, 0x0A, 0x01,
  0x09, 0x78, 0x07, 0x7E, 0x54,    0,    0,    0,
  0,    0,    0,    0,    0,    0,    0, 0x77,
  0, 0x7C, 0x7B, 0x79,    0, 0x71, 0x70, 0x69,
  0x72, 0x7A, 0x6B, 0x73, 0x74, 0x6C, 0x75, 0x7D,
  0x5B, 0x4C, 0x52, 0x41, 0x49, 0x4A };

  alt_u8 multi_byte_make_code[NUM_SCAN_CODES] = {
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
```

```
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0x1F, 0,
    0, 0x14, 0x27, 0x11, 0x2F, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0x70, 0x6C, 0x7D,
    0x71, 0x69, 0x7A, 0x75, 0x6B, 0x72, 0x74, 0,
    0x4A, 0, 0, 0, 0x5A, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0 };
////////////////////////////////////////////////////////////////////

// States for the Keyboard Decode FSM
typedef enum
{
  STATE_INIT,
  STATE_LONG_BINARY_MAKE_CODE,
  STATE_BREAK_CODE ,
  STATE_DONE
} DECODE_STATE;

//helper function for get_next_state
static alt_u8 get_multi_byte_make_code_index(alt_u8 code)
{
  alt_u8 i;
  for (i = 0; i < NUM_SCAN_CODES; i++ ) {
    if ( multi_byte_make_code[i] == code )
      return i;
  }
  return NUM_SCAN_CODES;
}

//helper function for get_next_state
static alt_u8 get_single_byte_make_code_index(alt_u8 code)
{
  alt_u8 i;
  for (i = 0; i < NUM_SCAN_CODES; i++ ) {
    if ( single_byte_make_code[i] == code )
      return i;
  }
  return NUM_SCAN_CODES;
}

//helper function for read_make_code
/* FSM Diagram (Main transitions)
 * Normal bytes: bytes that are not 0xF0 or 0xE0
 ______
 |     |
 |     |
 |    INIT ------ 0xF0 ----> BREAK CODE
 |     |                     /  |
 |     |                    /   |
 |    0xE0                 /    |
 Normal |                  /    Normal
 |     |      ----0xF0--->      |
 |     V     /                  |
 |    LONG  /                   V
 | MAKE/BREAK --- Normal ----> DONE
 |    CODE                       ^
 X-------------------------------|

*/

#define KB_RESET 0xFF
#define KB_SET_DEFAULT 0xF6
#define KB_DISABLE 0xF5
#define KB_ENABLE 0xF4
#define KB_SET_TYPE_RATE_DELAY 0xF3

/**
 * @brief The Enum type for the type of keyboard code received
```

```
 **/
typedef enum
{
    /** @brief --- Make Code that corresponds to an ASCII character.
        For example, the ASCII Make Code for letter <tt>A</tt> is 1C
     */
  KB_ASCII_MAKE_CODE = 1,
    /** @brief --- Make Code that corresponds to a non-ASCII character.
        For example, the Binary (Non-ASCII) Make Code for
        <tt>Left Alt</tt> is 11
     */
  KB_BINARY_MAKE_CODE = 2,
    /** @brief --- Make Code that has two bytes (the first byte is E0).
        For example, the Long Binary Make Code for <tt>Right Alt</tt>
        is "E0 11"
     */
  KB_LONG_BINARY_MAKE_CODE = 3,
    /** @brief --- Normal Break Code that has two bytes (the first byte is F0).
        For example, the Break Code for letter <tt>A</tt> is "F0 1C"
     */
  KB_BREAK_CODE = 4,
    /** @brief --- Long Break Code that has three bytes (the first two bytes
        are E0, F0). For example, the Long Break Code for <tt>Right Alt</tt>
        is "E0 F0 11"
     */
  KB_LONG_BREAK_CODE = 5,
    /** @brief --- Codes that the decode FSM cannot decode
     */
  KB_INVALID_CODE = 6
} KB_CODE_TYPE;


static DECODE_STATE get_next_state(DECODE_STATE state,
                            alt_u8 byte,
                            KB_CODE_TYPE *decode_mode,
                            alt_u8 *buf)
{
  DECODE_STATE next_state = STATE_INIT;
  alt_u16 idx = NUM_SCAN_CODES;

      switch (state) {
        case STATE_INIT:
          if ( byte == 0xE0 ) {
            next_state = STATE_LONG_BINARY_MAKE_CODE;
          } else if (byte == 0xF0) {
            next_state = STATE_BREAK_CODE;
          } else {
            idx = get_single_byte_make_code_index(byte);
            if ( (idx < 40 || idx == 68 || idx > 79) && ( idx != NUM_SCAN_CODES ) ) {
              *decode_mode = KB_ASCII_MAKE_CODE;
              *buf= ascii_codes[idx];
            } else {
              *decode_mode = KB_BINARY_MAKE_CODE;
              *buf = byte;
            }
            next_state = STATE_DONE;
          }
          break;
        case STATE_LONG_BINARY_MAKE_CODE:
          if ( byte != 0xF0 && byte!= 0xE0) {
            *decode_mode = KB_LONG_BINARY_MAKE_CODE;
            *buf = byte;
            next_state = STATE_DONE;
          } else {
            next_state = STATE_BREAK_CODE;
          }
          break;
        case STATE_BREAK_CODE:
          if ( byte != 0xF0 && byte != 0xE0) {
            *decode_mode = KB_BREAK_CODE;
            *buf = byte;
```

```
          next_state = STATE_DONE;
        } else {
          next_state = STATE_BREAK_CODE;
        }
        break;
      default:
        *decode_mode = KB_INVALID_CODE;
        next_state = STATE_INIT;
    }
  return next_state;
}

static unsigned char shftKeyMap(uint8 ch)
{ // okay this is a pretty horrible way to do this, but will suffice.
  switch(ch) {
    case '/': return '?';
    case ',': return '<';
    case '.': return '>';
    case ';': return ':';
    case '[': return '{';
    case ']': return '}';
    case '\\': return '|';
    case '-': return '_';
    case '=': return '+';
    case '`': return '~';
    case '\'': return '\"';
    default:
      return ch;
  }
}

static int read_make_code(KB_CODE_TYPE *decode_mode, alt_u8 *buf)
{
  alt_u8 byte = 0;
  int status_read =0;
  DECODE_STATE state = STATE_INIT;

  *decode_mode = KB_INVALID_CODE;

  do {
    status_read = read_data_byte_with_timeout(&byte, 0);
    //FIXME: When the user press the keyboard extremely fast, data may get
    //occasionally get lost

    if (status_read == PS2_ERROR)
      return PS2_ERROR;

    state = get_next_state(state, byte, decode_mode, buf);
  } while (state != STATE_DONE);

  return PS2_SUCCESS;
}

static int poll_make_code(KB_CODE_TYPE *decode_mode, alt_u8 *buf)
{
  alt_u8 byte = 0;
  int status_read =0;
  DECODE_STATE state = STATE_INIT;

  *decode_mode = KB_INVALID_CODE;

  do {
    status_read = read_data_byte_with_timeout(&byte, 1);
    //FIXME: When the user press the keyboard extremely fast, data may get
    //occasionally get lost

    if (status_read == PS2_ERROR)
      return PS2_ERROR;

    state = get_next_state(state, byte, decode_mode, buf);
  } while (state != STATE_DONE);
```

```
  return PS2_SUCCESS;
}

static alt_u32 set_keyboard_rate(alt_u8 rate)
{
  // send the set keyboard rate command
  int status_send = write_data_byte_with_ack(0xF3, DEFAULT_PS2_TIMEOUT_VAL);
  if ( status_send == PS2_SUCCESS ) {
  // we received ACK, so send out the desired rate now
    status_send = write_data_byte_with_ack(rate & 0x1F,
                                           DEFAULT_PS2_TIMEOUT_VAL);
  }
  return status_send;
}
static alt_u32 reset_keyboard()
{
  alt_u8 byte;
  // send out the reset command
  int status = write_data_byte_with_ack(0xff, DEFAULT_PS2_TIMEOUT_VAL);
  if ( status == PS2_SUCCESS) {
    // received the ACK for reset, now check the BAT result
    status = read_data_byte_with_timeout(&byte, DEFAULT_PS2_TIMEOUT_VAL);
    if (status == PS2_SUCCESS && byte == 0xAA) {
      // BAT succeed
    } else {
      // BAT failed
      status == PS2_ERROR;
    }
  }
  return status;
}

int cv_kbd_pollChar(cv_kbd* kbd)
{
  static const uint8 num_shft[10] = { ')', '!', '@', '#', '$', '%', '^', '&', '*', '(' };
  KB_CODE_TYPE decode_mode;
  uint8 key;
  int rv=0;

  rv = poll_make_code(&decode_mode, &key);
  if (rv == PS2_SUCCESS) {
    switch(decode_mode) {
      case KB_ASCII_MAKE_CODE:
        if(shftHeld(kbd->keyFlags)) {
          if(key >= 'a' && key <= 'z')
            rv = key-32;
          else if(key >= '0' && key <= '9')
            rv = num_shft[key-'0'];
          else
            rv = shftKeyMap(key);
        } else
          rv = key;
        break;
      case KB_LONG_BINARY_MAKE_CODE:
        // fall through
      case KB_BINARY_MAKE_CODE:
        switch (key) {
          case 0x5a: rv = '\n';     break;
          case 0x29: rv = ' ' ;     break;
          case 0x6c: rv = kbdHome;  break;
          case 0x69: rv = kbdEnd;   break;
          case 0x6b: rv = kbdLeft;  break;
          case 0x74: rv = kbdRight; break;
          case 0x75: rv = kbdUp;    break;
          case 0x72: rv = kbdDown;  break;
          case 0x66: rv = kbdBS;    break;
          case 0x71: rv = kbdDel;   break;
          case 0x12: kbd->keyFlags |= SHFT_MASK;
          case 0x59: kbd->keyFlags |= SHFT_MASK;
          default: /* ignore everything else for now */
```

```
          return kbdTimeout;
        };
        break;
      case KB_BREAK_CODE :
        if(key == 0x12 || key == 0x59)
          kbd->keyFlags &= ~SHFT_MASK;
      default:
        return kbdTimeout;
    }
  }
  return rv;
}

int cv_kbd_getInput(cv_kbd* kbd)
{
  static const uint8 num_shft[10] = { ')', '!', '@', '#', '$', '%', '^', '&', '*', '(' };
  KB_CODE_TYPE decode_mode;
  uint8 key;
  int rv=0;

  rv = read_make_code(&decode_mode, &key);
  if (rv == PS2_SUCCESS) {
    switch(decode_mode) {
      case KB_ASCII_MAKE_CODE:
        if(shftHeld(kbd->keyFlags)) {
          if(key >= 'a' && key <= 'z')
            rv = key-32;
          else if(key >= '0' && key <= '9')
            rv = num_shft[key-'0'];
          else
            rv = shftKeyMap(key);
        } else
          rv = key;
        break;
      case KB_LONG_BINARY_MAKE_CODE:
        // fall through
      case KB_BINARY_MAKE_CODE:
        switch (key) {
          case 0x5a: rv = '\n';     break;
          case 0x29: rv = ' ' ;     break;
          case 0x6c: rv = kbdHome;  break;
          case 0x69: rv = kbdEnd;   break;
          case 0x6b: rv = kbdLeft;  break;
          case 0x74: rv = kbdRight; break;
          case 0x75: rv = kbdUp;    break;
          case 0x72: rv = kbdDown;  break;
          case 0x66: rv = kbdBS;    break;
          case 0x71: rv = kbdDel;   break;
          case 0x12: kbd->keyFlags |= SHFT_MASK;
          case 0x59: kbd->keyFlags |= SHFT_MASK;
          default: /* ignore everything else for now */
            return kbdTimeout;
        };
        break;
      case KB_BREAK_CODE :
        if(key == 0x12 || key == 0x59)
          kbd->keyFlags &= ~SHFT_MASK;
      default:
        return kbdTimeout;
    }
  }
  return rv;
}

cv_status cv_kbd_construct(cv_kbd* kbd)
{
  cv_status status = cv_status_success;

  reset_keyboard();
  // set the repeat rate here?
```

```
  cv_status_return(status);
}

cv_status cv_kbd_destruct(cv_kbd* kbd)
{
  cv_status status = cv_status_success;
  cv_status_return(status);
}
```

>> cv-msg.h

```
#ifndef __cv_msg_h__498c335f_09a4_4ebd_98fa_ef09e3dceba6
#define __cv_msg_h__498c335f_09a4_4ebd_98fa_ef09e3dceba6


typedef enum cv_msg_type {

  eMsg_none = 0,
  eMsg_register,
  eMsg_unregister,
  eMsg_invite,
  eMsg_ringing,
  eMsg_busy,
  eMsg_hup,
  eMsg_answer,
  eMsg_number,
  eMsg_decline,
  eMsg_unavailable,

} cv_msg_type;


#endif /* __cv_msg_h__498c335f_09a4_4ebd_98fa_ef09e3dceba6 */
```

>> cv-rtp.h

```
#ifndef cv_rtp_h__faa0253e_d7a8_49e5_a6f6_809dd6c81f75
#define cv_rtp_h__faa0253e_d7a8_49e5_a6f6_809dd6c81f75

#include "defs.h"
#include "rtp_embedded.h"


typedef struct cv_rtp
{
  SOCKET    sock[2];

  uint16    remotePort;
  char      remoteAddr[32];
  char      localAddr[32];
  char      cname[64];

  int       nfsc;
  int       cid;

} cv_rtp;


typedef int (* sipCallback)(const char* packet, cv_rtp* rtps, void *pthis);


cv_status cv_rtp_construct(cv_rtp* prtp,
                           const char* localAddr,
                           const char* cname);

cv_status cv_rtp_start(cv_rtp* prtp);
```

```
int cv_rtp_waitReadable(cv_rtp* prtp, uint32 toMsecs);
/* returns non zero if the rtp soccket is readable  */

cv_status cv_rtp_send(cv_rtp* prtp, uint8* buff, uint32 size);
cv_status cv_rtp_recv(cv_rtp* prtp, uint8* buff, uint32 size,
                      uint32* bytesRead);
cv_status cv_rtp_stop(cv_rtp* prtp);

cv_status cv_rtp_destruct(cv_rtp* prtp);



#endif /* cv_rtp_h__faa0253e_d7a8_49e5_a6f6_809dd6c81f75 */
```

>> cv-rtp.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define BLEUGH
#include "includes.h"
#include "cv-rtp.h"
// #include "rtp_embedded.h"
#include "rtp_api.h"
#include "rtp_highlevel.h"


/* Functions that implement the RTP Scheduler */

/* We maintain a simple queue of events. */
/* If you're not using the library in a simple command-line tool like this,
   you will probably need to tie in to your UI library's event queue
   somehow, instead of using this simple approach.*/

#define PAYLOAD_TYPE_MULAW_8 0

struct evt_queue_elt {
  context cid;
  rtp_opaque_t event_opaque;
  int event_time;
  struct evt_queue_elt *next;
};

static struct evt_queue_elt* evt_queue = NULL;

static void insert_in_evt_queue(struct evt_queue_elt *elt)
{
  if (evt_queue == NULL || elt->event_time < evt_queue->event_time) {
    elt->next = evt_queue;
    evt_queue = elt;
  }
  else {
    struct evt_queue_elt *s = evt_queue;
    while (s != NULL) {
      if (s->next == NULL || elt->event_time < s->next->event_time) {
        elt->next = s->next;
        s->next = elt;
        break;
      }
      s = s->next;
    }
  }
}

void RTPSchedule(context cid, rtp_opaque_t opaque, struct timeval *tp)
{
```

```
  struct evt_queue_elt *elt;

  elt = (struct evt_queue_elt *) malloc(sizeof(struct evt_queue_elt));
  if (elt == NULL)
    return;

  elt->cid = cid;
  elt->event_opaque = opaque;
  elt->event_time = tp->tv_sec * 1000 + tp->tv_usec / 1000;

  insert_in_evt_queue(elt);
}

cv_status cv_rtp_send(cv_rtp* prtp, uint8* buffer, uint32 size)
{
  rtperror err, marker=1;
  err = RTPSend(prtp->cid, 1, marker, PAYLOAD_TYPE_MULAW_8, buffer, size);
  return err;
}

int waitReadable(SOCKET s, int tomsecs)
{
  fd_set read;
  struct timeval tv;

  tv.tv_sec = 0;
  tv.tv_usec = tomsecs * 1000;
  FD_ZERO(&read);
  FD_SET(s, &read);
  return (select(s, &read, NULL, NULL, &tv) > 0);
}

int cv_rtp_waitReadable(cv_rtp* prtp, uint32 tomsecs)
{
  return waitReadable(prtp->sock[0], tomsecs);
}

cv_status cv_rtp_recv(cv_rtp* prtp, uint8* buffer, uint32 size,
                      uint32* bytesRead)
{
  rtperror err;
  err = RTPReceive(prtp->cid, prtp->sock[0], (char*)buffer, &size);
  *bytesRead = size;
  return err;
}

cv_status cv_rtp_construct(cv_rtp* prtp,
                           const char* laddr,
                           const char* cname)
{
  cv_status status = cv_status_success;
  int clen;

  prtp->sock[0] = INVALID_SOCKET;
  prtp->sock[1] = INVALID_SOCKET;
  prtp->remotePort = 0;

  memset(prtp->remoteAddr, 0, sizeof(prtp->remoteAddr));
  memset(prtp->cname, 0, sizeof(prtp->cname));
  memset(prtp->localAddr, 0, sizeof(prtp->localAddr));

  if(cname == NULL)
    return cv_status_failure;

  clen = strlen(cname);
  if(clen > sizeof(prtp->cname)-1)
    return cv_status_failure;

  strcpy(prtp->cname, cname);
  strcpy(prtp->localAddr, laddr);
```

```
  cv_status_return(status);
}

cv_status cv_rtp_stop(cv_rtp* prtp)
{
  rtperror err;
  cv_status status = cv_status_success;

  if(prtp->cid == 0)
    cv_status_return(status);

  if((err = RTPCloseConnection(prtp->cid,"Goodbye!")) != RTP_OK)
    status = cv_status_failure;
  if ((err = RTPDestroy(prtp->cid)) != RTP_OK)
    status = cv_status_failure;

  prtp->cid = 0;

  cv_status_return(status);
}

cv_status cv_rtp_start(cv_rtp* prtp)
{
  cv_status status = cv_status_success;
  rtperror err;
  unsigned char ttl = 1;

  socktype sockt;
  int nfds = 0;

  err = RTPCreate(&prtp->cid);
  if (err != RTP_OK) {
    fprintf(stderr, "%s\n", RTPStrError(err));
    return -1;
  }

  err = RTPSessionAddSendAddr(prtp->cid, prtp->remoteAddr, prtp->remotePort, ttl);
  if (err != RTP_OK) {
    fprintf(stderr, "%s\n", RTPStrError(err));
    return -1;
  }

  err = RTPSessionSetReceiveAddr(prtp->cid, prtp->localAddr, prtp->remotePort);
  if (err != RTP_OK) {
    fprintf(stderr, "%s\n", RTPStrError(err));
    return -1;
  }

  err = RTPMemberInfoSetSDES(prtp->cid, 0, RTP_MI_CNAME, prtp->cname);
  if (err != RTP_OK) {
    fprintf(stderr, "%s\n", RTPStrError(err));
    return -1;
  }

  err = RTPMemberInfoSetSDES(prtp->cid, 0, RTP_MI_NAME, "rtp blows");
  if (err != RTP_OK) {
    fprintf(stderr, "%s\n", RTPStrError(err));
    return -1;
  }

  err = RTPOpenConnection(prtp->cid);
  if (err != RTP_OK) {
    fprintf(stderr, "%s\n", RTPStrError(err));
    return -1;
  }

  err = RTPSessionGetRTPSocket(prtp->cid, &sockt);
  if (err != RTP_OK) {
    fprintf(stderr, "%s\n", RTPStrError(err));
    return -1;
  }
```

```
  prtp->sock[0] = sockt;
  nfds = 0;

#ifdef __unix
  if (nfds < sockt) nfds = sockt;
#endif

  err = RTPSessionGetRTCPSocket(prtp->cid, &sockt);
  if (err != RTP_OK) {
    fprintf(stderr, "%s\n", RTPStrError(err));
    return -1;
  }
  prtp->sock[1] = sockt;

#ifdef __unix
  if (nfds < sockt) nfds = sockt;
#endif

  prtp->nfsc = nfds;

  cv_status_return(status);
}

cv_status cv_rtp_destruct(cv_rtp* prtp)
{
  cv_status status = cv_status_success;

  status = cv_rtp_stop(prtp);

  cv_status_return(status);
}
```

>> cv-sip.h

```
#ifndef __cv_sip_h__93737e04_3cf7_4b8b_a7d1_bea7326274dc
#define __cv_sip_h__93737e04_3cf7_4b8b_a7d1_bea7326274dc

#include "defs.h"
#include "cv-mbox.h"

typedef struct _sip_t {
  char         registrar[16];
  unsigned int localNumber;
  unsigned int remoteNumber;
  char         localTag[32];
  char         remoteTag[64];
  char         branchTag[64];
  char         callID[64];
  unsigned int CSeq;
  char         dialogOp[16];
  char         remoteAddress[16];
  unsigned int remotePort;
  char         localAddress[16];
  unsigned int localPort;
  int          contentLength;
} sip_t;

typedef struct _con_t {
  int sockd;
  struct sockaddr_in src;
  struct sockaddr_in dest;
  char buffer[1024];
  struct timeval tv;
} con_t;

typedef struct _sdp_t {
  unsigned long sessID;
```

```c
  unsigned long sessVer;
  int port;
  char addr[16];
} sdp_t;


//struct containing rtp data.
typedef enum _run { RUN, STOP } run_type;

typedef struct cv_sip {
  sip_t regSIP;
  sip_t incomingSIP;
  sip_t outgoingSIP;
  sdp_t incomingSDP;
  sdp_t outgoingSDP;
  // rtp_t rtpDat;
  con_t sipCon;
  int online;
  int incomingCall;
  int outGoingCall;
  int callInProgress;
  int regTime;

  cv_mbox mbox;
  cv_mbox *uibox;
} cv_sip;


cv_status cv_sip_construct(cv_sip*      psip,
                           const char*  localAddr,
                           const char*  registrarAddr,
                           unsigned int sipPort,
                           unsigned int rtpPort,
                           cv_mbox*     ambx);

cv_status cv_sip_destruct(cv_sip* psip);


#endif /* __cv_sip_h__93737e04_3cf7_4b8b_a7d1_bea7326274dc */
```

>> cv-sip.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#include "includes.h"
#include "socket.h"
#include "cv-sip.h"
#include "cv-mbox.h"
#include "cv-msg.h"

#undef TRUE
#define TRUE  1
#define FALSE 0

typedef enum {
  NEWNUM=17,
  REGISTER=12,UNREGISTER=13,INVITE=14,ACK=15,BYE=16,
  CANCEL=17,
  TRYING=0,RINGING=1,
  OK=2,
  FORBIDDEN=3,NOTFOUND=4,BUSY=5,
  TERMINATED=6,UNDECIPHERABLE=7,
  UNAVAILABLE=8,
  DECLINE=9,
  ERROR=10,
  NONE=11
```

```c
} sip_com_t;

//not public function defs:
void change_number(cv_sip *context, int number);
//finctions that go out and initite action

//change internal sip vars
cv_msg_type sip_com2Msg(sip_com_t sip);
void init_sip(sip_t* dat);
int set_registrar(sip_t *dat, const char* registrar);
void set_local_number(sip_t *dat, unsigned int to);
void set_remote_number(sip_t *dat, unsigned int from);
void set_local_sip_port(sip_t *dat, const unsigned int localPort);
void set_local_sip_addr(sip_t *dat);
void generate_CSeq(sip_t *dat);
void generate_branchTag(sip_t *dat);
void generate_localTag(sip_t *dat);
void generate_callID(sip_t *dat);
void set_content_length(sip_t *dat, const unsigned int len);
//change internal sdp vars
void init_sdp(sdp_t* dat);
void set_rtp_addr(sdp_t* dat,const char *addr);
void set_rtp_port(sdp_t* dat, int rtpport);
int get_sdpmsg_len(sdp_t* dat);
int get_sdpmsg_from_data(char* buf, sdp_t* dat);
int get_data_from_sdpmsg(sdp_t* dat, char* buf);
//functions that manipulate a socket stream
int create_connection(con_t *con, char *addr, int localPort,
    int remotePort);
int recv_conbuffer(con_t *con);
int send_conbuffer(con_t *con);
int close_connection(con_t *con);
void set_con_timeout(con_t *con, int sec, int usec);
//takes an input buffer pointer and returns the command.
sip_com_t get_command_from_sipmsg(char *buffer);
//takes an input buffer and populates a sip struct with
//the message data
int get_data_from_sipmsg(sip_t *dat, char *buffer);
//takes an input msg and populates a buffer with an
//appropriate sip msg
int get_sipmsg_from_data(char *buffer, sip_t *dat, sip_com_t com);


sip_com_t go_online(cv_sip *context);
sip_com_t go_offline(cv_sip *context);
sip_com_t make_call(cv_sip *context, int number);
sip_com_t end_call(cv_sip *context);
sip_com_t accept_call(cv_sip *context);
sip_com_t reject_call(cv_sip *context);
sip_com_t send_cancel(cv_sip *context);
sip_com_t send_busy(cv_sip *context);
//calls that reply to initiated action
int get_invite(cv_sip *context);
void start_call(cv_sip *context);
void get_cancel(cv_sip *context);
void  ack(cv_sip *context);

static cv_sip* g_ps = NULL;
void sip_main(void* arg);

TK_OBJECT(to_siptask);
TK_ENTRY(sip_main);

struct inet_taskinfo siptask = {
  &to_siptask,
  "sip-main",
  sip_main,
  5,
  16384
};
```

```c
cv_status cv_sip_construct(cv_sip* psip,
                           const char*  localAddr,
                           const char*  regaddr,
                           unsigned int sipport,
                           unsigned int rtpport,
                           cv_mbox*     ambx
                          )
{
  cv_status status = cv_status_success;

  status = cv_mbox_construct(&psip->mbox, 5);
  cv_status_returnIfFailed(status);
  psip->uibox = ambx;

  srand(0);
  init_sip(&psip->regSIP);
  init_sip(&psip->incomingSIP);
  init_sip(&psip->outgoingSIP);
  init_sdp(&psip->outgoingSDP);
  set_registrar(&psip->regSIP,regaddr);
  set_registrar(&psip->incomingSIP,regaddr);
  set_registrar(&psip->outgoingSIP,regaddr);
  set_local_sip_addr(&psip->regSIP);
  set_local_sip_addr(&psip->incomingSIP);
  set_local_sip_addr(&psip->outgoingSIP);
  set_rtp_addr(&psip->outgoingSDP,psip->regSIP.localAddress);
  set_local_sip_port(&psip->regSIP, sipport);
  set_local_sip_port(&psip->incomingSIP, sipport);
  set_local_sip_port(&psip->outgoingSIP, sipport);
  set_rtp_port(&psip->outgoingSDP,rtpport);
  generate_CSeq(&psip->regSIP);
  generate_CSeq(&psip->outgoingSIP);
  generate_callID(&psip->regSIP);

  psip->online = FALSE;
  psip->incomingCall = FALSE;
  psip->callInProgress = FALSE;
  psip->regTime = 0;

  g_ps = psip;
  TK_NEWTASK(&siptask);

  cv_status_return(status);
}

cv_status cv_sip_destruct(cv_sip* psip)
{
  cv_status status = cv_status_success;
  cv_status_return(status);
}

void sip_mainloop(cv_sip *context)
{
  cv_msg_type mcom;
  sip_com_t com;
  cv_msg    msg;
  sip_com_t reply;
  struct timeval tv;
  int number;

  printf("sip_mainloop: entry\n");

  //main loop here
  while(1) {
    reply = NONE;
    number = 0;

    //check mbox first
    cv_mbox_read(&context->mbox, &msg);
    mcom   = msg.cmd;
```

```
number = msg.arg;

if(context->online == FALSE) {
  //offline activities
  if(mcom == eMsg_register)
    go_online(context);
  else if(mcom == eMsg_number)
    change_number(context,number);
}
else {
  //check registration stat
  gettimeofday(&tv,NULL);
  if(tv.tv_sec-context->regTime > 598)
    go_online(context);
  if(context->callInProgress == TRUE) {
    if(mcom == eMsg_hup)
      reply = end_call(context);
  } else {
    //initiated msg
    if(mcom == eMsg_invite)
      reply = make_call(context,number);
    else if(mcom == eMsg_hup)
      reply = send_cancel(context);
    //msg response
    else if(mcom == eMsg_answer)
      reply = accept_call(context);

    // TODO: needed?
    //else if(com == DECLINE)
    //  reply = reject_call(context);

  }
}
if(reply != NONE) {
  cv_msg_init(&msg, sip_com2Msg(reply), 0);
  cv_mbox_write(context->uibox, &msg);
}

//now check the socket
set_con_timeout(&context->sipCon,0,5000);
recv_conbuffer(&context->sipCon);
//if(strlen(context->sipCon.buffer) != 0)
//  printf("Got:\n%s\n",context->sipCon.buffer);
com = get_command_from_sipmsg(context->sipCon.buffer);
if(com != NONE)
{
  printf("com: %d\n",com);
}
if(context->callInProgress == TRUE) {
  if(com == INVITE) {
    send_busy(context);
    com = NONE;
  }
  else if(com == BYE)
    end_call(context);
} else {
    //responses
  if(com == OK)
    start_call(context);
  else if(com == BUSY)
    ack(context);
  //new requests
  else if(com == INVITE) {
    number = get_invite(context);
    if(number != 0)
      com = INVITE;
    else
      com = NONE;
  }
  else if(com == CANCEL)
    get_cancel(context);
```

```c
    }

    if(com != NONE) {
      cv_msg_init(&msg, sip_com2Msg(com), number);
      cv_mbox_write(context->uibox, &msg);
    }
  }

  printf("sip_mainloop: exit\n");

}

void sip_main(void* arg)
{
  cv_sip* psip = g_ps;
  sip_mainloop(psip);
}

void change_number(cv_sip *context, int number)
{
  int rv;

  printf("change_number: %d\n", number);

  set_local_number(&context->regSIP,      number);
  set_local_number(&context->incomingSIP, number);
  set_local_number(&context->outgoingSIP, number);
  rv = TRUE;
}

sip_com_t go_online(cv_sip *context)
{
  int rv, retries;
  sip_com_t reply = NONE;
  if(context->regSIP.localNumber == 0) {
    printf("Need to set number first!!\n");
    rv = FALSE;
  }
  else{
    if(context->online == FALSE &&
        create_connection(&context->sipCon,context->regSIP.registrar,
                          context->regSIP.localPort,5060) == FALSE) {
      perror("Connecting client");
      context->online = FALSE;
      rv = FALSE;
    }
    else {
      struct timeval tv;
      gettimeofday(&tv,NULL);
      context->regTime = tv.tv_sec;
      context->online = TRUE;
      context->regSIP.CSeq++;
      generate_branchTag(&context->regSIP);
      generate_localTag(&context->regSIP);
      set_con_timeout(&context->sipCon,3,0);
      retries = 0;
      while(reply == NONE && retries < 5) {
        get_sipmsg_from_data(context->sipCon.buffer,&context->regSIP,REGISTER);
        retries++;
        printf("Sending:\n%s\n",context->sipCon.buffer);
        send_conbuffer(&context->sipCon);
        recv_conbuffer(&context->sipCon);
        reply = get_command_from_sipmsg(context->sipCon.buffer);
      }
      if(reply != OK){
        printf("Got %d instead\n", reply);
        printf("error w/reg\n");
        rv = FALSE;
      }
      else {
        printf("Got OK\n");
```

```
        rv = TRUE;
      }
    }
  }
  context->online = TRUE;
  return rv;
}

sip_com_t go_offline(cv_sip *context)
{
  int i, retries;
  sip_com_t reply;

  context->online = FALSE;
  context->regSIP.CSeq++;
  generate_branchTag(&context->regSIP);
  generate_localTag(&context->regSIP);
  reply = NONE;
  set_con_timeout(&context->sipCon,3,0);
  retries = 0;
  while(reply == NONE && retries < 5) {
    get_sipmsg_from_data(context->sipCon.buffer,&context->regSIP,UNREGISTER);
    retries++;
    printf("Sending:\n%s\n",context->sipCon.buffer);
    send_conbuffer(&context->sipCon);
    recv_conbuffer(&context->sipCon);
    reply = get_command_from_sipmsg(context->sipCon.buffer);
  }
  generate_callID(&context->regSIP);
  context->online = FALSE;
  return reply;
}

cv_msg_type sip_com2Msg(sip_com_t sip)
{
  switch (sip)
  {
    case TRYING:      // FALL THROUGH
    case RINGING:     return eMsg_ringing;

    case BUSY:        return eMsg_busy;

    case OK:          return eMsg_answer;

    case CANCEL:      // FALL THROUGH
    case BYE:         return eMsg_hup;

    case INVITE:      return eMsg_invite;

    case UNREGISTER:  return eMsg_unregister;

    case REGISTER:    return eMsg_register;

    case DECLINE:     return eMsg_decline;
    case UNAVAILABLE: return eMsg_unavailable;

    case ACK:         return eMsg_none;

   default:
      printf("unknown return to voip control: %d\n", sip);
      return eMsg_none;
  }

}

sip_com_t make_call(cv_sip *context, int number)
{
  sip_com_t reply;
  int i, retries, msgLen;
  char *sdpPtr;
```

```
  reply = NONE;

  if(number != 0)
  {
    set_remote_number(&context->outgoingSIP, number);
    context->outgoingSIP.CSeq++;
    generate_branchTag(&context->outgoingSIP);
    generate_localTag(&context->outgoingSIP);
    generate_callID(&context->outgoingSIP);

    msgLen = get_sdpmsg_len(&context->outgoingSDP);
    set_content_length(&context->outgoingSIP, msgLen);
    set_con_timeout(&context->sipCon,3,0);
    retries = 0;
    while(reply == NONE && retries < 5 ) {
      msgLen = get_sipmsg_from_data(context->sipCon.buffer,&context->outgoingSIP,INVITE);
      sdpPtr = msgLen + context->sipCon.buffer;
      get_sdpmsg_from_data(sdpPtr,&context->outgoingSDP);
      retries++;
      printf("Sending:\n%s\n",context->sipCon.buffer);
      send_conbuffer(&context->sipCon);
      recv_conbuffer(&context->sipCon);
      reply = get_command_from_sipmsg(context->sipCon.buffer);
      printf("Got:\n%s\n",context->sipCon.buffer);
    }
    set_content_length(&context->outgoingSIP,0);
  }
  else
  {
    printf("Must set number before call!\n");
    reply = ERROR;
  }
  return reply;
}

sip_com_t end_call(cv_sip *context) {
  sip_com_t reply;
  int i;
  int retries;
  context->outgoingSIP.CSeq++;
  generate_branchTag(&context->outgoingSIP);
  reply = NONE;
  retries = 0;
  set_con_timeout(&context->sipCon,3,0);
  while(reply == NONE && retries < 5) {
    get_sipmsg_from_data(context->sipCon.buffer,&context->outgoingSIP,BYE);
    retries++;
    printf("Sending:\n%s\n",context->sipCon.buffer);
    send_conbuffer(&context->sipCon);
    for(i = 0; i < 500; i++) {
      recv_conbuffer(&context->sipCon);
      reply = get_command_from_sipmsg(context->sipCon.buffer);
      if(reply != NONE)
        break;
    }
    printf("Got:\n%s\n",context->sipCon.buffer);
  }
  context->callInProgress = FALSE;
  return reply;
}

sip_com_t accept_call(cv_sip *context)
{
  char* sdpPtr;
  sip_com_t reply;
  int i, msgLen, retries;

  //start_rtp_session(&context->rtpDat,context->incomingSDP.addr,
  //    context->outgoingSDP.port,context->incomingSDP.port);

  generate_localTag(&context->outgoingSIP);
```

```
  strcpy(context->incomingSIP.localTag,context->outgoingSIP.localTag);
  msgLen = get_sdpmsg_len(&context->outgoingSDP);
  set_content_length(&context->incomingSIP,msgLen);
  reply = NONE;
  set_con_timeout(&context->sipCon,3,0);
  retries = 0;
  while(reply == NONE && retries < 5) {
    msgLen = get_sipmsg_from_data(context->sipCon.buffer,&context->incomingSIP,OK);
    sdpPtr = context->sipCon.buffer+msgLen;
    get_sdpmsg_from_data(sdpPtr,&context->outgoingSDP);
    retries++;
    printf("Sending:\n%s\n",context->sipCon.buffer);
    send_conbuffer(&context->sipCon);
    recv_conbuffer(&context->sipCon);
    reply = get_command_from_sipmsg(context->sipCon.buffer);
    printf("Got:\n%s\n",context->sipCon.buffer);
  }
  set_content_length(&context->outgoingSIP,0);
  context->callInProgress = TRUE;
  return reply;
}

sip_com_t reject_call(cv_sip *context)
{
  sip_com_t reply;
  int i, msgLen, retries;

  msgLen = get_data_from_sipmsg(&context->incomingSIP,context->sipCon.buffer);
  if(strcmp(context->incomingSIP.callID,context->outgoingSIP.callID)!=0) {
    printf("Error: callIDs do not match\n");
    reply = ERROR;
  }
  else {
    if(msgLen == -1)
    {
      printf("invalid msg found\n");
      return 1;
    }
    get_sipmsg_from_data(context->sipCon.buffer,&context->incomingSIP,DECLINE);
    reply = NONE;
    retries = 0;
    while(reply == NONE && retries < 5)
    {
      retries++;
      printf("Sending:\n%s\n",context->sipCon.buffer);
      send_conbuffer(&context->sipCon);
      for(i = 0; i < 500; i++) {
        recv_conbuffer(&context->sipCon);
        reply = get_command_from_sipmsg(context->sipCon.buffer);
        if(reply != NONE)
          break;
      }
    }
  }
  return reply;
}

sip_com_t send_cancel(cv_sip *context) {
  sip_com_t reply;
  int msgLen;
  int i;
  int retries;
  msgLen = get_sipmsg_from_data(context->sipCon.buffer,&context->outgoingSIP,CANCEL);
  if(msgLen == -1) {
    printf("Something not right happened!\n");
    reply = ERROR;
  }
  else {
    reply = NONE;
    retries = 0;
    do{
```

```
      while(reply == NONE && retries < 5) {
        retries++;
        printf("Sending:\n%s\n",context->sipCon.buffer);
        send_conbuffer(&context->sipCon);
        for(i = 0; i < 500; i++) {
          recv_conbuffer(&context->sipCon);
          reply = get_command_from_sipmsg(context->sipCon.buffer);
          if(reply == TERMINATED)
            break;
        }
      }
      for(i = 0; i < 500; i++) {
        recv_conbuffer(&context->sipCon);
        reply = get_command_from_sipmsg(context->sipCon.buffer);
        if(reply == OK)
          break;
      }
    }while(reply == TERMINATED);
    msgLen = get_sipmsg_from_data(context->sipCon.buffer,&context->outgoingSIP,ACK);
    if(msgLen == -1) {
      printf("Something not right happened!\n");
      reply = ERROR;
    }
    printf("Sending:\n%s\n",context->sipCon.buffer);
    send_conbuffer(&context->sipCon);
  }
  return reply;
}

int get_invite(cv_sip *context)
{
  int number, msgLen;
  char *sdpPtr;

  msgLen = get_data_from_sipmsg(&context->incomingSIP,context->sipCon.buffer);
  if(msgLen == -1)
  {
    printf("Error reading msg!\n");
    number = 0;
  }
  else
  {
    generate_localTag(&context->incomingSIP);
    strcpy(context->outgoingSIP.callID,    context->incomingSIP.callID);
    strcpy(context->outgoingSIP.remoteTag, context->incomingSIP.remoteTag);
    strcpy(context->outgoingSIP.localTag,  context->incomingSIP.localTag);
    strcpy(context->outgoingSIP.branchTag, context->incomingSIP.branchTag);
    sdpPtr = msgLen + context->sipCon.buffer;
    get_data_from_sdpmsg(&context->incomingSDP,sdpPtr);
    get_sipmsg_from_data(context->sipCon.buffer, &context->incomingSIP, RINGING);
    send_conbuffer(&context->sipCon);
  }
  return context->incomingSIP.remoteNumber;
}

void start_call(cv_sip *context)
{
  char* sdpPtr;
  int msgLen;

  msgLen = get_data_from_sipmsg(&context->outgoingSIP,
                                context->sipCon.buffer);
  strcpy(context->incomingSIP.remoteTag,context->outgoingSIP.remoteTag);
  sdpPtr = context->sipCon.buffer+msgLen;
  get_data_from_sdpmsg(&context->incomingSDP,sdpPtr);
  //start_rtp_session(&context->rtpDat,context->incomingSDP.addr,
  //    context->outgoingSDP.port,context->incomingSDP.port);
  msgLen = get_sipmsg_from_data(context->sipCon.buffer,&context->outgoingSIP,ACK);
  printf("Sending:\n%s\n",context->sipCon.buffer);
  send_conbuffer(&context->sipCon);
  context->callInProgress = TRUE;
```

```
}

void get_cancel(cv_sip *context)
{
  sip_com_t reply = NONE;
  int i, retries = 0;

  while(reply == NONE && retries < 5) {
    retries++;
    get_sipmsg_from_data(context->sipCon.buffer,&context->incomingSIP,TERMINATED);
    printf("Sending:\n%s\n",context->sipCon.buffer);
    send_conbuffer(&context->sipCon);
    strcpy(context->incomingSIP.dialogOp,"CANCEL");
    get_sipmsg_from_data(context->sipCon.buffer,&context->incomingSIP,OK);
    printf("Sending:\n%s\n",context->sipCon.buffer);
    send_conbuffer(&context->sipCon);
    for(i = 0; i < 500; i++) {
      recv_conbuffer(&context->sipCon);
      reply = get_command_from_sipmsg(context->sipCon.buffer);
      if(reply != NONE)
        break;
    }
  }
}

sip_com_t send_busy(cv_sip *context) {
  sip_com_t reply;
  int i, msgLen, retries;

  msgLen = get_data_from_sipmsg(&context->incomingSIP,context->sipCon.buffer);
  if(strcmp(context->incomingSIP.callID,context->outgoingSIP.callID)!=0) {
    printf("Error: callIDs do not match\n");
    reply = ERROR;
  }
  else {
    if(msgLen == -1)
    {
      printf("invalid msg found\n");
      return 1;
    }
    get_sipmsg_from_data(context->sipCon.buffer,&context->incomingSIP,BUSY);
    reply = NONE;
    retries = 0;
    while(reply == NONE && retries < 5)
    {
      retries++;
      printf("Sending:\n%s\n",context->sipCon.buffer);
      send_conbuffer(&context->sipCon);
      for(i = 0; i < 500; i++) {
        recv_conbuffer(&context->sipCon);
        reply = get_command_from_sipmsg(context->sipCon.buffer);
        if(reply != NONE)
          break;
      }
    }
  }
  return reply;
}

void ack(cv_sip *context) {
  get_sipmsg_from_data(context->sipCon.buffer,&context->outgoingSIP,ACK);
  printf("Sending:\n%s\n",context->sipCon.buffer);
  send_conbuffer(&context->sipCon);

}
```

## >> sip_abstr.h

```
#ifndef SDP_ABSTR_H
```

```c
#define SDP_ABSTR_H

typedef struct _sdp_t sdp_t;

struct _sdp_t{
  unsigned long sessID;
  unsigned long sessVer;
  int port;
  char addr[16];
};

void set_rtp_port(sdp_t *dat, int port);
int set_rtp_addr(sdp_t *dat, const char *addr);

void init_sdp(sdp_t *dat);
//take sdp object and fill buffer with sdp message.
int get_sdpmsg_from_data(char *buffer, sdp_t *dat);
//get buffer and fill sdp object with dat
int get_data_from_sdpmsg(sdp_t *dat, char *buffer);
//get the length of a generated msg with the content
int get_sdpmsg_len(sdp_t *dat);

#endif
```

>> sip.c

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#include "socket.h"
#include "sip_abstr.h"

#define TRUE 1
#define FALSE 0

#define alt_u32 unsigned long

/*written to match reply list, so we can just look up and grab
  the corresponding command from the list w/ the enum type*/
const char *sip_com_list[11] = {
  "100 Trying","180 Ringing",
  "200 OK",
  "403 Forbidden","404 Not Found","486 Busy",
  "487 Request Terminated","493 Undecipherable",
  "503 Service Unavailable",
  "603 Decline",
  ""
};

//private functions and bufs
void generate_tag(char *tag, int len);

//public functions
void init_sip(sip_t *dat) {
  memset(dat->registrar,0,16);
  memset(dat->localTag,0,16);
  memset(dat->remoteTag,0,32);
  memset(dat->branchTag,0,64);
  memset(dat->callID,0,64);
  memset(dat->remoteAddress,0,16);
  memset(dat->localAddress,0,16);
  memset(dat->dialogOp,0,16);
  dat->localNumber = 0;
  dat->remoteNumber = 0;
  dat->remotePort = 0;
  dat->localPort = 0;
}
int set_registrar(sip_t* dat, char* registrar) {
  int status;
```

```
  struct sockaddr_in testAddr;
  if(strlen(registrar) < 15) {
    if(inet_aton(registrar,&testAddr.sin_addr)!=0) {
      strcpy(dat->registrar,registrar);
      status = TRUE;
    }
    else {
      status = FALSE;
    }
  }
  else {
    status = FALSE;
  }
  return status;
}
void set_remote_number(sip_t* dat, const unsigned int to) {
  dat->remoteNumber = to;
}
void set_local_number(sip_t* dat, const unsigned int from){
  dat->localNumber = from;
}
void set_local_sip_port(sip_t *dat, const unsigned int localPort){
  dat->localPort = localPort;
}
void set_local_sip_addr(sip_t *dat){
  alt_u32 localAddr;
  alt_u32 netmask;
  alt_u32 gw_addr;
  int use_dhcp;
  get_ip_addr(NULL,&localAddr,&netmask,&gw_addr,&use_dhcp);
  localAddr = ntohl(localAddr);
  sprintf(dat->localAddress,"%u.%u.%u.%u",
      (unsigned)(localAddr>>24)&0xff,
      (unsigned)(localAddr>>16)&0xff,
      (unsigned)(localAddr>>8)&0xff,
      (unsigned)localAddr&0xff);
 /* struct ifaddrs *interfaces;
  getifaddrs(&interfaces);
  while(interfaces !=NULL)
  {
    if(interfaces->ifa_addr->sa_family == AF_INET &&
        strcmp(interfaces->ifa_name,"lo")!=0)
      strcpy(dat->localAddress,
          inet_ntoa(((struct sockaddr_in*)interfaces->ifa_addr)->sin_addr));
    interfaces=interfaces->ifa_next;
  }*/
}

void generate_CSeq(sip_t *dat){
  dat->CSeq = (unsigned int)rand()>>4;
}
void generate_branchTag(sip_t *dat){
  strcpy(dat->branchTag,"z9hG4bK");
  generate_tag(dat->branchTag+7,32);
}
void generate_localTag(sip_t *dat){
  generate_tag(dat->localTag,32);
}
void generate_callID(sip_t *dat){
  generate_tag(dat->callID,32);
}
void set_content_length(sip_t *dat, const unsigned int len){
  dat->contentLength = len;
}

//takes an input buffer pointer and returns the command.
sip_com_t get_command_from_sipmsg(char *buffer) {
  /*read top line of incoming message.  Match it to known commands
    and return the sweet sweet results.*/
  char comline[32];
  char *endline;
```

```
  sip_com_t command;
  endline = strpbrk(buffer,"\r\n\0");
  if(endline == NULL){
    //printf("No command found!\n");
    command = NONE;
  }
  else{
    if(endline-buffer < 31)
      strncpy(comline,buffer,endline-buffer);
    else
      strncpy(comline,buffer,30);
    comline[31] = 0;
    if(strlen(comline) == 0)
      command = NONE;
    else if(strstr(comline, "INVITE") != NULL)
      command = INVITE;
    else if(strstr(comline, "ACK") != NULL)
      command = ACK;
    else if(strstr(comline, "BYE") != NULL)
      command = BYE;
    else if(strstr(comline, "CANCEL") != NULL)
      command = CANCEL;
    else if(strstr(comline, "100") != NULL)
      command = TRYING;
    else if(strstr(comline, "180") != NULL)
      command = RINGING;
    else if(strstr(comline, "200") != NULL)
      command = OK;
    else if(strstr(comline, "403") != NULL)
      command = FORBIDDEN;
    else if(strstr(comline, "404") != NULL)
      command = NOTFOUND;
    else if(strstr(comline, "486") != NULL)
      command = BUSY;
    else if(strstr(comline, "487") != NULL)
      command = TERMINATED;
    else if(strstr(comline, "493") != NULL)
      command = UNDECIPHERABLE;
    else if(strstr(comline, "503") != NULL)
      command = UNAVAILABLE;
    else if(strstr(comline, "603") != NULL)
      command = DECLINE;
    else
      command = NONE;
  }
  return command;
}
//takes an input buffer and populates a sip struct with
//the message data
int get_data_from_sipmsg(sip_t *dat, char *buffer) {
  int packetLength;
  char *wordbPtr,*wordePtr;
  int wordlen;
  wordlen = 0;
  packetLength = 0;
  char * linePtr;
  wordbPtr = strstr(buffer,"branch=");
  if(wordbPtr == NULL) {
    printf("Warning: no branch tag\n");
  }
  else {
    wordbPtr += 7;
    wordePtr = strpbrk(wordbPtr,"; \n\r");
    wordlen = wordePtr-wordbPtr;
    if(wordlen > 64-1) {
      printf("Warning: branch tag too long\n");
    }
    else {
      strncpy(dat->branchTag,wordbPtr,wordlen);
      dat->branchTag[wordlen] = 0;
    }
```

```c
}
wordbPtr = strstr(buffer,"From:");
if(wordbPtr == NULL) {
  printf("Error: Can't find From tag.\n");
  packetLength = -1;
}
else {
  linePtr = strchr(wordbPtr,'\n');
  wordbPtr = strstr(wordbPtr,"sip:");
  if(wordbPtr == NULL || wordbPtr > linePtr) {
    printf("Error: No address found.\n");
    packetLength = -1;
  }
  else {
    wordbPtr += 4;
    dat->remoteNumber = atoi(wordbPtr);
    if(dat->remoteNumber == 0) {
      printf("Error: No address found.\n");
      packetLength = -1;
    }
  }
  wordbPtr = strstr(wordbPtr,"tag");
  if(wordbPtr == NULL || wordbPtr > linePtr) {
    printf("Warning: tag not found\n");
  }
  else {
    wordbPtr += 4;
    wordePtr = strpbrk(wordbPtr," \n;\r");
    wordlen = wordePtr-wordbPtr;
    if(wordlen > 63) {
      printf("Warning: tag too long\n");
    }
    else {
      strncpy(dat->remoteTag,wordbPtr,wordlen);
      dat->remoteTag[wordlen] = 0;
    }
  }
}
wordbPtr = strstr(buffer,"CSeq:");
if(wordbPtr == NULL) {
  printf("Warning: CSeq not found\n");
}
else {
  wordbPtr += 5;
  dat->CSeq = atoi(wordbPtr);
  while(*wordbPtr == ' ')/* || *wordbPtr < '0' ||
      *wordbPtr > '9')*/
    wordbPtr++;
  wordbPtr = strpbrk(wordbPtr," \r\n\0;");
  while(*wordbPtr == ' ')/* || *wordbPtr < '0' ||
      *wordbPtr > '9')*/
    wordbPtr++;
  wordePtr = strpbrk(wordbPtr," \r\n\0;");
  wordlen = wordePtr - wordbPtr;
  strncpy(dat->dialogOp,wordbPtr,wordlen);
  dat->dialogOp[wordlen] = 0;
}
wordbPtr = strstr(buffer,"Call-ID:");
if(wordbPtr == NULL) {
  printf("Error: Can not find Call ID tag\n");
  packetLength = -1;
}
else {
  wordbPtr += 9;
  wordePtr = strpbrk(wordbPtr,"\n ;\r");
  wordlen = wordePtr - wordbPtr;
  strncpy(dat->callID,wordbPtr,wordlen);
  dat->callID[wordlen] = 0;
}
wordbPtr = strstr(buffer,"Contact:");
if(wordbPtr == NULL) {
```

```
    printf("Error: Can not find Contact tag\n");
    packetLength = -1;
  }
  else {
    linePtr = strchr(wordbPtr,'\n');
    wordbPtr = strchr(wordbPtr,'@');
    if(wordbPtr == NULL || wordbPtr > linePtr) {
      printf("Error: Can not find address\n");
      packetLength = -1;
    }
    else {
      wordbPtr += 1;
      wordePtr = strchr(wordbPtr,':');
      if(wordePtr == NULL || wordePtr > linePtr) {
        printf("Warning: No port found, using default port\n");
        dat->remotePort = 5062;
        wordePtr = strchr(wordbPtr,'>');
      }
      else {
        dat->remotePort = atoi(wordePtr+1);
        if(dat->remotePort == 0) {
          printf("Error: Port cannot be found\n");
          packetLength = -1;
        }
      }
      wordlen = wordePtr-wordbPtr;
      if(wordlen > 15) {
        printf("Error: Address format is not correct."
            "Must be in XXX.XXX.XXX.XXX notation\n");
        packetLength = -1;
      }
      else {
        strncpy(dat->remoteAddress,wordbPtr,wordlen);
        dat->remoteAddress[wordlen] = 0;
        struct in_addr addr;
        if(!inet_aton(dat->remoteAddress,&addr)) {
          printf("Error: Address format is not correct."
              "Must be in XXX.XXX.XXX.XXX notation\n");
          packetLength = -1;
        }
      }
    }
  }
  wordbPtr = strstr(buffer,"\r\n\r\n");
  if(wordbPtr == NULL)
  {
    wordbPtr = strstr(buffer,"\n\n");
    if(wordbPtr == NULL){
      printf("Error: Can not find the end of the SIP packet!\n");
      packetLength = -1;
    }
    else{
      wordbPtr += 2;
    }
  }
  else{
    wordbPtr += 4;
  }
  if(packetLength != -1) {
    packetLength = wordbPtr-buffer;
  }
  return packetLength;
}
//takes an input msg and populates a buffer with an
//appropriate sip msg
int get_sipmsg_from_data(char *buffer, sip_t *dat, sip_com_t command) {
  switch(command) {
    case REGISTER:
      sprintf(buffer,
        "REGISTER sip:%s SIP/2.0\r\n"
        "Via: SIP/2.0/UDP %s:%d;branch=%s\r\n"
```

```
      "Max-Forwards: 70\r\n"
      "From: sip:%d@%s;tag=%s\r\n"
      "To: sip:%d@%s\r\n"
      "Call-ID: %s\r\n"
      "CSeq: %d REGISTER\r\n"
      "Contact: <sip:%d@%s:%d>\r\n"
      "Expires: 600\r\n"
      "Content-Length: 0\r\n\r\n",
      dat->registrar,
      dat->localAddress,dat->localPort,dat->branchTag,
      dat->localNumber,dat->registrar,dat->localTag,
      dat->localNumber,dat->registrar,
      dat->callID,
      dat->CSeq,
      dat->localNumber,dat->localAddress,dat->localPort);
    break;
  case UNREGISTER:
    sprintf(buffer,
      "REGISTER sip:%s SIP/2.0\r\n"
      "Via: SIP/2.0/UDP %s:%d;branch=%s\r\n"
      "Max-Forwards: 70\r\n"
      "From: sip:%d@%s;tag=%s\r\n"
      "To: sip:%d@%s\r\n"
      "Call-ID:%s\r\n"
      "CSeq: %d REGISTER\r\n"
      "Contact: <sip:%d@%s:%d>\r\n"
      "Expires: 0\r\n"
      "Content-Length: 0\r\n\r\n",
      dat->registrar,
      dat->localAddress,dat->localPort,dat->branchTag,
      dat->localNumber,dat->registrar,dat->localTag,
      dat->localNumber,dat->registrar,
      dat->callID,
      dat->CSeq,
      dat->localNumber,dat->localAddress,dat->localPort);
    break;
  case INVITE:
    sprintf(buffer,
      "INVITE sip:%d@%s SIP/2.0\r\n"
      "Via: SIP/2.0/UDP %s:%d;branch=%s\r\n"
      "Max-Forwards: 70\r\n"
      "From: sip:%d@%s;tag=%s\r\n"
      "To: sip:%d@%s\r\n"
      "Call-ID:%s\r\n"
      "CSeq: %d INVITE\r\n"
      "Contact: <sip:%d@%s:%d>\r\n"
      "Content-Type: application/sdp\r\n"
      "Content-Length: %d\r\n\r\n",
      dat->remoteNumber,dat->registrar,
      dat->localAddress,dat->localPort,dat->branchTag,
      dat->localNumber,dat->registrar,dat->localTag,
      dat->remoteNumber,dat->registrar,
      dat->callID,
      dat->CSeq,
      dat->localNumber,dat->localAddress,dat->localPort,
      dat->contentLength);
    break;
  case ACK:
    sprintf(buffer,
      "ACK sip:%d@%s SIP/2.0\r\n"
      "Via: SIP/2.0/UDP %s:%d;branch=%s\r\n"
      "Max-Forwards: 70\r\n"
      "From: sip:%d@%s;tag=%s\r\n"
      "To: sip:%d@%s;tag=%s\r\n"
      "Call-ID:%s\r\n"
      "CSeq: %d ACK\r\n"
      "Content-Length: 0\r\n\r\n",
      dat->remoteNumber,dat->registrar,
      dat->localAddress,dat->localPort,dat->branchTag,
      dat->localNumber,dat->registrar,dat->localTag,
      dat->remoteNumber,dat->registrar,dat->remoteTag,
```

```
      dat->callID,dat->CSeq);
    break;
  case CANCEL:
    sprintf(buffer,
      "CANCEL sip:%d@%s SIP/2.0\r\n"
      "Via: SIP/2.0/UDP %s:%d;branch=%s\r\n"
      "Max-Forwards: 70\r\n"
      "From: sip:%d@%s;tag=%s\r\n"
      "To: sip:%d@%s;\r\n"
      "Call-ID:%s\r\n"
      "CSeq: %d CANCEL\r\n"
      "Content-Length: 0\r\n\r\n",
      dat->remoteNumber,dat->registrar,
      dat->localAddress,dat->localPort,dat->branchTag,
      dat->localNumber,dat->registrar,dat->localTag,
      dat->remoteNumber,dat->registrar,
      dat->callID,dat->CSeq);
    break;
  case BYE:
    sprintf(buffer,
      "BYE sip:%d@%s SIP/2.0\r\n"
      "Via: SIP/2.0/UDP %s:%d;branch=%s\r\n"
      "Max-Forwards: 70\r\n"
      "From: sip:%d@%s;tag=%s\r\n"
      "To: sip:%d@%s;tag=%s\r\n"
      "Call-ID:%s\r\n"
      "CSeq: %d BYE\r\n"
      "Content-Length: 0\r\n\r\n",
      dat->remoteNumber,dat->registrar,
      dat->localAddress,dat->localPort,dat->branchTag,
      dat->localNumber,dat->registrar,dat->localTag,
      dat->remoteNumber,dat->registrar,dat->remoteTag,
      dat->callID,dat->CSeq);
    break;
  case TRYING:
  case RINGING://fallthrough
  case OK://fallthrough
  case FORBIDDEN://fallthrough
  case NOTFOUND://fallthrough
  case BUSY://fallthrough
  case UNAVAILABLE://fallthrough
  case DECLINE://fallthrough
    sprintf(buffer,
      "SIP/2.0 %s\r\n"
      "Via: SIP/2.0/UDP %s:%d;branch=%s;rport\r\n"
      "From: sip:%d@%s;tag=%s\r\n"
      "To: sip:%d@%s;tag=%s\r\n"
      "Call-ID: %s\r\n"
      "Contact: <sip:%d@%s:%d>\r\n"
      "CSeq: %d %s\r\n",
      sip_com_list[command],
      dat->localAddress,dat->localPort,dat->branchTag,
      dat->remoteNumber,dat->registrar,dat->remoteTag,
      dat->localNumber,dat->registrar,dat->localTag,
      dat->callID,
      dat->localNumber,dat->localAddress,dat->localPort,
      dat->CSeq,dat->dialogOp);
    if(dat->contentLength != 0)
      sprintf(buffer+strlen(buffer),"Content-Type: application/sdp\r\n");
    sprintf(buffer+strlen(buffer),"Content-Length: %d\r\n\r\n",
        dat->contentLength);
    break;
  case UNDECIPHERABLE:
    sprintf(buffer,
      "SIP/2.0 %s\r\n\r\n",
      sip_com_list[command]);
    break;
  case ERROR:
    sprintf(buffer, "");
    break;
}
```

```
  return strlen(buffer);
}



void generate_tag(char *tag, int len) {
  static const char *tag_map ="0123456789abcdefghijklmnopqrstuvwxyz"
                       "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  int i;
  unsigned int randInt;
  for(i = 0; i < len-1; i++) {
    randInt = (unsigned int)rand() % 62;
    tag[i] = tag_map[randInt];
  }
  tag[i] = 0;
}
```

>> con_abstr.h

```
#ifndef CON_ABSTR_H
#define CON_ABSTR_H

#include <string.h>

typedef struct _con_t con_t;

struct _con_t{
  SOCKET sockd;
  struct sockaddr_in src;
  struct sockaddr_in dest;
  char buffer[1024];
  struct timeval tv;
};

int create_connection(con_t *con, char *addr, int localPort, int remotePort);
void set_con_timeout(con_t *con, int sec, int usec);
int send_conbuffer(con_t *con);
int recv_conbuffer(con_t *con);
int close_connection(con_t *con);

#endif
```

>> con.c

```
#include "socket.h"
#include "con_abstr.h"


#define SOCKET int

#undef TRUE
#define TRUE 1
#define FALSE 0


int create_connection(con_t *con, char *addr, int localPort, int remotePort){
  int status;
  status = TRUE;
  if(!inet_aton(addr,&(con->dest.sin_addr))) {
    status = FALSE;
  }
  else{
    con->dest.sin_family = AF_INET;
    con->dest.sin_port = htons(remotePort);
    con->src.sin_addr.s_addr = INADDR_ANY;
    con->src.sin_port = htons(localPort);
```

```c
    //creating the socket
    con->sockd = socket(AF_INET,SOCK_DGRAM,0);
    if(con->sockd != -1)
    {
      if(bind(con->sockd,(struct sockaddr *)&con->src,
          sizeof(struct sockaddr_in)) != -1)
      {
        status = TRUE;
      }
      else
      {
        status = FALSE;
      }
    }
    else
    {
      status = FALSE;
    }
  }
  return status;
}
void set_con_timeout(con_t *con, int sec, int usec) {
    con->tv.tv_sec = sec;
    con->tv.tv_usec = usec;
}


int send_conbuffer(con_t *con)
{
  int bytes_sent;
  bytes_sent = sendto(con->sockd,con->buffer,strlen(con->buffer),0,
                      (struct sockaddr*)&(con->dest),
                      sizeof(struct sockaddr_in));
  return bytes_sent;
}

int recv_conbuffer(con_t *con){
  int bytes_recvd = 0;
  fd_set read;
  FD_ZERO(&read);
  FD_SET(con->sockd, &read);
  //clear buffer
  memset((void*)(con->buffer),0,1024*sizeof(char));
  if(select(con->sockd, &read, NULL, NULL, &con->tv) > 0)
  { //receive 1kb
    //socklen_t fromlen;
    //fromlen = sizeof(struct sockaddr_in);
    bytes_recvd = recvfrom(con->sockd,(void*)con->buffer,1024-1,0,NULL,NULL);
  //null terminate string
    if(bytes_recvd > 0)
      con->buffer[bytes_recvd] = 0;
  }
  return bytes_recvd;
}
int close_connection(con_t *con){
  return close(con->sockd);
}
```

>> defs.h

```c
#ifndef __defs_h__8e9dae76_a46b_49ac_8932_b2bd73e454b8
#define __defs_h__8e9dae76_a46b_49ac_8932_b2bd73e454b8


#ifndef BLEUGH
typedef char int8;
typedef short int16;
typedef int int32;
#endif
```

```
typedef unsigned char uint8;
typedef unsigned short uint16;
typedef unsigned int uint32;

typedef int cv_status;

#define cv_status_success 0
#define cv_status_failure -1

extern const char* get_localIPAddress();

extern cv_status errHook(cv_status status);

#define cv_status_succeeded(s) ((s) == cv_status_success)
#define cv_status_return(s) return errHook(s)
#define cv_status_returnIfFailed(s) if(s) return errHook(s)


#endif /* __defs_h__8e9dae76_a46b_49ac_8932_b2bd73e454b8 */
```

>> main.c

```
#include <stdlib.h>
#include <stdio.h>

#include "includes.h"
#include <alt_iniche_dev.h>
#include "rtp_embedded.h"
#include "cv-lcd.h"
#include "cv-kbd.h"

#define   TASK_STACKSIZE       1024
OS_STK    init_stk[TASK_STACKSIZE];

#define TASK1_PRIORITY 1

extern void voip_main(void* arg);

#define IP4_ADDR(ipaddr, a,b,c,d) ipaddr = \
           htonl(((alt_u32)(a & 0xff) << 24) | ((alt_u32)(b & 0xff) << 16) | \
           ((alt_u32)(c & 0xff) << 8) | (alt_u32)(d & 0xff))


static unsigned char macaddr[6] = { 0x00, 0x07, 0xed, 0xff, 0x06, 0x00 };
static unsigned char ipbyte = 0;

const char* cv_getlocalIPAddress()
{
  static inited = 0;
  static char laddr[16] = {0};

  if(!inited) {
    inited = 1;
    sprintf(laddr, "192.168.1.%d", ipbyte);
  }
  return laddr;
}

int get_ip_addr(alt_iniche_dev              *p_dev,
                ip_addr                     *p_addr,
                ip_addr                     *p_netmask,
                ip_addr                     *p_gw_addr,
                int                         *p_use_dhcp)
{
  /* provide default static setup */
  IP4_ADDR(*p_addr, 192, 168, 1, ipbyte);
  IP4_ADDR(*p_gw_addr, 192, 168, 1, 2);
  IP4_ADDR(*p_netmask, 255, 255, 255, 0);
```

```c
#if 0
  *p_use_dhcp = 1;
#else
  *p_use_dhcp = 0;
#endif

  return 1;
}

int get_mac_addr(NET net, unsigned char mac_addr[6])
{
  int rv = -1;

  /* first 3 bytes are altera's vendor id */
  /* last 3 bytes are picked from serial number sticker */
  mac_addr[0] = macaddr[0];
  mac_addr[1] = macaddr[1];
  mac_addr[2] = macaddr[2];
  mac_addr[3] = macaddr[3];
  mac_addr[4] = macaddr[4];
  mac_addr[5] = macaddr[5];

  /* return the mac address in the array */
  rv = 0;

  return rv;
}

TK_OBJECT(to_ssstask);
TK_ENTRY(voip_main);

struct inet_taskinfo voiptask = {
  &to_ssstask,
  "voip-main",
  voip_main,
  4,
  8192
};


static void init(void* dummy)
{
  cv_status status = cv_status_success;
  cv_kbd kbd;
  cv_lcd lcd;
  int done = 0, pos=0, inp;
  char number[16];

  status = cv_lcd_construct(&lcd);
  status = cv_kbd_construct(&kbd);

  status = cv_lcd_clear(&lcd);
  status = cv_lcd_print(&lcd, "mac byte:");

  while(!done)
  {
    inp = cv_kbd_pollChar(&kbd);
    if(inp != kbdTimeout) {
      if(isAscii(inp)) {
        char ch = mkAscii(inp);
        switch(ch) {
          case '\r':
          case '\n':
            macaddr[5] = (unsigned char) atoi(number);
            done = 1;
            break;
          default:
            if(pos < sizeof(number)) {
              number[pos++] = ch;
              status = cv_lcd_appendChar(&lcd, ch);
```

```
          } else
            printf("ignoring character, > 16\n");
          break;
      }
    }
  }
}

done = 0;
pos = 0;
memset(number, 0, sizeof(number));

status = cv_lcd_clear(&lcd);
status = cv_lcd_print(&lcd, "ipa byte:");

while(!done)
{
  inp = cv_kbd_pollChar(&kbd);
  if(inp != kbdTimeout) {
    if(isAscii(inp)) {
      char ch = mkAscii(inp);
      switch(ch) {
        case '\r':
        case '\n':
          ipbyte = atoi(number);
          done = 1;
          break;
        default:
          if(pos < sizeof(number)) {
            number[pos++] = ch;
            status = cv_lcd_appendChar(&lcd, ch);
          } else
            printf("ignoring character, > 16\n");
          break;
      }
    }
  }
}

done = 0;
status = cv_lcd_clear(&lcd);

sprintf(number, "ip 192.168.1.%d\n", (int)ipbyte);
status = cv_lcd_print(&lcd, number);
sprintf(number, "ma %02x%02x%02x%02x%02x%02x",
        (int)macaddr[0],
        (int)macaddr[1],
        (int)macaddr[2],
        (int)macaddr[3],
        (int)macaddr[4],
        (int)macaddr[5]
       );
status = cv_lcd_print(&lcd, number);

while(!done)
{
  inp = cv_kbd_pollChar(&kbd);
  if(inp != kbdTimeout) {
    if(isAscii(inp)) {
      char ch = mkAscii(inp);
      switch(ch) {
        case '\r':
        case '\n':
          done = 1;
          break;
      }
    }
  }
}

printf("Initializing iniche stack ... \n");
```

```
  alt_iniche_init();
  netmain();
  while (!iniche_net_ready)
    TK_SLEEP(1);

  printf("Iniche stack initialized ... \n");

  TK_NEWTASK(&voiptask);

  printf("Voip server launched ... \n");

  OSTaskDel(OS_PRIO_SELF);
}

int main(void)
{
  printf("Starting ucosii ... \n");
  OSTaskCreateExt(init,
                  NULL,
                  (void *)&init_stk[TASK_STACKSIZE],
                  TASK1_PRIORITY,
                  TASK1_PRIORITY,
                  init_stk,
                  TASK_STACKSIZE,
                  NULL,
                  0);
  OSStart();
}
```

>> sdp_abstr.h

```
#ifndef SDP_ABSTR_H
#define SDP_ABSTR_H

typedef struct _sdp_t sdp_t;

struct _sdp_t{
  unsigned long sessID;
  unsigned long sessVer;
  int port;
  char addr[16];
};

void set_rtp_port(sdp_t *dat, int port);
int set_rtp_addr(sdp_t *dat, const char *addr);

void init_sdp(sdp_t *dat);
//take sdp object and fill buffer with sdp message.
int get_sdpmsg_from_data(char *buffer, sdp_t *dat);
//get buffer and fill sdp object with dat
int get_data_from_sdpmsg(sdp_t *dat, char *buffer);
//get the length of a generated msg with the content
int get_sdpmsg_len(sdp_t *dat);

#endif
```

>> sdp.c

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#include "socket.h"
#include "sdp_abstr.h"
```

```c
#define TRUE 1
#define FALSE 0

void init_sdp(sdp_t *dat) {
  dat->sessID = rand();
  dat->sessVer = dat->sessID;
  dat->port = 0;
  memset(dat->addr,0,16);
}
void set_rtp_port(sdp_t *dat, int port) {
  dat->port = port;
}
int set_rtp_addr(sdp_t *dat, const char *addr) {
  int status;
  struct sockaddr_in testAddr;
  if(strlen(addr) < 15) {
    if(inet_aton(addr,&testAddr.sin_addr)!=0) {
      strcpy(dat->addr,addr);
      status = TRUE;
    }
    else {
      status = FALSE;
    }
  }
  else {
    status = FALSE;
  }
}

int get_data_from_sdpmsg(sdp_t *dat, char *buffer) {
//take sdp object and fill buffer with sdp message.
  int packetLength;
  char *wordbPtr, *wordePtr;
  packetLength = 0;
  wordbPtr = strstr(buffer,"o=");
  if(wordbPtr == NULL) {
    printf("Error, could not find origin tag!\n");
    packetLength = -1;
  }
  else {
    wordbPtr = strpbrk(wordbPtr,"\r\n \0");
    dat->sessID = strtoul(wordbPtr,NULL,10);
    while(isspace(*wordbPtr) != 0)
      wordbPtr++;
    wordbPtr = strpbrk(wordbPtr,"\r\n \0");
    dat->sessVer = strtoul(wordbPtr,NULL,10);
    while(isspace(*wordbPtr) != 0)
      wordbPtr++;
    wordbPtr = strpbrk(wordbPtr,"\r\n \0");
    while(isspace(*wordbPtr) != 0)
      wordbPtr++;
    wordbPtr = strpbrk(wordbPtr,"\r\n \0");
    while(isspace(*wordbPtr) != 0)
      wordbPtr++;
    wordbPtr = strpbrk(wordbPtr,"\r\n \0");
    while(isspace(*wordbPtr) != 0)
      wordbPtr++;
    wordePtr = strpbrk(wordbPtr,"\r\n \0");
    while(isspace(*wordbPtr) != 0)
      wordbPtr++;
    strncpy(dat->addr,wordbPtr,wordePtr-wordbPtr);
    dat->addr[wordePtr-wordbPtr] = 0;
  }
  wordbPtr = strstr(wordbPtr,"m=audio");
  if(wordbPtr == NULL) {
    printf("Could not find media descriptor!\n");
    packetLength = -1;
  }
  else {
    wordbPtr += 7;
```

```
    dat->port = atoi(wordbPtr);
  }
  wordePtr = strstr(wordbPtr,"\r\n\r\n");
  if(wordbPtr == NULL)
  {
    wordePtr = strstr(buffer,"\n\n");
    if(wordePtr == NULL){
      printf("Error: Can not find the end of the SIP packet!\n");
      packetLength = -1;
    }
    else{
      wordePtr += 2;
    }
  }
  else{
    wordePtr += 4;
  }
  if(packetLength != -1) {
    packetLength = wordePtr-buffer;
  }
  return packetLength;
}
//get buffer and fill sdp object with dat
int get_sdpmsg_from_data(char *buffer, sdp_t *dat){
  sprintf(buffer,
      "v=0\r\n"
      "o=- %d %d IN IP4 %s\r\n"
      "s=teamVOIP\r\n"
      "c=IN IP4 %s\r\n"
      "t=0 0\r\n"
      "m=audio %d RTP/AVP 0 101\r\n"
      "a=rtpmap:0 PCMU/8000\r\n"
      "a=sendrecv\r\n"
      "a=fmtp:101 0-15\r\n"
      "a=ptime:20\r\n\r\n",
      dat->sessID,dat->sessVer,dat->addr,
      dat->addr,
      dat->port);
  return strlen(buffer);
}
//get the length of a generated msg with the content
int get_sdpmsg_len(sdp_t *dat) {
  char temp[512];
  get_sdpmsg_from_data(temp,dat);
  return strlen(temp);
}
```

>> socket.h

```
#ifndef __socket_h__52a1796f_cea2_4725_8b08_4b117a26c4cf
#define __socket_h__52a1796f_cea2_4725_8b08_4b117a26c4cf


#include <ipport.h>
#include <osport.h>
#include <tcpport.h>
#include <ip.h>
typedef int SOCKET;



#endif /* __socket_h__52a1796f_cea2_4725_8b08_4b117a26c4cf */
```

>> voip_main.c

```
#include <stdlib.h>
#include <stdio.h>
```

```c
#include "cv-voip.h"

void voip_main(void* arg)
{
  cv_status status = cv_status_success;
  cv_voip voip;

  status = cv_voip_construct(&voip);

  status = cv_voip_start(&voip);


  status = cv_voip_destruct(&voip);
}
```

>> rtp_highlevel.c

```c
/* rtp_unix.c: RTP API types, structures, and functions specific to the
               Windows implementation of the library

Copyright 1997, 1998 Lucent Technologies; all rights reserved
*/
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>

#include "rtp_embedded.h"
#include "rtp_api.h"
#include "rtp_lowlevel.h"
#include "rtp_highlevel.h"
#include "rtp_highlevel_internal.h"
#include "sysdep.h"

#include "global.h"      /* from RFC 1321 */
#include "md5.h"         /* from RFC 1321 */


extern int IsMulticast(struct in_addr addr);
static void hl_changed_sockaddr_callback(context cid,
                                         person p,
                                         struct sockaddr *sa,
                                         int is_rtcp);


/* High-level API functions */
rtperror RTPCreate(context *the_context)
{
  rtperror err;

  hl_context *uc;

  uc = (hl_context *) malloc(sizeof(hl_context));
  if (uc == NULL)
    return errordebug(RTP_CANT_ALLOC_MEM, "RTPCreate",
                     "out of memory\n");

  err = RTPLowLevelCreate(the_context);
  if (err != RTP_OK)
    goto bailout;

  err = RTPSessionSetHighLevelInfo(*the_context, (void*)uc);
  if (err != RTP_OK)
    goto bailout;

  uc->connection_opened = FALSE;
```

```c
  uc->send_addr_list = NULL;
  uc->recv_addr_list = NULL;

  uc->rtp_sourceaddr.sin_family = AF_UNSPEC;
  uc->rtcp_sourceaddr.sin_family = AF_UNSPEC;

  uc->use_encryption = _RTP_DEFAULT_ENCRYPTION;
  uc->key = NULL;

  uc->encrypt_initfunc = NULL;
  uc->encrypt_encryptfunc = NULL;
  uc->encrypt_decryptfunc = NULL;

  uc->PreventEntryIntoFlaggingFunctions = FALSE;
  uc->SendErrorCallBack = NULL;
  uc->ChangedMemberAddressCallBack = NULL;

  err = RTPSetChangedMemberSockaddrCallBack(*the_context,
                                        &hl_changed_sockaddr_callback);
  if (err != RTP_OK)
    goto bailout;

  return RTP_OK;

bailout:
  free(uc);
  return err;
}


rtperror RTPDestroy(context cid)
{
  rtperror err;
  address_holder_t *s, *t;

  hl_context *uc;

  err = RTPSessionGetHighLevelInfo(cid, (void **)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;

  /* Context exists.  Now check if connection is open.  If so,
     close it. */
  if (uc->connection_opened){
    err = RTPCloseConnection(cid, NULL);
    if (err != RTP_OK){
      return err;
    }
  }

  /* Remove the receiver list (if it exists) */
  if (uc->recv_addr_list != NULL) {
    free(uc->recv_addr_list);
  }

  /* Remove the sender list (if it exists) */
  s = uc->send_addr_list;
  while (s != NULL) {
    t = s->next;
    free(s);
    s = t;
  }

  free(uc);
  err = RTPLowLevelDestroy(cid);
  return err;
}

/* This function adds a destination for sending packets. They can be
   either unicast or multicast. TTL has no meaning for unicast, and
```

```
   may be given as any value. The library will set it to zero before storing
   it anyway. The port is in host byte order.

   The function also creates and connects the sockets for RTP and RTCP.
   If multicast, it also sets the ttl. The resulting sockets are stored
   in the context. When it's time to send, the send function can be
   used directly.

   You should not call this function with zero port number. Send port numbers
   should never be dynamic. */

rtperror RTPSessionAddSendAddr(context cid, char *addr, u_int16 port, u_int8 ttl){
  address_holder_t *holder;
  struct sockaddr_in saddr;
  int len, nRet;
  struct in_addr translation;
  hl_context *uc;
  rtperror err;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;

  if (port == 0) {
    return errordebug(RTP_BAD_PORT, "RTPSessionAddSendAddr",
                      "Port number zero not allowed");
  }
  /* If the port is odd, assume it's the RTCP port */

  if((port & 1) == 1)
    port--;

  if((holder = (address_holder_t *) malloc(sizeof(address_holder_t))) == 0) {
    return errordebug(RTP_CANT_ALLOC_MEM, "RTPSessionAddSendAddr",
                      "Cannot allocate memory");
  }


  /* Translate address */
  translation = host2ip(addr);
  if(translation.s_addr == (u_int32) -1) {
    free(holder);
    return errordebug(RTP_BAD_ADDR, "RTPSessionAddSendAddr",
                      "Could not resolve address");
  }


  /* Write values of address, port to context */
  holder->address = translation;
  holder->port = htons(port);
  holder->deleteflag = FALSE;
  holder->ttl = 0;
  if(IsMulticast(translation)) holder->ttl = ttl;


  /* Create the RTP and RTCP sockets for this sender */

  holder->rtpsocket = _sys_create_socket(SOCK_DGRAM);

  if (holder->rtpsocket == _SYS_INVALID_SOCKET){
    free(holder);
    return errordebug(RTP_CANT_GET_SOCKET, "RTPSessionAddSendAddr",
                      "couldn't get RTP socket for context %d",  (int)cid);
  }

  holder->rtcpsocket = _sys_create_socket(SOCK_DGRAM);
  if (holder->rtcpsocket == _SYS_INVALID_SOCKET){
    _sys_close_socket(holder->rtpsocket);
    free(holder);
    return errordebug(RTP_CANT_GET_SOCKET, "RTPSessionAddSendAddr",
```

```
                  "couldn't get RTCP socket for context %d",  (int)cid);
}

/* Connect them, first RTP socket */

memset(&saddr, 0, sizeof(saddr));
saddr.sin_family = AF_INET;
saddr.sin_addr = holder->address;
saddr.sin_port = htons(port);

if(_sys_connect_socket(holder->rtpsocket, &saddr) == _SYS_SOCKET_ERROR) {
  err = errordebug(RTP_CANT_GET_SOCKET, "RTPSessionAddSendAddr",
                  "couldn't connect RTP socket for context %d",  (int)cid);
  goto bailout;
}

/* Now RTCP socket */
saddr.sin_port = htons(port+1);

if(_sys_connect_socket(holder->rtcpsocket, &saddr) == _SYS_SOCKET_ERROR) {
  err = errordebug(RTP_CANT_GET_SOCKET, "RTPSessionAddSendAddr",
                   "couldn't connect RTCP socket for context %d",  (int)cid);
  goto bailout;
}

if(IsMulticast(holder->address)) {
  /* Set multicast TTL if needed */
        nRet = _sys_set_ttl(holder->rtpsocket, ttl);
        if(nRet == _SYS_SOCKET_ERROR) {
                err = errordebug(RTP_CANT_SET_SOCKOPT, "RTPSessionAddSendAddr",
                        "couldn't set RTP TTL for context %d",  (int)cid);
                goto bailout;
        }
        nRet = _sys_set_ttl(holder->rtcpsocket, ttl);

        if(nRet == _SYS_SOCKET_ERROR) {
                err = errordebug(RTP_CANT_SET_SOCKOPT, "RTPSessionAddSendAddr",
                        "couldn't set RTCP TTL for context %d",  (int)cid);
                goto bailout;
        }



        /* Determine source addresses, for loopback detection */
        /* XXX: multiple multicast destinations might have different sources */

        len = sizeof(struct sockaddr_in);

        if(_sys_get_socket_name(holder->rtpsocket, &uc->rtp_sourceaddr) == _SYS_SOCKET_ERROR) {
                err = errordebug(RTP_CANT_GET_SOCKET, "RTPSessionAddSendAddr",
                        "Couldn't get RTP source address for context %d", (int)cid);
                goto bailout;
        }
        if(_sys_get_socket_name(holder->rtcpsocket, &uc->rtcp_sourceaddr) == _SYS_SOCKET_ERROR)
{
                err = errordebug(RTP_CANT_GET_SOCKET, "RTPSessionAddSendAddr",
                        "Couldn't get RTCP source address for context %d", (int)cid);
                goto bailout;
        }
}

/* Add address to list */

holder->next = uc->send_addr_list;
uc->send_addr_list = holder;

return RTP_OK;

bailout:

_sys_close_socket(holder->rtpsocket);
```

```
  _sys_close_socket(holder->rtcpsocket);
  free(holder);

  return err;
}

/* This function removes addresses from the send list. The port is in host
   byte order. The address, port, and ttl must match exactly in order to
   remove the element from the list.

   The function will also close the associated sockets. */

rtperror RTPSessionRemoveSendAddr(context cid, char *addr, u_int16 port, u_int8 ttl) {
  address_holder_t *holder;
  struct in_addr translation;
  hl_context *uc;
  rtperror err;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;

  holder = uc->send_addr_list;

  /* If the port is odd, assume it's the RTCP port */

  if((port & 1) == 1)
    port--;

  translation = host2ip(addr);
  if(translation.s_addr == (u_int32) -1) {
    return errordebug(RTP_BAD_ADDR, "RTPSessionRemoveSendAddr",
                      "Could not resolve address");
  }


  /* TTL matching is only done for multicast. For unicast, all TTL's
     are set to zero */

  if(!IsMulticast(translation)) ttl = 0;

  while(holder != NULL) {
    if(!(holder->deleteflag) &&
       (holder->address.s_addr == translation.s_addr) &&
       (holder->port == htons(port)) &&
       (holder->ttl == ttl))
      break;

    holder = holder->next;
  }

  /* Now holder is either NULL if there was no match, else it points
     to the address which matched */

  if(holder == NULL) {
    return errordebug(RTP_BAD_ADDR, "RTPSessionRemoveSendAddr",
                      "No such address");
  } else {
    holder->deleteflag = TRUE;
    return RTP_OK;
  }
}

/* This function sets the address and port that the library listens to
   for incoming packets. Currently, you can only listen to a single
   socket. For unicast operation, setting the address to NULL will
   cause the library to use INADDR_ANY to bind to. Setting the port
   to zero will cause the library to obtain a dynamic port number to
   listen to for RTP. The RTCP port will then be bound to the port one
   higher than this. Once the socket has been created and opened (as a
```

```
   result of calling RTPOpenConnection, you can use
   RTPSessionGetReceiveAddr to read the port number that was actually
   used. For multicast, the address is the multicast group to listen
   to.

   Listening to a multicast address should? get you the unicast
   packets destined for the same port.

   The port is in host byte order.

   You cannot call this function once OpenConnection has been called.
   */

rtperror RTPSessionSetReceiveAddr(context cid, char *address, u_int16 port){
  address_holder_t *holder;
  struct in_addr translation;
  hl_context *uc;
  rtperror err;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;

  if (uc->connection_opened){
    return errordebug(RTP_FIXED_WHEN_OPEN, "RTPSessionSetLocalAddr",
                      "Cannot change address during opened connection");
  }

  /* If the port is odd, assume it's the RTCP port */

  if((port & 1) == 1)
    port--;

  translation = host2ip(address);
  if(translation.s_addr == (u_int32) -1) {
    return errordebug(RTP_BAD_ADDR, "RTPSessionSetReceiveAddr",
                      "Could not resolve address");
  }

  if(uc->recv_addr_list == NULL) {

    /* Create new address structure */
    if((holder = (address_holder_t *) malloc(sizeof(address_holder_t))) == 0) {
      return errordebug(RTP_CANT_ALLOC_MEM, "RTPSessionAddSendAddr",
                      "Cannot allocate memory");
    }

    holder->address = translation;
    if (address == NULL) holder->address.s_addr = 0;
    holder->port = htons(port);
    holder->ttl = 0;

    uc->recv_addr_list = holder;
  } else {

    /* Modify existing values */
    holder = uc->recv_addr_list;
    holder->address = translation;
    if(address == NULL) holder->address.s_addr = 0;
    holder->port = htons(port);
    holder->ttl = 0;
  }


  return RTP_OK;
}

/* This function returns the receive address and port number, in host
   order. They must have been previously set with RTPSessionSetReceiveAddr
   in order for this to work. */
```

```c
rtperror RTPSessionGetReceiveAddr(context cid, char *addr, u_int16 *port){
  hl_context *uc;
  rtperror err;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;

  if(uc->recv_addr_list == NULL) {
    return errordebug(RTP_BAD_ADDR, "RTPSessionGetReceiveAddr",
                      "Address not yet set");
  }

  strcpy(addr, inet_ntoa(uc->recv_addr_list->address));
  *port = ntohs(uc->recv_addr_list->port);

  return RTP_OK;
}


rtperror RTPOpenConnection(context cid){
  struct sockaddr_in saddr;
  int dynamic_ports,bind_count,problem, nRet;
  socktype rtpskt, rtcpskt;
  hl_context *uc;
  rtperror err;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;

  if (uc->connection_opened){
    RTPCloseConnection(cid,NULL);
  }

  /* First check if the user has set the local address */
  if(uc->recv_addr_list == NULL) {
    return errordebug(RTP_BAD_ADDR, "RTPOpenConnection",
                      "Address not yet set");
  }

  /* Set a flag for dynamic ports */

  if(uc->recv_addr_list->port == 0)
    dynamic_ports = 1;
  else
    dynamic_ports = 0;

  /* For dynamic ports, we choose a port randomly, and try
     and bind to it, plus the one one higher for RTCP. If either
     fail, we iterate _BIND_COUNTER times, and then give up */

  bind_count = 0;
  while(bind_count < _BIND_COUNTER) {
    bind_count++;

    /* We only use ports in the dynamic range - 49152 - 65535 */

    if(dynamic_ports == 1)
      uc->recv_addr_list->port =
       htons(_UDP_PORT_BASE + 2 * ((u_int16) (drand48() * _UDP_PORT_RANGE)));


    /* Create the RTP and RTCP sockets */
    uc->recv_addr_list->rtpsocket = _sys_create_socket(SOCK_DGRAM);
    rtpskt = uc->recv_addr_list->rtpsocket;
    if (uc->recv_addr_list->rtpsocket == _SYS_INVALID_SOCKET){
      return errordebug(RTP_CANT_GET_SOCKET, "RTPOpenConnection",
```

```
                   "couldn't get RTP socket for context %d",  (int)cid);
}
uc->recv_addr_list->rtcpsocket = _sys_create_socket(SOCK_DGRAM);
rtcpskt = uc->recv_addr_list->rtcpsocket;
if (uc->recv_addr_list->rtcpsocket == _SYS_INVALID_SOCKET){
  _sys_close_socket(rtpskt);
  return errordebug(RTP_CANT_GET_SOCKET, "RTPOpenConnection",
                   "couldn't get RTCP socket for context %d",  (int)cid);
}

/* bind sockets */

memset(&saddr, 0, sizeof(saddr));
saddr.sin_family = AF_INET;
saddr.sin_addr = uc->recv_addr_list->address;
saddr.sin_port = uc->recv_addr_list->port;

/* If the address is multicast or null, bind to INADDR_ANY */

if((uc->recv_addr_list->address.s_addr == 0) ||
   IsMulticast(saddr.sin_addr))
  saddr.sin_addr.s_addr = INADDR_ANY;

/* RTP port bind */
problem = 0;
if((problem = _sys_bind(rtpskt, &saddr)) == _SYS_SOCKET_ADDRNOTAVAIL) {
             saddr.sin_addr.s_addr = INADDR_ANY;
             problem = _sys_bind(rtpskt, &saddr);
}

/* Address in use, try another port if we're doing dynamic ports */
if((problem == _SYS_SOCKET_ADDRINUSE) &&
   (dynamic_ports == 1)) {
  _sys_close_socket(rtpskt);
  _sys_close_socket(rtcpskt);
  continue;
} else if(problem != 0) {
  printf("RTP_CANT_BIND_SOCKET0: %d\n",problem);
  return errordebug(RTP_CANT_BIND_SOCKET, "RTPOpenConnection",
                             "couldn't bind RTP address for context %d",  (int)cid);
}

/* No error! */

saddr.sin_port = htons(ntohs(uc->recv_addr_list->port) + 1);

/* Bind to RTCP port */

problem = 0;
if((problem = _sys_bind(rtcpskt, &saddr)) == _SYS_SOCKET_ADDRNOTAVAIL) {
  /* The user specified a nonlocal address - probably they want to
     send to this as a unicast address, so try INADDR_ANY */
  saddr.sin_addr.s_addr = INADDR_ANY;
  problem = _sys_bind(rtpskt, &saddr);
}

/* Address in use, try another port if we're doing dynamic ports */

if((problem == _SYS_SOCKET_ADDRINUSE) &&
   (dynamic_ports == 1)) {
  _sys_close_socket(rtpskt);
  _sys_close_socket(rtcpskt);
  continue;
} else if(problem != 0) {
  printf("RTP_CANT_BIND_SOCKET1: %d\n",problem);
  return errordebug(RTP_CANT_BIND_SOCKET, "RTPOpenConnection",
                   "couldn't bind RTCP address for context %d", (int)cid);
}

break;
```

```
  }

  /* Now we are here either because of success, or looping
     too much */

  if(bind_count == _BIND_COUNTER) {
    return errordebug(RTP_CANT_BIND_SOCKET, "RTPOpenConnection",
                      "couldn't bind dynamic address for context %d", (int)cid);
  }


  /* Allow reuse of the address and port  */

  if (IsMulticast(uc->recv_addr_list->address)){ /* Multicast */
#if 0
    /* Every member of the session is a member of the multicast session */
          memset(&saddr, 0, sizeof(saddr));
          saddr.sin_family = AF_INET;
          saddr.sin_addr = uc->recv_addr_list->address;
          saddr.sin_port = uc->recv_addr_list->port;
          nRet = _SYS_SOCKET_ERROR; //_sys_join_mcast_group(rtpskt, &saddr);
          if(nRet == _SYS_SOCKET_ERROR) {
                  _sys_close_socket(rtpskt);
                  _sys_close_socket(rtcpskt);
                  return errordebug(RTP_CANT_SET_SOCKOPT, "RTPOpenConnection",
                          "couldn't join RTP multicast group for context %d",  (int)cid);
          }
          nRet = _sys_join_mcast_group(rtcpskt, &saddr);
          if(nRet == _SYS_SOCKET_ERROR) {
                  _sys_close_socket(rtpskt);
                  _sys_close_socket(rtcpskt);
                  return errordebug(RTP_CANT_SET_SOCKOPT, "RTPOpenConnection",
                          "couldn't join RTCP multicast group for context %d",  (int)cid);
          }
#endif
  }

  /* Schedule the first rtcp packet, and initialize some data structures */
  err = RTPStartSession(cid);
  if (err != RTP_OK)
    return err;

  uc->connection_opened = TRUE;

  return RTP_OK;
}


rtperror RTPCloseConnection(context cid, char *reason){
  address_holder_t *s;
  hl_context *uc;
  rtperror err;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;

  if (!uc->connection_opened){
    return RTP_OK; /* Connection is already closed */
  }

  err = RTPStopSession(cid, reason);
  if (err != RTP_OK)
    return err;


  if (_sys_close_socket(uc->recv_addr_list->rtpsocket) < 0){
    return errordebug(RTP_CANT_CLOSE_SESSION, "RTPCloseConnection",
                      "context %d couldn't close RTP session", (int)cid);
  }
```

```
  if (_sys_close_socket(uc->recv_addr_list->rtcpsocket) < 0){
    return errordebug(RTP_CANT_CLOSE_SESSION, "RTPCloseConnection",
                      "context %d couldn't close RTCP session", (int)cid);
  }

  s = uc->send_addr_list;
  while(s != NULL) {
    if (_sys_close_socket(s->rtpsocket) < 0){
      return errordebug(RTP_CANT_CLOSE_SESSION, "RTPCloseConnection",
                        "context %d couldn't close RTP session", (int)cid);
    }
    if (_sys_close_socket(s->rtcpsocket) < 0){
      return errordebug(RTP_CANT_CLOSE_SESSION, "RTPCloseConnection",
                        "context %d couldn't close RTCP session", (int)cid);
    }
    s = s->next;
  }

  uc->connection_opened = FALSE;
  return(err);
}

rtperror RTPSessionGetRTPSendSocket(context cid, socktype *value){
  rtperror err;
  hl_context *uc;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;

  if (!uc->connection_opened){
    return errordebug(RTP_NOSOCKET, "RTPSessionGetRTPSendSocket",
                      "context %d, connection not yet opened.", (int)cid);
  }
  *value = uc->send_addr_list->rtpsocket;
  return RTP_OK;
}

rtperror RTPSessionGetRTPSocket(context cid, socktype *value){
  rtperror err;
  hl_context *uc;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;

  if (!uc->connection_opened){
    return errordebug(RTP_NOSOCKET, "RTPSessionGetRTPSocket",
                      "context %d, connection not yet opened.", (int)cid);
  }
  *value = uc->recv_addr_list->rtpsocket;
  return RTP_OK;
}

rtperror RTPSessionGetRTCPSocket(context cid, socktype *value){
  rtperror err;
  hl_context *uc;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;

  if (!uc->connection_opened){
    return errordebug(RTP_NOSOCKET, "RTPSessionGetRTCPSocket",
                      "context %d, connection not yet opened.", (int)cid);
  }
  *value = uc->recv_addr_list->rtcpsocket;
  return RTP_OK;
```

```
}



rtperror RTPSend(context cid, int32 tsinc, int8 marker,
                 int16 pti, int8 *payload, int len)
{
  int buflen=12, rundelete;
  address_holder_t *s, *prevs;
  hl_context *uc;
  rtperror err, errchk;
  u_int8* bptr;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    return err; /* The cid is bogus */

  if (uc->PreventEntryIntoFlaggingFunctions){
    return errordebug(RTP_CANT_CALL_FUNCTION, "RTPSendVector",
                      "context %d, cannot be called now",
                      (int)cid);
  }

  if(uc->send_addr_list == NULL) {
    return errordebug(RTP_BAD_ADDR, "RTPSendVector",
                      "context %d, no send addresses",
                      (int)cid);
  }

  uc->PreventEntryIntoFlaggingFunctions = TRUE;

  err = RTPBuildRTPHeader(cid, tsinc, marker, pti, FALSE,
                          len, payload, &buflen);
  if (err != RTP_OK)
    return err; /* The cid is bogus */
  len += buflen;

  s = uc->send_addr_list;

  err = RTP_OK;
  rundelete = FALSE;
  while(s != NULL)
  {
    if(s->deleteflag == FALSE)
    {
      errchk = sendto(s->rtpsocket, payload, len, 0, NULL, 0);
      if (errchk < 0)
      {
        printf("fuck!\n");
        err = errordebug(RTP_SOCKET_WRITE_FAILURE, "RTPSendVector",
                         "context %d could not write to RTP socket",
                         (int)cid);
        if (uc->SendErrorCallBack != NULL) {
          uc->SendErrorCallBack(cid,
                                inet_ntoa(s->address),
                                ntohs(s->port),
                                s->ttl);
        }
        if (s->deleteflag == TRUE) {
          rundelete = TRUE;
        }
      }
    } else {
      rundelete = TRUE;
    }
    //s = s->next;
    s = NULL;
  }

  /* Now, we clean up the send list and remove all that have been deleted.
     We know that this needs to be done if rundelete is TRUE */
```

```
  prevs = NULL;
  if(rundelete == TRUE)
  {
    s = uc->send_addr_list;
    while(s != NULL)
    {
      if(s->deleteflag == TRUE)
      {
        if(prevs == NULL)
          uc->send_addr_list = s->next;
        else
          prevs->next = s->next;

        _sys_close_socket(s->rtpsocket);
        _sys_close_socket(s->rtcpsocket);
        free(s);
      }
      prevs = s;
      s = s->next;
    }
  }

  uc->PreventEntryIntoFlaggingFunctions = FALSE;
  return(err);
}

rtperror RTPReceive(context cid, socktype socket,
                    char *rtp_pkt_stream, int *len)
{
  int read_len, tot_len;
  struct sockaddr from_addr;
  int fromaddrlen;
  struct sockaddr_in *check_addr, *from_addr_in;
  int isRTCP, possible_loopback;

  rtperror err;
  hl_context *uc;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;

  if (uc->PreventEntryIntoFlaggingFunctions){
    return errordebug(RTP_CANT_CALL_FUNCTION, "RTPReceive",
                      "context %d, cannot be called now",
                      (int)cid);
  }
  if (socket != uc->recv_addr_list->rtpsocket &&
      socket != uc->recv_addr_list->rtcpsocket){
    return errordebug(RTP_SOCKET_MISMATCH, "RTPReceive",
                     "context %d, socket provided not RTP socket nor RTCP socket", (int)cid);
  }
  uc->PreventEntryIntoFlaggingFunctions = TRUE;

  fromaddrlen = sizeof(from_addr);
  read_len = _sys_recvfrom(socket, rtp_pkt_stream, *len, 0, &from_addr, &fromaddrlen);
  if (read_len == -1) {
    return errordebug(RTP_SOCKET_READ_FAILURE, "RTPReceive",
                     "Could not read from socket %d", socket);
  }

  if (read_len == *len){
    /* If we get here, then the buffer was not large enough to hold
       the whole packet. */
    tot_len = read_len;
    /* Keep reading until we drain the buffer */
    while (read_len == *len){
      read_len = _sys_recvfrom(socket, rtp_pkt_stream, *len, 0, &from_addr, &fromaddrlen);
      tot_len += read_len;
    }
```

```
    *len = tot_len;

    uc->PreventEntryIntoFlaggingFunctions = FALSE;
    return errordebug(RTP_INSUFFICIENT_BUFFER, "RTPReceive",
                     "context %d, insufficient buffer provided to hold packet", (int)cid);
  }
  *len = read_len;

  /* XXX encryption: decrypt here */

  isRTCP = (socket == uc->recv_addr_list->rtcpsocket);

  /* If our fromaddr agrees with the appropriate source addr, mark this as a
     possible loopback to RTPPacketReceived.

     On sensible systems where getsockname() does the right thing for
     connected UDP sockets, we check if the addr and port match.
     Unfortunately, on some systems (Solaris and Windows) getsockname() puts
     INADDR_ANY for the addr; there, we can only check if the ports
     match. */

  if (isRTCP) check_addr = (struct sockaddr_in *)&uc->rtcp_sourceaddr;
  else        check_addr = (struct sockaddr_in *)&uc->rtp_sourceaddr;

  from_addr_in = (struct sockaddr_in *)&from_addr;

  possible_loopback =
    (check_addr->sin_family != AF_UNSPEC &&
     (check_addr->sin_addr.s_addr == from_addr_in->sin_addr.s_addr ||
      check_addr->sin_addr.s_addr == INADDR_ANY) /* Solaris, Winsock */ &&
     from_addr_in->sin_port == check_addr->sin_port);

  err = RTPPacketReceived(cid, rtp_pkt_stream, read_len,
                          from_addr, fromaddrlen,
                          isRTCP, possible_loopback);

  uc->PreventEntryIntoFlaggingFunctions = FALSE;
  return err;
}

rtperror RTPSessionRemoveFromContributorList(context cid, u_int32 ssrc,
                                             char *reason)
{
  rtperror err;
  hl_context *uc;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;

  err = RTPSessionLowLevelRemoveFromContributorList(cid, ssrc, reason);

  return err;
}

/* This is for the low-level code -- interface with RTPSchedule */

struct timer_info {
  context cid;
  int32 timer_type;
  u_int32 data;
  char *reason;
};

void RTPSetTimer(context cid, int32 timer_type, u_int32 data,
                char *str, struct timeval *tp)
{
  rtperror err;
  hl_context *uc;
  struct timer_info *ti;
```

```
  struct timeval now;
  struct timeval notime = {0,0};

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return;

  /* Build a timer_info */
  ti = (struct timer_info *) malloc(sizeof(struct timer_info));
  if (ti == NULL)
    /* XXX log debug info? */
    return;

  ti->cid = cid;
  ti->timer_type = timer_type;
  ti->data = data;
  if (str != NULL)
    ti->reason = strdup(str);
  else
    ti->reason = NULL;

  /* If the time-to-send isn't in the future, and if we're not in a callback,
     send immediately */
  gettimeofday(&now, NULL);
  if (!uc->PreventEntryIntoFlaggingFunctions &&
      TimeExpired(tp, &now, &notime)) {
    RTPExecute(cid, (rtp_opaque_t) ti);
  }
  else {
    RTPSchedule(cid, (rtp_opaque_t) ti, tp);
  }
}


rtperror RTPExecute(context cid, rtp_opaque_t opaque){
  rtperror err;
  hl_context *uc;
  struct timer_info *ti;
  int errchk, rundelete, buflen;
  address_holder_t *s, *prevs;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;

  ti = (struct timer_info *) opaque;

  if (ti->cid != cid) {
    return errordebug(RTP_UNKNOWN_CONTEXT, "RTPExecute",
                      "context %d in arg != context %d in opaque",
                      (int)cid, (int)ti->cid);
  }

  if (uc->PreventEntryIntoFlaggingFunctions){
    return errordebug(RTP_CANT_CALL_FUNCTION, "RTPExecute",
                      "context %d, cannot be called now",
                      (int)cid);
  }
  uc->PreventEntryIntoFlaggingFunctions = TRUE;
  /* XXX encryption: block sizes, sub-parts */
  buflen = _RTP_MAX_PKT_SIZE;
  switch(ti->timer_type) {
  case RTP_TIMER_SEND_RTCP:
    err = RTPBuildRTCPPacket(cid, RTCP_SUBPARTS_ALL, 0,
                                              uc->packet_buffer, &buflen);
    break;
  case RTP_TIMER_SEND_BYE_ALL:
    err = RTPBuildByePacket(cid, FALSE, 0, ti->reason, 0,
                                             uc->packet_buffer, &buflen);
```

```
    break;
  case RTP_TIMER_SEND_BYE_CONTRIBUTOR:
  case RTP_TIMER_SEND_BYE_COLLISION:
    err = RTPBuildByePacket(cid, TRUE, ti->data, ti->reason, 0,
                                          uc->packet_buffer, &buflen);
    break;
  default:
    goto cleanup;
  }

  if (err != RTP_OK) {
    if (err == RTP_DONT_SEND_NOW) {
      /* This is a legitimate "error" message and should be suppressed */
      /* XXX: should any other return codes be suppressed? */
      err = RTP_OK;
    }
    goto cleanup;
  }

  /* XXX encryption: encrypt here */

  s = uc->send_addr_list;

  err = RTP_OK;
  rundelete = FALSE;
  while(s != NULL) {
    if(s->deleteflag == FALSE) {
      errchk = _sys_send(s->rtcpsocket, uc->packet_buffer, buflen, 0);
      if (errchk < 0){
        if (uc->SendErrorCallBack != NULL) {
          uc->SendErrorCallBack(cid,
                                inet_ntoa(s->address),
                                ntohs(s->port),
                                s->ttl);
        }
        if (s->deleteflag == TRUE) {
          rundelete = TRUE;
        }
      }
    }
    else {
      rundelete = TRUE;
    }
    s = s->next;
  }

  /* Now, we clean up the send list and remove all that have been deleted.
     We know that this needs to be done if rundelete is TRUE */

  prevs = NULL;
  if (rundelete == TRUE) {
    s = uc->send_addr_list;

    while(s != NULL) {
      if(s->deleteflag == TRUE) {
        if(prevs == NULL)
          uc->send_addr_list = s->next;
        else
          prevs->next = s->next;

        _sys_close_socket(s->rtpsocket);
        _sys_close_socket(s->rtcpsocket);
        free(s);
      }
      prevs = s;
      s = s->next;
    }
  }

cleanup:
  if (ti->reason != NULL) {
```

```
    free(ti->reason);
  }
  free(ti);
  uc->PreventEntryIntoFlaggingFunctions = FALSE;
  return err;
}

rtperror RTPSessionSetEncryption(context cid, encryption_t value){
  rtperror err;
  hl_context *uc;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;

  uc->use_encryption = value;
  return RTP_OK;
}

rtperror RTPSessionGetEncryption(context cid, encryption_t *value){

  rtperror err;
  hl_context *uc;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;

  *value =  uc->use_encryption;
  return RTP_OK;
}


rtperror RTPSessionSetEncryptionFuncs(context cid,
                                    void (*init)(context, void*),
                                    void (*encrypt)(context, char*,
                                                    int, void*),
                                    void (*decrypt)(context, char*,
                                                    int, void*)){
  rtperror err;
  hl_context *uc;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;

  uc->encrypt_initfunc = init;
  uc->encrypt_encryptfunc = encrypt;
  uc->encrypt_decryptfunc = decrypt;
  return RTP_OK;
}

rtperror RTPSessionSetKey(context cid, void* value){
  rtperror err;
  hl_context *uc;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;


  uc->key = value;
  if (uc->encrypt_initfunc != NULL){
    uc->encrypt_initfunc(cid, value);
  }
  return RTP_OK;
}
```

```
rtperror RTPSessionGetKey(context cid, void **value){
  rtperror err;
  hl_context *uc;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;

  *value = uc->key;
  return RTP_OK;
}


rtperror RTPSetSendErrorCallBack(context cid,
       void (*f)(context, char *, u_int16, u_int8)) {
  rtperror err;
  hl_context *uc;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;

  uc->SendErrorCallBack = f;
  return RTP_OK;
}

static void hl_changed_sockaddr_callback(context cid,
                                         person p,
                                         struct sockaddr *sa,
                                         int is_rtcp)
{
  rtperror err;
  hl_context *uc;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return;

  if (uc->ChangedMemberAddressCallBack != NULL) {
    char portstr[_RTP_MAX_PORT_STR_SIZE];
    struct sockaddr_in *si = (struct sockaddr_in *) sa;

    sprintf(portstr, "%hu", ntohs(si->sin_port));
    uc->ChangedMemberAddressCallBack(cid, p,
                               inet_ntoa(si->sin_addr),
                               portstr,
                               is_rtcp);
  }
}

rtperror RTPSetChangedMemberAddressCallBack(context cid,
       void (*f)(context, person, char*, char*, int))
{
  rtperror err;
  hl_context *uc;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;

  uc->ChangedMemberAddressCallBack = f;
  return RTP_OK;
}

rtperror RTPMostRecentAddr(context cid, char *addr, char *port) {
  struct sockaddr_in si;
```

```
  rtperror err;

  err = RTPMostRecentSockaddr(cid, (struct sockaddr *)&si);
  if (err != RTP_OK)
    return err;

  strcpy(addr, inet_ntoa(si.sin_addr));
  sprintf(port, "%hu", ntohs(si.sin_port));

  return RTP_OK;
}


rtperror RTPMemberInfoGetRTPAddr(context cid, person p,
                                char *addr, char *port) {
  struct sockaddr_in si;
  rtperror err;

  err = RTPMemberInfoGetRTPSockaddr(cid, p, (struct sockaddr *)&si);
  if (err != RTP_OK)
    return err;

  strcpy(addr, inet_ntoa(si.sin_addr));
  sprintf(port, "%hu", ntohs(si.sin_port));

  return RTP_OK;
}

rtperror RTPMemberInfoGetRTCPAddr(context cid, person p,
                                 char *addr, char *port) {
  struct sockaddr_in si;
  rtperror err;

  err = RTPMemberInfoGetRTCPSockaddr(cid, p, (struct sockaddr *)&si);
  if (err != RTP_OK)
    return err;

  strcpy(addr, inet_ntoa(si.sin_addr));
  sprintf(port, "%hu", ntohs(si.sin_port));

  return RTP_OK;
}



/* Initialize random number generators with a random seed. */
/* Compile with -D_RTP_SEMI_RANDOM to get repeatable behavior
 * for testing.
 */

void InitRandom(){
  struct timeval curtime;
#ifdef _RTP_SEMI_RANDOM
  return;  /* shuts off seeding random generators */
#endif
  gettimeofday(&curtime, NULL);
  srand48((int) curtime.tv_usec);
}

/* random32: generate a 32-bit random number.
 * Without _RTP_SEMI_RANDOM, this is (hopefully) a cryptographically-secure
 * hash of non-externally-predictable values.
 */

/***************************
 * Random # generator code *
 * Straight from RFC 1889  *
 ***************************/

/*
 * Generate a random 32-bit quantity.
```

```
 */

#define MD_CTX MD5_CTX
#define MDInit MD5Init
#define MDUpdate MD5Update
#define MDFinal MD5Final

u_long md_32(char *string, int length)
{
  MD_CTX context;
  union {
    char   c[16];
    u_long x[4];
  } digest;
  u_long r;
  int i;

  MDInit (&context);
  MDUpdate (&context, string, length);
  MDFinal ((unsigned char *)&digest, &context);
  r = 0;
  for (i = 0; i < 3; i++) {
    r ^= digest.x[i];
  }
  return r;

}                                /* md_32 */

/*
 * Return random unsigned 32-bit quantity. Use 'type' argument if you
 * need to generate several different values in close succession.
 */


/********************************
 * Code from RFC 1889 ends here *
 ********************************/


/* rtp_encrypt.c */

#if 0 /* XXX encryption */
rtperror DoEncryption(context cid, struct iovec *pktpart, int pktlen,
                      int IsRTP){
  /* Encrypts and sends packet */
  /* NOTE: Can't use _RTP_Bufferspace to hold encrypted packet because
     the RTCP packet is already in _RTP_Bufferspace */
  char encryptbuf[_RTP_MAX_PKT_SIZE];
  int32 random_header = random32(cid);
  int tot_len, rundelete;
  int i, errchk;
  int use_socket;
  address_holder_t *s, *prevs;
  unix_context *uc;
  rtperror err;

  err = RTPSessionGetHighLevelInfo(cid, (void**)&uc);
  if (err != RTP_OK)
    /* The cid is bogus */
    return err;

#ifdef _RTP_DEBUG
  printf("Encrypting ");
  if (IsRTP){
    printf("RTP\n");
  }
  else printf("RTCP\n");
#endif


  err = RTP_OK;
```

```
if (uc->encrypt_encryptfunc == NULL){
  return errordebug(RTP_CANT_USE_ENCRYPTION, "DoEncryption",
                    "context %d encryption function not set",
                    (int)cid);
  return RTP_CANT_USE_ENCRYPTION;
}
/* Now copy the data into the buffer where DES can then be performed */
memcpy(encryptbuf, (char*) &random_header, sizeof(random_header));
tot_len = sizeof(random_header);
for (i=0; i < pktlen; i++){
  memcpy(&encryptbuf[tot_len], (char*)  pktpart[i].iov_base,
         pktpart[i].iov_len);
  tot_len += pktpart[i].iov_len;
}


/* NOTE: Here is where we want to call the encryption algorithm */

uc->encrypt_encryptfunc(cid, encryptbuf,
                                    (int) tot_len,
                                    uc->key);

/* Now send the packet */
s = uc->send_addr_list;
rundelete = FALSE;
while(s != NULL) {

  if(IsRTP)
    use_socket = s->rtpsocket;
  else
    use_socket = s->rtcpsocket;

  if(s->deleteflag == FALSE) {
    errchk = send(use_socket, encryptbuf, tot_len, 0);


    if (errchk < 0){
      err = errordebug(RTP_SOCKET_WRITE_FAILURE, "DoEncryption",
                       "context %d couldn't send encrypted packet",
                       (int)cid);
      /* XXX need prevent entry flag set */
      if (uc->SendErrorCallBack != NULL) {
        uc->SendErrorCallBack(cid,
                                          inet_ntoa(s->address),
                                          ntohs(s->port),
                                          s->ttl);

      }
    }
  } else {
    rundelete = TRUE;
  }

  s = s->next;
}

prevs = NULL;
if(rundelete == TRUE) {
  s = uc->send_addr_list;

  while(s != NULL) {
    if(s->deleteflag == TRUE) {

      if(prevs == NULL)
        uc->send_addr_list = s->next;
      else
        prevs->next = s->next;

      close(s->rtpsocket);
      close(s->rtcpsocket);
      free(s);
    }
```

```
      prevs = s;
      s = s->next;
    }
  }


  return err;
}
#endif


/* Returns TRUE if the addresses don't match:
   1) If of type AF_INET: addresses must match and ports must match
   or the odd port must be one greater than the even port.
   (XXX: keep this?)
   2) If of another type, the first complen bytes must match exactly.

   The low-level code makes sure that neither address is of family
   _RTP_ADDRESS_NOT_YET_KNOWN before calling this function */
int FromDifferentSource(struct sockaddr *addr1, struct sockaddr *addr2,
                      int complen){
  struct sockaddr_in *inetaddr1, *inetaddr2;

  if (addr1->sa_family != addr2->sa_family){
    return TRUE;
  }

  if (addr1->sa_family == AF_INET){
    inetaddr1 = (struct sockaddr_in*) addr1;
    inetaddr2 = (struct sockaddr_in*) addr2;

    return ((inetaddr1->sin_addr.s_addr != inetaddr2->sin_addr.s_addr) ||
            ((ntohs(inetaddr1->sin_port) & (~1)) !=
             (ntohs(inetaddr2->sin_port) & (~1))));
  }

  /* Not of type internet: compare full network address */
  return (memcmp(addr1, addr2, complen) != 0);
}
```

>> dm9000a.c

```
#include <stdio.h>
#include "dm9000a.h"
#include "dm9000a_regs.h"
#include "basic_io.h"
#include <ether.h>

dm9000a_dev g_dm9ka;
int autoReset = 0;

#define DM9KA dm9000a_dev*

void dm9ka_stats(void * pio, int iface);
int dm9ka_init(int iface);
int dm9ka_pkt_send(PACKET pkt);
int dm9ka_close(int iface);

static void rwdelay()
{
   asm volatile (
  "    nop\n"
  "    nop\n"
  "    nop\n"
  "    nop\n"
  "    nop\n"
  "    nop\n"
  "    nop\n"
  "    nop\n"
```

```
  "    nop\n"
  "    nop\n"
  "    nop\n"
  "    nop\n"
  );
}

void dm9000a_iow(unsigned int reg, unsigned int data)
{
  IOWR(DM9000A_INST_BASE, IO_addr, reg);
  rwdelay();
  IOWR(DM9000A_INST_BASE, IO_data, data);
}
unsigned int dm9000a_ior(unsigned int reg)
{
  IOWR(DM9000A_INST_BASE, IO_addr, reg);
  rwdelay();
  return IORD(DM9000A_INST_BASE, IO_data);
}

void phy_write(unsigned int reg, unsigned int value)
{
  /* set PHY register address into EPAR REG. 0CH */
  dm9000a_iow(0x0C, reg | 0x40); /* PHY register address setting,
          and DM9000_PHY offset = 0x40 */

  /* fill PHY WRITE data into EPDR REG. 0EH & REG. 0DH */
  dm9000a_iow(0x0E, ((value >> 8) & 0xFF));   /* PHY data high_byte */
  dm9000a_iow(0x0D, value & 0xFF);            /* PHY data low_byte */

  /* issue PHY + WRITE command = 0xa into EPCR REG. 0BH */
  dm9000a_iow(0x0B, 0x8);                  /* clear PHY command first */
  IOWR(DM9000A_INST_BASE, IO_data, 0x0A);  /* issue PHY + WRITE command */
  usleep(STD_DELAY);
  IOWR(DM9000A_INST_BASE, IO_data, 0x08);  /* clear PHY command again */
  usleep(50);  /* wait 1~30 us (>20 us) for PHY + WRITE completion */
}

/* DM9000_init I/O routine */
unsigned int dm9000a_reset(unsigned char *mac_address)
{
  unsigned int  i;
  /* set the internal PHY power-on (GPIOs normal settings) */
  dm9000a_iow(0x1E, 0x01);  /* GPCR REG. 1EH = 1 selected
                               GPIO0 "output" port for internal PHY */
  dm9000a_iow(0x1F, 0x00);  // GPR  REG. 1FH GEPIO0
                            //   Bit [0] = 0 to activate internal PHY */
  msleep(10);                // wait > 2 ms for PHY power-up ready

  /* software-RESET NIC */
  dm9000a_iow(NCR, 0x03);   /* NCR REG. 00 RST Bit [0] = 1 reset on,
                               and LBK Bit [2:1] = 01b MAC loopback on */

  usleep(20);               /* wait > 10us for a software-RESET ok */

  dm9000a_iow(NCR, 0x00);   /* normalize */
  dm9000a_iow(NCR, 0x03);
  usleep(20);
  dm9000a_iow(NCR, 0x00);
  dm9000a_iow(ISR, 0x3F); /* clear the ISR status: PRS, PTS, ROS, ROOS 4 bits,
                             by RW/C1 */
  dm9000a_iow(NSR, 0x2C); /* clear the TX status: TX1END, TX2END, WAKEUP 3 bits,
                             by RW/C1 */

  /* set GPIO0=1 then GPIO0=0 to turn off and on the internal PHY */
  dm9000a_iow(0x1F, 0x01);  // GPR PHYPD Bit [0] = 1 turn-off PHY */
  dm9000a_iow(0x1F, 0x00);  // PHYPD Bit [0] = 0 activate phyxcer */
  msleep(20);               /* wait >4 ms for PHY power-up */

  /* set PHY operation mode */
```

```
  phy_write(0,PHY_reset);   /* reset PHY registers back to the default state */
  usleep(50);               /* wait >30 us for PHY software-RESET ok */
  phy_write(16, 0x404);     /* turn off PHY reduce-power-down mode only */
  phy_write(4, PHY_txab);   /* set PHY TX advertised ability:
                               ALL + Flow_control */
  phy_write(0, 0x1200);     /* PHY auto-NEGO re-start enable
                               (RESTART_AUTO_NEGOTIATION +
                               AUTO_NEGOTIATION_ENABLE)
                               to auto sense and recovery PHY registers */
  msleep(20);                /* wait >2 ms for PHY auto-sense
                               linking to partner */

  /* store MAC address into NIC */
  for (i = 0; i < 6; i++)
    dm9000a_iow(16 + i, mac_address[i]);

  /* clear any pending interrupt */
  dm9000a_iow(ISR, 0x3F); /* clear the ISR status: PRS, PTS, ROS, ROOS 4 bits,
                              by RW/C1 */
  dm9000a_iow(NSR, 0x2C); /* clear the TX status: TX1END, TX2END, WAKEUP 3 bits,
                              by RW/C1 */

  /* program operating registers~ */
  dm9000a_iow(NCR, NCR_set); /* NCR REG. 00 enable the chip functions
                      (and disable this MAC loopback mode back to normal) */
  dm9000a_iow(0x08, BPTR_set); /* BPTR  REG.08  (if necessary) RX Back Pressure
                                  Threshold in Half duplex moe only:
                                  High Water 3KB, 600 us */
  dm9000a_iow(0x09, FCTR_set);  /* FCTR  REG.09  (if necessary)
                                   Flow Control Threshold setting
                                   High/ Low Water Overflow 5KB/ 10KB */
  dm9000a_iow(0x0A, RTFCR_set); /* RTFCR REG.0AH (if necessary)
                                   RX/TX Flow Control Register enable TXPEN, BKPM
                                   (TX_Half), FLCE (RX) */
  dm9000a_iow(0x0F, 0x00);      /* Clear the all Event */
  // dm9000a_iow(0x2D, 0x80);      /* Switch LED to mode 1 */

  /* set other registers depending on applications */
  dm9000a_iow(ETXCSR, ETXCSR_set); /* Early Transmit 75% */

  /* enable interrupts to activate DM9000 ~on */
  dm9000a_iow(IMR, INTR_set);   /* IMR REG. FFH PAR=1 only,
                                   or + PTM=1& PRM=1 enable RxTx interrupts */

  /* enable RX (Broadcast/ ALL_MULTICAST) ~go */
  dm9000a_iow(RCR, RCR_set | RX_ENABLE | PASS_MULTICAST);
  /* RCR REG. 05 RXEN Bit [0] = 1 to enable the RX machine/ filter */

  /* RETURN "DEVICE_SUCCESS" back to upper layer */
  return  (dm9000a_ior(0x2D)==0x80) ? DMFE_SUCCESS : DMFE_FAIL;
}

unsigned int TransmitPacket(unsigned char *data_ptr, unsigned int tx_len)
{
  unsigned int i;

  /* mask NIC interrupts IMR: PAR only */
  dm9000a_iow(IMR, PAR_set);

  /* issue TX packet's length into TXPLH REG. FDH & TXPLL REG. FCH */
  dm9000a_iow(0xFD, (tx_len >> 8) & 0xFF);  /* TXPLH High_byte length */
  dm9000a_iow(0xFC, tx_len & 0xFF);         /* TXPLL Low_byte  length */

  /* wirte transmit data to chip SRAM */
  IOWR(DM9000A_INST_BASE, IO_addr, MWCMD);  /* set MWCMD REG. F8H
            TX I/O port ready */
  for (i = 0; i < tx_len; i += 2) {
    rwdelay();
    IOWR(DM9000A_INST_BASE, IO_data, (data_ptr[i+1]<<8)|data_ptr[i] );
  }
```

```
  rwdelay();
  /* issue TX polling command activated */
  dm9000a_iow(TCR , TCR_set | TX_REQUEST);  /* TXCR Bit [0] TXREQ auto clear
                after TX completed */

  /* wait for transmit complete */
  while(!(dm9000a_ior(NSR)&0x0C)) {
    rwdelay();
  }

  /* clear the NSR Register */
  dm9000a_iow(NSR,0x00);

  /* re-enable NIC interrupts */
  dm9000a_iow(IMR, INTR_set);

  /* RETURN "TX_SUCCESS" to upper layer */
  return  DMFE_SUCCESS;
}

#ifndef MTU
#define MTU 1514
#endif

int prep_dm9000a(int index)
{
  DM9KA dm9ka = &g_dm9ka;
  NET ifp;

  ifp = nets[index];
  ifp->n_mib->ifAdminStatus = 2;   /* status = down */
  ifp->n_mib->ifOperStatus = 2;    /* will be set up in init()  */
  ifp->n_mib->ifLastChange = cticks * (100/TPS);
  ifp->n_mib->ifPhysAddress = (u_char*)dm9ka->mac_addr;
  ifp->n_mib->ifDescr = (u_char*)"DM9000A series ethernet";
  ifp->n_lnh = ETHHDR_SIZE;        /* ethernet header size */
  ifp->n_hal = 6;                  /* hardware address length */
  ifp->n_mib->ifType = ETHERNET;   /* device type */
  ifp->n_mtu = MTU;                /* max frame size */

  /* install our hardware driver routines */
  ifp->n_init   = dm9ka_init;
  ifp->pkt_send = dm9ka_pkt_send;
  ifp->n_close  = dm9ka_close;
  ifp->n_stats  = dm9ka_stats;

#ifdef IP_V6
  ifp->n_flags |= (NF_NBPROT | NF_IPV6);
#else
  ifp->n_flags |= NF_NBPROT;
#endif

  get_mac_addr(dm9ka->netp, dm9ka->mac_addr);

  /* set cross-pointers between iface and smsc structs */
  dm9ka->netp    = ifp;
  dm9ka->intnum  = DM9000A_INST_IRQ;
  dm9ka->regbase = DM9000A_INST_BASE;
  dm9ka->sending = 0;
  dm9ka->rx_ints = 0;
  dm9ka->tx_ints = 0;
  dm9ka->rcv_len = 0;
  dm9ka->snd_len = 0;
  dm9ka->tosend.q_len = 0;
  dm9ka->tosend.q_max = 0;

  ifp->n_local = (void*)dm9ka;

  return ++index;
}
```

```
/* HAL init ...
 * just init dev structs and let the stack know we are here
 */
error_t dm9000a_init(alt_iniche_dev *p_dev)
{
  prep_dm9000a(p_dev->if_num);
  return 0;
}

static unsigned char dm9000a_rxReady(DM9KA dm9ka)
{
  unsigned char rv = 0;
  /* dummy read a byte from MRCMDX REG. F0H */
  rv = dm9000a_ior(MRCMDX);
  /* got most updated byte: rx_READY */
  rv = IORD(dm9ka->regbase, IO_data) & 0x03;
  return rv;
}

static void dm9000a_isr(int iface)
{
  unsigned char rx_rdy, istatus;
  unsigned int  tmp, rx_sts, i, rx_len;
  struct ethhdr * eth;
  PACKET pkt;
  DM9KA dm9ka = (DM9KA)nets[iface]->n_local;

  /* mask NIC interrupts IMR: PAR only */
  dm9000a_iow(IMR, PAR_set);
  istatus = dm9000a_ior(ISR);

  rx_rdy = dm9000a_rxReady(dm9ka);
  rwdelay();

  while(rx_rdy == DM9000_PKT_READY)
  {
    /* get RX Status & Length from RX SRAM */
    /* set MRCMD REG. F2H RX I/O port ready */
    IOWR(dm9ka->regbase, IO_addr, MRCMD);
    rwdelay();
    rx_sts = IORD(dm9ka->regbase,IO_data);
    rwdelay();
    rx_len = IORD(dm9ka->regbase,IO_data);

    /* Check this packet_status: GOOD or BAD? */
    if( !(rx_sts & 0xBF00) && (rx_len < MAX_PACKET_SIZE) )
    {
      if ((pkt = pk_alloc(rx_len + ETHHDR_BIAS)) == NULL)
      { /* couldn't get a free buffer for rx */
        dm9ka->netp->n_mib->ifInDiscards++;
        /* treat packet as bad, dump it from RX SRAM */
        for (i = 0; i < rx_len; i += 2) {
          rwdelay();
          tmp = IORD(dm9ka->regbase, IO_data);
        }
      }
      else
      { /* packet allocation succeeded */
        unsigned char* data_ptr = pkt->nb_buff + ETHHDR_BIAS;
        /* read 1 received packet from RX SRAM into RX packet buffer */
        for (i = 0; i < rx_len; i += 2) {
          rwdelay();
          tmp = IORD(dm9ka->regbase, IO_data);
          *data_ptr++ = tmp & 0xFF;
          *data_ptr++ = (tmp>>8) & 0xFF;
        }

        pkt->nb_prot = pkt->nb_buff + ETHHDR_SIZE;
        pkt->nb_plen = rx_len - 14;
        pkt->nb_tstamp = cticks;
        pkt->net = dm9ka->netp;
```

```
        /* set packet type for demux routine */
        eth = (struct ethhdr *)(pkt->nb_buff + ETHHDR_BIAS);
        pkt->type = eth->e_type;

        /* shove packet into iniche stack's recv queue */
        putq(&rcvdq, pkt);
        SignalPktDemux();
      }
    } else {
      /* this packet is bad, dump it from RX SRAM */
      for (i = 0; i < rx_len; i += 2) {
        rwdelay();
        tmp = IORD(dm9ka->regbase, IO_data);
      }
      rx_len = 0;
    }

    rwdelay();
    rx_rdy = dm9000a_rxReady(dm9ka);
    rwdelay();
  }

  if (rx_rdy & 0x02)
  { /* status check first byte: rx_READY Bit[1:0] must be "00"b or "01"b */
    /* software-RESET NIC */
    autoReset++;
    dm9000a_iow(NCR, 0x03);   /* NCR REG. 00 RST Bit [0] = 1 reset on,
                                 and LBK Bit [2:1] = 01b MAC loopback on */
    usleep(20);               /* wait > 10us for a software-RESET ok */
    dm9000a_iow(NCR, 0x00);   /* normalize */
    dm9000a_iow(NCR, 0x03);
    usleep(20);
    dm9000a_iow(NCR, 0x00);
    /* program operating registers~ */
    dm9000a_iow(NCR, NCR_set); /* NCR REG. 00 enable the chip functions
                         (and disable this MAC loopback mode back to normal) */
    dm9000a_iow(0x08, BPTR_set); /* BPTR  REG.08 (if necessary) RX Back Pressure
                                    Threshold in Half duplex mode only:
                                    High Water 3KB, 600 us */
    dm9000a_iow(0x09, FCTR_set);  /* FCTR  REG.09  (if necessary)
                                   Flow Control Threshold setting High/Low Water
                                   Overflow 5KB/ 10KB */
    dm9000a_iow(0x0A, RTFCR_set); /* RTFCR REG.0AH (if necessary)
                                     RX/TX Flow Control Register
                                     enable TXPEN, BKPM (TX_Half), FLCE (RX) */
    dm9000a_iow(0x0F, 0x00);      /* Clear the all Event */
    dm9000a_iow(0x2D, 0x80);      /* Switch LED to mode 1 */
    /* set other registers depending on applications */
    dm9000a_iow(ETXCSR, ETXCSR_set); /* Early Transmit 75% */
    /* enable interrupts to activate DM9000 ~on */
    dm9000a_iow(IMR, INTR_set);   /* IMR REG. FFH PAR=1 only,
                                     or + PTM=1& PRM=1 enable RxTx interrupts */
    /* enable RX (Broadcast/ ALL_MULTICAST) ~go */
    dm9000a_iow(RCR , RCR_set | RX_ENABLE | PASS_MULTICAST);
    /* RCR REG. 05 RXEN Bit [0] = 1 to enable the RX machine/ filter */
  }

  /* Clear the DM9000A ISR: PRS, PTS, ROS, ROOS 4 bits, by RW/C1 */
  dm9000a_iow(ISR, 0x3F);
  /* Re-enable DM9000A interrupts */
  dm9000a_iow(IMR, INTR_set);
}


int netisrs = 0;

void dm9Ka_isr_wrap(void *context, u_long intnum)
{
  netisrs++;
  dm9000a_isr((int)context);
```

```c
}

int dm9ka_init(int iface)
{
  int err;
  DM9KA   dm9ka;

  /* get pointer to device structure */
  dm9ka = (DM9KA)nets[iface]->n_local;

  err = dm9000a_reset(dm9ka->mac_addr);

  /* register the ISR with the ALTERA HAL interface */
  err = alt_irq_register (dm9ka->intnum, (void *)iface, dm9Ka_isr_wrap);
  if (err)
    return (err);

  nets[iface]->n_mib->ifAdminStatus = 1;    /* status = UP */
  nets[iface]->n_mib->ifOperStatus  = 1;

  return (0);
}


int dm9ka_close(int iface)
{
  DM9KA   dm9ka;
  printf("dm9ka_close\n");
  nets[iface]->n_mib->ifAdminStatus = 2;    /* status = down */

   /* get pointer to device structure */
  dm9ka = (DM9KA)nets[iface]->n_local;

    /* software-RESET NIC */
  dm9000a_iow(NCR, 0x03);   /* NCR REG. 00 RST Bit [0] = 1 reset on,
                                 and LBK Bit [2:1] = 01b MAC loopback on */
  usleep(20);               /* wait > 10us for a software-RESET ok */
  dm9000a_iow(NCR, 0x00);   /* normalize */
  dm9000a_iow(NCR, 0x03);
  usleep(20);
  dm9000a_iow(NCR, 0x00);

  /* this should reset these registers anyway, but 'just in case' */
  dm9000a_iow(IMR,  0x00);  /* no interrupts */
  dm9000a_iow(RCR , 0x00);  /* disable receive */
  dm9000a_iow(0x0F, 0x00);  /* Clear the all Event */

  nets[iface]->n_mib->ifOperStatus = 2;     /* status = down */
  return 0;
}

void dm9ka_stats(void * pio, int iface)
{
  DM9KA dm9ka;
  printf("dm9ka_stats\n");
  dm9ka = (DM9KA)(nets[iface]->n_local);

  /*
    ns_printf(pio, "Interrupts: rx:%lu, tx:%lu alloc:%lu, total:%lu\n",
            smsc->rx_ints, smsc->tx_ints, smsc->alloc_ints, smsc->total_ints);
    ns_printf(pio, "coll1:%lu collx:%lu overrun:%lu mdint:%lu\n",
            smsc->coll1, smsc->collx, smsc->rx_overrun, smsc->mdint);
    ns_printf(pio, "Sendq max:%d, current %d. IObase: 0x%lx ISR %d\n",
            smsc->tosend.q_max, smsc->tosend.q_len, smsc->regbase, smsc->intnum);
  */

  return;
}

extern void irq_Mask(void);
extern void irq_Unmask(void);
```

```c
int dm9ka_pkt_send(PACKET pkt)
{
  unsigned int rv, slen;
  DM9KA dm9ka = (DM9KA)pkt->net->n_local;
  slen = pkt->nb_plen - ETHHDR_BIAS;
  if(slen < 64) slen = 64;

  //irq_Mask();

  rv = TransmitPacket(pkt->nb_prot + ETHHDR_BIAS, slen);
  if(rv == DMFE_SUCCESS) {
    /* update packet statistics */
    dm9ka->netp->n_mib->ifOutOctets += (u_long)pkt->nb_plen;
    if(*pkt->nb_prot & 0x80)
      dm9ka->netp->n_mib->ifOutNUcastPkts++;
    else
      dm9ka->netp->n_mib->ifOutUcastPkts++;
  }

  //irq_Unmask();

  if(pkt) pk_free(pkt);

  return (0);      /* alloc done interrupt will start xmit */
}
```