Crack The Covid – 19 Crisis

By NASSCOM Future Skills and IBM

**TEAM NAME: Prithivi :** Ground level Solution

Title of the Proposed Project: **VoIP**

**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;

# 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);

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);

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;

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

#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);

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

#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;

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

#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

} 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

};

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);

}

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

#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

#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;

}

}

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);

//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

#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);

#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"

#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>

#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

/\* 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;

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. \*/

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?)

* 1. 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);

}

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);

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 \*/

}