

# Demo Program

Programmer's Reference

© Zentel Telecom Ltd., 2009



# Table of Contents

Introduction	5
Compiling the applications	8
Running the demo application	9
<b>DEMO.TES</b>	<b>10</b>
DEMO.TES program description	10
<b>CHANTASK.TES</b>	<b>13</b>
CHANTASK.TES program description	13
The check_outbound() function	17
The check_inbound() function	26
<b>OUT_IVR_TASK.TES</b>	<b>30</b>
OUT_IVR_TASK program description	30
out_setup() function description	33
out_dial() function description	36
out_progress() function description	37
out_release() function description	39
<b>IN_IVR_TASK.TES</b>	<b>40</b>
IN_IVR_TASK.TES	40
<b>SCRDEMO.TES</b>	<b>42</b>
SCRDEMO.TES program description	42
<b>COMMAND.TES</b>	<b>47</b>
COMMAND.TES program description	47





of channel control tasks that are spawned is defined in a text file called DEMO.CFG which also defines whether a particular channel is an inbound or outbound channel.

If a channel is defined as an inbound channel then the CHANTASK.TES program will go into a loop waiting for an inbound call, then when an inbound call is detected it will spawn the inbound IVR task (IN\_IVR\_TASK.TES) which will simply play a voice message then hang up the call.

If a channel is defined as an outbound channel then the CHANTASK.TES will immediately spawn the outbound IVR task (OUT\_IVR\_TASK) which will wait a random amount of time before initiating a dial on the channel, then pausing for another random amount of time before hanging up.

In summary, the demo program can be configured (through the DEMO.CFG) file to spawn any number of channel control tasks (one for each channel), which can then either be inbound channels simply waiting for a call and then playing a message before hanging up, or they can be outbound channels which continuously dial out on the channel after pausing for a random amount of time.

Before describing these individual programs in more detail there are a couple of important techniques that have been used in the overall architecture of the demo programs which should be mentioned.

You will notice that there is a high level of specialization in the various tasks that comprise the demo system. For example the SCRDEMO.TES program is in charge of updating the application terminal screen for each channel -none of the other tasks writes to the screen directly (except for scrolling log messages).

Also the CHANTASK.TES program is the only program that makes any calls to the Aculab call handling function libraries to control the inbound or outbound calls to and from the system.

The actual IVR tasks that play messages and/or receive DTMF input from the caller only interact with the channel control task through either TCP/IP connections (using the CXSOCK.DLL functions) or using internal Telecom Engine messages (using the CXMSG.DLL functions).

By providing such specialisation of functionality it would be possible to completely replace the CHANTASK.TES task with another one that perhaps uses a different protocol or even uses completely different hardware. The CHANTASK.TES program supplied with the demo programs assumes that the hardware is Aculab running the ISDN protocol, but this could be changed to another application with a protocol other than ISDN (such as SS7 or CAS) without needing to change the inbound or outbound IVR tasks.

This modular design is a powerful methodology that is recommended when designing of a telecommunications systems, and the demo program shows some of these techniques in action.

#### Cross Over Testing:

It might be quite useful to define the channels of the first E1 port on a board to be inbound channels, whilst defining the the channels on the second E1 port to be outbound channels. Then if a cross-over cable is used between the first and second E1 port and the Aculab Configuration is defined correctly (E.g NET ISDN on one side and USER ISDN on the other (with correct clocking)), then the DEMO application can be used to call out of the channels on the second port and be answered by the channels on the first port.

Under this configuration the program will automatically continue to dial out on one side and answer on the other indefinitely, without needing a second system or connection to a digital PSTN.

The following sections describe each of the individual demo program tasks in more detail.

-0-

## Compiling the applications

A batch file has been provided which can be run from a command prompt to compile any of the demonstration programs. This batch file is called MK.BAT and contains the following commands:

```
set INCDIR=..\include;..\common;.
set FUNCDIR=..\common;.
tcl -e %1 %2
```

The INCDIR environment variable tells the compiler where to find any header files specified by the \$include statement. The FUNCDIR environment variable defines where any external function (\*.FUN) files reside.

This batch file assumes that the Telecom Engine binaries are included in the PATH environment variable and so to compile any of the programs one simply needs to type one of the following from the command line one after the other.

```
mk demo
mk chantask
mk scrdemo
mk command
mk in_ivr_task
mk out_ivr_task
```

Note that some of these programs are longer than the 100 line limit imposed by the Evaluation version of the compile. For this reason these programs come pre-compiled, but a full development dongle is required if you wish to modify these source files to make your own changes.

-0-



## Running the demo application

Beneath the DEMO folder where the source resides there is a sub-directory called RUN which contains a batch file called RUN.BAT.

This batch file contains the following commands:

```
set TEXDIR=..  
del *.log  
tex demo
```

The TEXDIR environment variable specifies the location of the executable TEX files which is this case is the directory above).

The RUN subdirectory also contains the config files that are needed to run the DEMO. The first configuration file (ACUCFG.CFG) is used by the standard Telecom Engine library files for the Aculab hardware (CXACULAB.DLL and CXACUDSP.DLL), and this file simply lists the serial numbers of the Aculab boards in the order that you want to open them. The ports on these boards will then be opened in turn. The first E1 port on the first signalling board specified in the ACUCFG.CFG file will become port 0 and will then increase sequentially. The first voice channel found on the first media board specified in the ACUCFG.CFG file will become voice channel 1 and will increase sequentially from there.

The second configuration file (DEMO.CFG) is aimed at the DEMO.TEX program and tells the DEMO.TEX program which channels are active and whether they are inbound or outbound channels (see [DEMO.TES program description](#)).

Assuming that the ACUCFG.CFG and DEMO.CFG files have been correctly configured and that the Telecom Engine binary files have been included in the PATH environment variable, then the RUN.BAT will launch the demo application.

-0-

# DEMO.TES

## DEMO.TES program description

The DEMO.TES program is the 'master' program that is run to start the demonstration application. The first thing that the DEMO.TES does is to open the DEMO.CFG text file which defines information for each of the E1 ports that will be used by the demo program. Each E1 port is known as a *trunk* and consists of up to 31 bearer channels (30 for ISDN since there is always one signalling port, SS7 can have any number up to 31 depending on how many signalling channels there are).

The DEMO.CFG file has the following format:

```
<Number of trunks>
<trunk1 port num (0..x)>,<trunk1
vector>,<inbound/outbound(0/1)>,<vox_offset>,<ivr_task>
<trunk2 port num (0..x)>,<trunk2
vector>,<inbound/outbound(0/1)>,<vox_offset>,<ivr_task>
etc
```

For example a valid DEMO.TES file for two inbound trunks would be:

```
2
0,11111111111111111111111111111111,0,1,in_ivr_task
1,11111111111111111111111111111111,0,31,in_ivr_task
```

and for two outbound trunks:

```
2
0,11111111111111111111111111111111,1,1,out_ivr_task
1,11111111111111111111111111111111,1,31,out_ivr_task
```

and for two inbound and two outbound trunks:

```
4
0,11111111111111111111111111111111,0,1,in_ivr_task
1,11111111111111111111111111111111,0,31,in_ivr_task
2,11111111111111111111111111111111,1,0,out_ivr_task
3,11111111111111111111111111111111,1,0,out_ivr_task
```

The first field, <trunk port num>, defines the E1 port number for the trunk ranging from 0 upwards.

The second field, <trunk vector>, is a 31 character channel vector, where each character represents one of the channels on the E1 port. 1 represents an active bearer channel and 0 represents an inactive bearer channel (e.g. a signalling channel). Since this version of the demo was written for ISDN then the vectors all have channel 16 excluded since it is used for signalling in ISDN.

The third field, <inbound/outbound flag>, defines whether channels on the trunk are inbound or outbound channels. for inbound set this field to 0, for outbound set this field to 1.

The fourth field, <vox\_offset>, defines the offset of first voice channel to be used by the first channel of the E1 trunk. For this demonstration only the inbound channels require a voice channel (to play a message to the inbound caller). For outbound channels this can be set to 0 to indicate that no voice channel needs to be allocated.

The fifth field, <ivr\_task>, defines the name of the task that will be spawned. For inbound channels the task is only spawned when a call is received on any of the channels on the E1 trunk. For outbound channels the task is spawned automatically after a random period of time. In this version of the demo program the inbound task should be set to "IN\_IVR\_TASK" which simply plays the message "DEMO.VOX" then hangs up. The outbound task is called OUT\_IVR\_TASK and sends TCP/IP messages to the CHANTASK program to control the outbound call after which it pauses a random period of time, then hangs up. For outbound calls there is no need for a voice channel in this version.

Looking into the source code we can see that after some trivial initialisation the first thing the DEMO.TES program does is to open the DEMO.CFG file:

```
fd=sys_fopen("DEMO.CFG","rs");
if(fd < 0)
    errlog("Could not open DEMO.CFG: err=",fd);
    stop;
endif
```

It then reads the <number\_of\_trunks> field using the sys\_fhgetline() function call:

```
// read the first line from the CFG file
num_trunks=sys_fhgetline(fd);
```

The program then spawns the screen handling task (SCRDEMO) passing to it various parameters to control the look of the screen, including the number of trunks that has just been read from the DEMO.CFG file:

```
// Spawn the screen driver passing:
// arg1=num_trunks,
// arg2=left column len,tot_chans
// arg3=right column len, tot_chans
var arg2:127;
var arg3:127;
arg2="1," & (num_trunks/2*31);
arg3=((num_trunks/2*31)+1) & "," & (num_trunks/2*31);
scrtask_pid=task_spawn("SCRDEMO",num_trunks,arg2,arg3);
```

After spawning the SCRDEMO task the program then sends an initialisation message to it. This is done using an internal Telecom Engine message (using the CXMSG.DLL library) as follows (it also waits for a response):

```
// Need to send a Screen initialisation message to the screen handler..
msg_put("SCREEN",SCR_INIT);

// The SCRDEMO task will send back a TE message containing a single
character ("!")
// to indicate that the screen has been initialised... so wait for this
before
// continuing..
msg_get(60);
```

Once this is done the DEMO.TES program then enters a loop for each trunk and will read the trunk configuration lines one at a time and extract the comma delimited fields from it using

the strtok() function from the CXSTRINGS.DLL library. This information is used to spawn a channel control task (CHANTASK), one for each channel of each trunk. Note also that for each trunk a message is sent to the screen handler to display information for that trunk (this will be described in more detail in the SCRDEMO.TES program description):

```
// loop through trunks..
for(trunk=1;trunk<=num_trunks;trunk++)
    // read the next configuration line from the DEMO.CFG file..
    cfg_str=sys_fhgetline(fd);

    if(cfg_str streq "")
        errlog("Invalid DEMO.CFG file (too few lines)");
        stop;
    endif

    // Extract the fields (port,Vector,direction,ivr_task)
    strtok("","");
    port=strtok(cfg_str,"");
    vector=strtok(cfg_str,"");
    tracelog("DEMO: PORT=",port," vector=",vector);
    dir_flag=strtok(cfg_str,"");
    vox_offset=strtok(cfg_str,"");
    ivr_task=strtok(cfg_str,"");

    // Send screen initialisation message for trunk
    msg_put("SCREEN",SCR_TRUNK & "," & (((trunk-1) * 31)+1) & ",31");

    // The SCRDEMO task will send back a message containing a single
character ("!")
    // to indicate that the trunk has been initialised... so wait for this
before
    // continuing..
    msg_get(60);

    adjust=0;

    // For each active channel in the vector mask spawn the channel
controller
    for(chan=1;chan <= 31; chan++,logical_line++)
        // Only spawn channel control task if vector mask for this channel
is 1
        if(substr(vector,chan,1) eq 1)

task_spawn("chantask",logical_line,port,chan,dir_flag,(vox_offset eq
0)?0:vox_offset+chan-(adjust+1),ivr_task);
        else
            // We don't waste a voice channel on inactive E1 ports to
adjust for this
            adjust++;
        endif
    endfor

endfor
```

Once this has completed then the DEMO program will exit. There should now be one CHANTASK task running for every active bearer channel defined in the DEMO.CFG. If the trunk is defined as an inbound trunk then the CHANTASK will simply be waiting for an inbound call. for outbound channels the outbound IVR task (OUT\_IVR\_TASK) will be spawned after a random pause.

# CHANTASK.TES

## CHANTASK.TES program description

For every active channel on every trunk defined in the DEMO.CFG file there will be a CHANTASK task spawned. The CHANTASK.TES is the file containing the source code for these tasks.

The CHANTASK task will act differently depending upon whether the channel it is controlling has been defined as an inbound channel or an outbound channel.

For inbound channels the CHANTASK program will simply wait for an inbound call to arrive after which it will spawn the ivr task specified in the DEMO.CFG file. The inbound ivr task provided with the demo programs (IN\_IVR\_TASK) simply plays a message (DEMO.VOX) then issues a command back to the CHANTASK program instructing it to hang up and wait for the next call.

For outbound channels the CHANTASK program waits a random amount of time before spawning the ivr task specified in the DEMO.CFG file. The outbound ivr task provided with the demo programs (OUT\_IVR\_TASK) then makes a TCP/IP connection to the CHANTASK program and sends control messages to it to instruct it to make an outbound call. Once the outbound call is connected the OUT\_IVR\_TASK will pause for another random period before issuing a hangup request.

So in both cases the CHANTASK program will spawn a child task which then communicates with the CHANTASK program by passing messages. For the inbound side the program employs internal Telecom engine messages implemented using the CXMSG.DLL function library whereas the outbound side uses the TCP/IP protocol implemented using the CXSOCK.DLL library.

After some trivial initialisation the first thing the CHANTASK program does is to obtain the arguments passed to it by the DEMO.TES master program:

```
// Get the passed arguments..
line=arg(1);
port=arg(2);
channel=arg(3);
in_out_flag=arg(4);
vox_chan=arg(5);
ivr_task=arg(6);
```

The arguments passed down from the DEMO.TES program are as follows: *line* is the logical line number (not the physical channel number); *port* is the aculab E1 port number (starting from 0); *channel* is the physical channel number on the E1 port, *in\_out\_flag* is set to 0 if this is an inbound channel, 1 if it is an outbound channel; *vox\_chan* is the voice channel number to use; *ivr\_task* is the name of the child task to spawn to control the inbound or outbound call.

The next line updates the screen status for the port through the screen handler task. This is function, *scr\_st1()* is in the *common* function directory:

```
// Set the status of the line on the screen to initialising..
scr_st1(line,"Init");
```

After this, if this is an outbound channel, the program attempts to open a listening port through which it will receive socket connections and then the messages used to control the outbound calling protocol. A unique port number is obtained by adding the task ID of the CHANTASK task to a PORT\_OFFS which has been defined as 6000 in the declaration section at the top of the program:

```
// If this is an outbound channel then create a listening socket to receive
commands//
if(in_out_flag <> DIR_IN_ONLY)
    // Use the task ID and the PORT_OFFS to generate a unique listening
port for this channel
    l_sock=Slisten(PORT_OFFS+task_getpid());
    if(l_sock < 0)
        errlog(serv_name & ": Failed to get listening socket: on port
", (PORT_OFFS+task_getpid()), " err=", l_sock);
        stop;
    endif
    debug(serv_name & ": line" & line & " GOT LISTENING SOCKET=" & l_sock & "
on port=" & (PORT_OFFS+task_getpid()));
endif
```

The *Slisten()* call returns a listening socket when can then be used to receive inbound socket connections from other tasks.

After this the H.100 (or SC bus routing is done). If a voice channel is specified then the voice channel is made to listen to the network channel and the network channel is made to listen to the voice channel in a full duplex connection via the H.100 bus as follows:

```
#####
##### INITIAL SC/H.100 BUS ROUTING
#####

#####
// Stop E1 channel from listening to anything...
CCunlisten(port,channel);
if(vox_chan > 0)
    // Stop voice channel from listening to anything
    SMunlisten(vox_chan);

    // Make vox listen to net port
    SMlisten(vox_chan,CCgetslot(port,channel));

    // Make net port listen to vox
    CClisten(port,channel,SMgetslot(vox_chan));

endif

// Get the h.100 slot of the E1 port for use later..
slot_no=CCgetslot(port,channel);
```

```
#####
#####
```

Next the channel state variable is set and the screen is updated again with the new channel status through the *scr\_st1()* function:

```
pstn_call_flag=PSTN_CALL_IDLE; // Set channel status to indicate there
is no outbound call at the moment

// Update the screen..
if(in_out_flag eq DIR_IN_ONLY)
```

```

scr_st1(line,"Wait");
scr_print(line,"","Waiting For Line Seizure....","");

else if(in_out_flag eq DIR_OUT_ONLY)
scr_st1(line,"Idle");
else
scr_st1(line,"InOut");
endif endif

```

If the channel is an inbound channel the it is initialised ready to accept and incoming call with the CCenablein() function from the aculab call control library (CXACULAB.DLL)

```

#####
##### INITIALISE THE CHANNEL READY TO MAKE/RECEIVE A CALL
#####

#####
if(in_out_flag <> DIR_OUT_ONLY)
debug(serv_name & ": ENABLING INBOUND CALLS ON Logical CHANNEL=" & line);
// Enable incoming calls on this channel
if (CCenablein(port,channel) < 0)
errlog(serv_name & ": An error occurred enabling inbound calls on
port=",port," channel= ",channel);
stop;
endif
endif
#####

#####

```

If the channel is an outbound channel then the outbound IVR task is then spawned which will make a TCP/IP connection to this CHANTASK task and issue commands to initiate the outbound call:

```

#####
##### IF THIS IS AN OUTBOUND CHANNEL THEN WE SPAWN OUTBOUND TASK
#####

#####
if(in_out_flag <> DIR_IN_ONLY)
task_spawn(ivr_task,line,port,channel,vox_chan);
endif

```

After this the program goes into a loop calling the *check\_inbound()* and *check\_outbound()* functions alternatively. The reason for the loop is to make provision to allow bi-directional channels in the future. However for this version the channel must be only inbound or outbound not both, so the loop is redundant at present:

```

// now loop checking for inbound calls or outbound socket connects
while(1)
check_outbound();
check_inbound();
task_sleep(2); // Stop tight loop to allow
endwhile

```

-0-



## The `check_outbound()` function

The following section describes the `check_outbound()` function which polls for inbound socket connections and then receives the TCP/IP messages that control the outbound call. These TCP/IP messages implement a simplified outbound call protocol implemented using text based messages.

All the TCP/IP messages sent and received by the program use the two utility functions: `TCPsend()` and `TCPrecv()` which are defined in the `common` function directory. All the messages sent over the socket connection are terminated with a carriage return character (ASCII code 13 (0x0d)) and the `TCPsend()` and `TCPrecv()` take care of adding this or using it to distinguish the end of a received message respectively.

The first statement of the `check_outbound()` function checks whether the channel is an outbound channel, and if not, the function simply returns immediately:

```
if(in_out_flag eq DIR_IN_ONLY)
    return 0;
endif
```

After this the function checks for an inbound socket connection using the `Saccept()` function call. This function will return -3 if there are no pending socket connections, otherwise it will return an incoming socket handle. Note that if the channel is outbound only then the call will 'block' for 3600 seconds waiting for an incoming connection, but if the channel is bi-directional (then the call will not block and will return immediately (note that although bi-direction channels are provided for here they have not been fully implemented in this version). Here is the code for the check for an inbound socket connection:

```
// If this is outbound only then we can block for a while (1 hr)
if(in_out_flag eq DIR_OUT_ONLY)
    debug(serv_name & "Line " & line & " Going into BLOCKING
Saccept()...");
    # Wait up to 1 hour for inbound connection
    a_sock=Saccept(l_sock,3600);
    debug(serv_name & "Line " & line & " Returned from BLOCKING Saccept()
with sock=" & a_sock);
    // otherwise assume bi-directional channel so just do a quick check for
socket connection
    else
        a_sock=Saccept(l_sock);
    endif

    if(a_sock < 0)
        // Check for error (hopefully we will never see this since it is not
clear if we could recover)
        if(a_sock <> EWOULDBLK)
            errlog(serv_name & ": Error in Saccept attempt to get another
listening socket");
            Sclose(l_sock);

            // Attempt to recover (by obtaining another listening socket..)
            while(1)
                l_sock=Slisten(PORT_OFFS+task_getpid());
                if(l_sock < 0)
                    errlog(serv_name & ": Error listening on port " &
(PORT_OFFS+task_getpid()) & " retrying...");
                else
                    debug(serv_name & ": Listening socket is " & l_sock);
                    break;
            endwhile
        endif
        sleep(5);
    endwhile
```

```

        endif
        return 0;
    // else we got a inbound socket connection (to start an outbound call)
    else
        debug(serv_name & ": line=" & line & " RECEIVED INBOUND SOCKET
CONNECTION (TO START OUTBOUND CALL)");
        pstn_call_flag=PSTN_CALL_START; # We now have an active connection
    endif

```

If an inbound socket connection is received then the *pstn\_call\_flag* status is set to PSTN\_CALL\_START and the function goes into a loop waiting for inbound messages to arrive to control the issue of an outbound call on the channel. The messages to control an outbound call have been implemented using a simple text based protocol where the message format is as follows:

<Command>,<Session ID>,<Data>

Where <Command> can be one of the following as defined in DEMO.INC from the *include* sub-directory:

```

// Message commands for driving outbound calls.. Client-> Chantask
const C_PS_DIAL="60";
const C_PS_RELEASE="61";
const C_PS_SETUP="70";

// Response commands indicating call progress.. Chantask->Client
const C_PS_ALERTING="62";
const C_PS_PROCEEDING="63";
const C_PS_PROGRESSING="64";
const C_PS_DISCONNECTED="65";
const C_PS_TASKFAILURE="66";
const C_PS_CONNECTED="67";
const C_PS_ATTACH="68";
const C_PS_DETACH="69";
const C_PS_SETRESP="71";
const C_PS_NOTIFY="73";

```

The program then enters a loop, alternatively checking for received messages from the client task and checking for events on the outbound channel (only after the dial has been initiated).

There are three possible messages that can be received from the client side:

C\_PS\_SETUP - Exchanges some information about the H.100 bus slots for routing purposes prior to dialing out.

C\_PS\_DIAL - Initiates the actual outbound call

C\_PS\_RELEASE - Request hangup and release of the outbound call.

The code for this loop is as follows (with some detail removed and replaced with pseudo code):

```

#####
# Loop looking for messages - stay in this loop until either socket closed
or dial session complete

#####
while(1)

    // Go check socket for incoming message..
    // Use TCPreadd to get packet all in one go (with overlapped mulitple
messages stored in omsg)
    rc=TCPreadd(a_sock,&msg,&omsg,0);

    // Check for lost socket

```

```

if(rc < 0)
    // Make it look like a release
    rcvd_sess = ps_session;
    msg=C_PS_RELEASE & "," & NORMAL_CLEARING;
    strtok("","");
    command=strtok(msg,"");
else if(rc eq 1)
    # extract the command
    strtok("","");
    command=strtok(msg,"");
    rcvd_sess = ps_session;
endif endif

# if we get a message
if(rc eq 1)
switch(command)
    // C_PS_SETUP ,session, slot_type, slot_num ,offered_vox_chan [,
extra_parms]
    // C_PS_SETRESP,result, slot_type, slot_num ,vox_chan_used,
defer_vox_routing_flag
    case C_PS_SETUP:
        etc....
        //
C_PS_DIAL,session,dest_tel,caller_ID,num_type,cli_pass,idd_prefix[,dest_subaddr
, CLI_subaddr]    case C_PS_DIAL:
        etc....
        // C_PS_RELEASE,session,<cause>
    case C_PS_RELEASE:
        etc....
endswitch
endif

// If a dial has been initiated
if(pstn_call_flag eq PSTN_CALL_PROGRESS)
    EventCode = CCstate(port, channel);
    if (LastEventCode <> EventCode)
        send back call progress message..
        etc..
    endif
endif
endwhile

```

The first message received from the client task is a C\_PS\_SETUP message which contains the following data:

70,<slot\_type>,<slot\_num>,<offered\_vox\_chan>

- The <slot\_type> defines whether switching is to be done over a local logical H.100 bus or some other external inter-chassis switching type (only local bus TS\_TYPE\_LOGICAL is supported by the demo (see *demo.inc* for these definitions)).
- The <slot num> is the handle for the H.100 timeslot
- The <offered\_vox\_chan> is only used by protocols that require a voice channel (such as R2/MF) and is not needed by the demo.

In a real application the <setup> message would allow for another channel (voice or E1 channel) to be automatically connected to the outbound call (Either to connect two conversations together or to play a message to the answered party). However in the demo program the setup message is largely redundant.

In response the CHANTASK sends back a C\_PS\_SETRESP message:

71,<result code>, <slot type>, <local\_bus\_slot> ,<vox\_chan\_used>,  
<defer\_vox\_routing\_flag>

- The <result code> is 0 for successful setup or a negative error code (E.g. if <slot\_type> not supports)
- The <local\_bus\_slot> is the h.100 (or external bus slot) that the outbound E1 channel has been nailed to.
- The <vox\_chan\_used> is the actual voice channel that was used (only applies to R2 etc channels)
- The <defer\_routing\_flag> is set to 1 if the inbound channel should defer listening to the outbound channel until the first call progress message (e.g. C\_PS\_PROCEEDING, C\_PS\_ALERTING etc) is received. This will happen in some protocols (like R2) where the inbound caller would hear the exchange of R2 MF tones if the routing was done immediately after the C\_PS\_SETRESP message was received. For ISDN, SS7 etc <defer\_routing\_flag> is set to 0 so the H.100 bus routing connection can be made immediately.

The next message received from the client task would be a C\_PS\_DIAL message which instructs the outbound channel to dial a given number. The format of this message is as follows:

60,<dest\_tel>,<caller\_ID>,<num\_type>,<cli\_pass>,<idd\_prefix>[,<dest\_subaddr>,<CLI\_subaddr>

- The <dest\_tel> is the destination telephone number
- The <caller ID> is the caller id passed from the client application
- The <num\_type> should be set to one of C\_TYPE\_LOCAL, C\_TYPE\_NATIONAL, or C\_TYPE\_INTERNATIONAL as defined in DEMO.INC
- <cli\_pass> should be set to 1 to pass the given caller ID or 0 to block caller ID
- The <idd\_prefix> will be set to the prefix used for international calls (Eg 00 or 010 etc)
- The <dest\_subaddr> and <CLI\_subaddr> allow for additional address information to be sent (e.g. extension)

This message causes the program to actually initiate an outbound call on the E1 channel. The code for this is as follows:

```

    case C_PS_DIAL:
        // Check that the pstn_call_flag is in the correct state to receive a
        C_PS_DIAL message
        if (pstn_call_flag eq PSTN_CALL_PROGRESS or pstn_call_flag eq
        PSTN_CALL_MONITOR)
            errlog(serv_name, ": ERROR - GOT C_PS_DIAL DURING ALREADY ACTIVE
            DIAL");
        else
            // Extract all the fields from the message
            dest_tel_no= strtok(msg, ",");
            caller_id= strtok(msg, ",");
            dest_num_type= strtok(msg, ",");
            cli_pass= strtok(msg, ",");
            idd_prefix= strtok(msg, ",");
            dest_sub_addr= strtok(msg, ",");
            caller_sub_addr= strtok(msg, ",");

            ##### Normalise Destination Telephone number and set number
            type #####
            // if the destination number starts with idd_prefix then remove it
            and
            if (substr(dest_tel_no, 1, length(idd_prefix)) streq idd_prefix)
                dest_tel_no= substr(dest_tel_no, length(idd_prefix)+1);
                debug(serv_name & ": Stripping IDD prefix from dest num gives "
            & dest_tel_no);
                dest_num_type= C_TYPE_INTERNATIONAL;
            endif

            // Assume its a national number if it is not specified
            if (dest_num_type streq "")
                dest_num_type= C_TYPE_NATIONAL;

```

```

endif

##### Convert internal number type to the Aculab ISDN number
type and plan number plan
if(dest_num_type eq C_TYPE INTERNATIONAL)
    dest_ntype=NT_INTERNATIONAL;
    dest_nplan=NP_UNKNOWN;    // Hardcode this for now..
else if(dest_ntype eq NT_NATIONAL)
    dest_ntype=NT_NATIONAL;
    dest_nplan=NP_UNKNOWN;    // Hardcode this for now..
else if(dest_ntype eq NT_SUBSCRIBER_NUMBER)
    dest_ntype=NT_SUBSCRIBER_NUMBER;
    dest_nplan=NP_UNKNOWN;    // Hardcode this for now..
// For any other type assume national
else
    dest_ntype=NT_NATIONAL;
    dest_nplan=NP_UNKNOWN;    // Hardcode this for now..
endif endif endif

debug(serv_name & ": CALLER ID=" & caller_id & " cli_pass=" &
cli_pass);

##### Set the override caller ID (the one actually being sent
to the network) depending on the cli_pass flag
// The cli_pass flag defines how the caller ID is passed (0=None,
1=Use one in message,2=use default)
if(not cli_pass)
    o_caller_id="";
else if(cli_pass eq 1)
    o_caller_id=caller_id;  # Use the passed CLI
else
    o_caller_id=default_cid;  # Use the default CID
endif endif

##### Update the status on the screen
scr_st2(line,11,"DIAL");
scr_print(line,"","CID=" & o_caller_id & " TEL=" & dest_tel_no
, "");

##### Setup the call and do the dial..

CCsetparm(port,channel,PARM_TYPE_OUT,CP_Q931_DEST_NUMBERING_TYPE,dest_ntype);
CCsetparm(port,channel,PARM_TYPE_OUT,CP_Q931_DEST_NUMBERING_PLAN,dest_nplan);

// Hard code these for now..
CCsetparm(port, channel,PARM_TYPE_OUT,
CP_Q931_ORIG_NUMBERING_TYPE, NT_NATIONAL);    # hard code for now
CCsetparm(port, channel,PARM_TYPE_OUT,
CP_Q931_ORIG_NUMBERING_PLAN, NP_UNKNOWN);    # hard code for now

connected_flag=0;
debug(serv_name & "Now making call, number = '" & dest_tel_no & "'
cid=" & o_caller_id & " type=" & dest_ntype & " plan=" & dest_nplan);

rc=CCmkcall(port, channel, dest_tel_no,o_caller_id);

##### Clear the last event received variable and change the
pstn_call_flag status
LastEventCode = "";
pstn_call_flag=PSTN_CALL_PROGRESS; # We now have a call in
progress
endif

```

The last line of this code sets the *pstn\_call\_flag* to PSTN\_CALL\_PROGRESS so that the second part of the loop will now check for changes in the state of the channel and pass these changes back to the client task as call progress messages. Before looking at that code there is one

more case statement which handles the receipt of a C\_PS\_RELEASE statement. Note that if the TCP/IP socket is unexpectedly closed by the client then the program simulates the receipt of a C\_PS\_RELEASE statement (see the code at the top of the loop above). Here's where the C\_PS\_RELEASE is handled:

```

case C_PS_RELEASE:
    # Check for bad message
    if(not pstn_call_flag)
        errmsg(serv_name,": ERROR - RECEIVED C_PS_RELEASE MESSAGE WITH NO
ACTIVE DIAL");
    else
        # If call is in progress then hangup the call
        if(pstn_call_flag eq PSTN_CALL_PROGRESS or pstn_call_flag eq
PSTN_CALL_MONITOR)
            CCdisconnect(port,channel,LC_NORMAL);
            connected_flag=0;
            wait_idle_then_release();

            // handle bi-directional channel (future use)
            if(in_out_flag <> DIR_OUT_ONLY)
                CCenablein(port,channel);
            endif
        endif

        // Update the screen with new status
        if(in_out_flag eq DIR_OUT_ONLY)
            scr_stl(line,"Idle");
        else
            scr_stl(line,"InOut");
        endif

        // If H.100 routing has been done then remove routing
        if(link_flag)
            // Stop network from listening to anything...
            CCunlisten(port,channel);
            // reconnect any voice channel to the E1 channel
            if(vox_chan > 0)
                SMunlisten(vox_chan);
                # Make vox listen to net again
                SMlisten(vox_chan,CCgetslot(port,channel));
            endif
            link_flag=0;
        endif

        // disconnect the accepted socket
        Sclose(a_sock);
        pstn_call_flag = PSTN_CALL_IDLE;
        return 0;
    endif

```

So the C\_PS\_RELEASE message, once received, will cause the call to be disconnected (if it hasn't already) then the call session is released, the H.100 bus routing is reset, the screen is updated and the inbound TCP/IP socket is closed. The *wait\_idle\_then\_release()* function, does what it says, i.e it waits for the channel state to go IDLE then it calls the CCrelease() function. This is done after a CCdisconnect() has been issued on the channel:

```

func wait_idle_then_release()
dec
    var firststate:8;
    var state:8;
end

firststate=CCstate(port,channel);
if(firststate <> CS_IDLE)

```

```

// loop waiting for channel to go idle
while(1)
    CCwait(port,channel,CC_WAIT_FOREVER,&state,firststate);
    if(state eq CS_IDLE)
        break;
    else
        firststate=state;
    endif
    // prevent tight loop so window messages can be processed
    task_sleep(1);
endwhile
end

// State is IDLE so release..
CCrelease(port,channel);

endfunc

```

Once the C\_PS\_DIAL message has been received and an outbound call is in progress then the second part of the loop checks for changes in the state of the call on the channel and passes these changes to the client task as call progress messages (C\_PS\_ALERTING, C\_PS\_PROCEEDING etc). The code for this is as follows:

```

##### POLL THE CHANNEL TO CHECK FOR EVENTS #####
if(pstn_call_flag eq PSTN_CALL_PROGRESS)
    EventCode = CCstate(port, channel);

    // check is new event has been received
    if (LastEventCode <> EventCode)
        LastEventCode = EventCode;
        ldatetime = sys_date() & "," & sys_time();

        switch(EventCode)
            case CS_CALL_CONNECTED: # call answered
                ##### Get the start date and time of the
                call (for working out duration later)
                ssdate=sys_date();
                sstime=sys_time();

                connected_flag=1;

                ##### Update the screen
                status..#####
                debug(serv_name & ": Call state CONNECTED");
                scr_st2(line,2,"ANSW");

                ##### Send back C_PS_CONNECTED message.
                #####
                msg = C_PS_CONNECTED & "," & EventData & "," &
                ldatetime;
                rc = TCPsend(a_sock,msg);

                ##### If BUS slot was given in SETUP message
                then do bus connection here!
                if(tslot_type eq TS_TYPE_SC)
                    debug(serv_name & ": SCB: making port=" &
                    port & " channel=" & channel & " listen to (sent) tslot_no=" & tslot_no);
                    # Don't route timeslot if -1 specified
                    if(tslot_no >= 0)
                        CCunlisten(port,channel);
                        logical_bus_slot=24*4096+tslot_no;
                        CClisten(port,channel,logical_bus_slot);
                        link_flag=1;
                    endif
                else if(tslot_type eq TS_TYPE_LOGICAL)

```

```

        debug(serv_name & ": !!!LOGICAL: making port="
& port & " channel=" & channel & " listen to (sent) tslot_no=" & tslot_no);
        if(tslot_no >=0)
            CCunlisten(port,channel);
            CClisten(port,channel,tslot_no);
            link_flag=1;
        endif
    else
        errlog(serv_name,"UNSUPPORTED BUS
TYPE=",tslot_type);
    endif endif
    # This shouldn't happen as we should get a DISCONNECT
first - handle it just in case
    case CS_IDLE:
        cause=NORMAL_CLEARING;
        debug(serv_name & ": Call state IDLE WITHOUT
DISCONNECT!");
        if(ssdate streq "")
            call_duration = 0;
        else
            call_duration =
timesub(date(),time(),ssdate,sstime);
        endif

        scr_st2(line,3,"HGUP");
        ssdate = "";

        // Send back C_PS_DISCONNECT message to
controlling task
        msg = C_PS_DISCONNECTED & "," & cause & "," &
ldatetime & "," & call_duration;
        rc = TCPsend(a_sock,msg);
    case CS_REMOTE_DISCONNECT:
        debug(serv_name & ": Call state DISCONNECTED");
        cause=CCgetcause(port,channel,1);
        debug(serv_name & " DISCONNECTED gave cause=" &
cause);

        if(ssdate streq "")
            call_duration = 0;
        else
            call_duration =
timesub(date(),time(),ssdate,sstime);
        endif

        scr_st2(line,3,"HGUP");
        ssdate = "";

        // Send back C_PS_DISCONNECT message to
controlling task
        msg = C_PS_DISCONNECTED & "," & cause & "," &
ldatetime & "," & call_duration;
        rc = TCPsend(a_sock,msg);

    case CS_WAIT_FOR_OUTGOING:
        debug(serv_name & ": Call state
WAIT_FOR_OUTGOING");
    case CS_OUTGOING_RINGING:
        # destination
terminal got call request
        debug(serv_name & ": Call state OUTGOING RINGING
received");

        EventData=0;
        scr_st2(line,12,"RING");
        msg = C_PS_ALERTING & "," & EventData & "," &
ldatetime;

        rc = TCPsend(a_sock,msg);
    case CS_PROGRESS:
        # received progress event
        debug(serv_name & ": Call state PROGRESSING

```



```

received");
                                EventData=0;
                                msg = C_PS_PROGRESSIONING & "," & EventData & "," &
ldatetime;
                                rc = TCPSend(a_sock,msg);
                                case CS_OUTGOING_PROCEEDING:      # network accepted
call request
                                debug(serv_name & ": Call state PROCEEDING
received");
                                EventData=0;
                                msg = C_PS_PROCEEDING & "," & EventData & "," &
ldatetime;
                                rc = TCPSend(a_sock,msg);
                                case CS_NOTIFY:
                                debug(serv_name & ": Call state NOTIFY received");
                                EventData=0;
                                msg = C_PS_NOTIFY & "," & EventData & "," &
ldatetime;
                                rc = TCPSend(a_sock,msg);
                                default:
                                errlog(serv_name & "Unexpected event ", EventCode);
                                endswitch
                                endif
                                endif

```

So in all cases above, once a change in state is received then a call progress message is sent back to the client task.

All the call progress messages have the format:

<Progress type>,<Event Data>,<timestamp: DDMMYYHHMMSS>

With the exception of the C\_PS\_DISCONNECTED message which has the call duration appended as follows:

C\_PS\_DISCONNECTED,<Event Data>,<timestamp: DDMMYYHHMMSS>,<Call duration>

For calls that did not get answered then the duration will be set to 0.

If the channel state is changed to CS\_CONNECTED then any H.100 routing is done to connect the conversations (only if bus information was supplied in the C\_PS\_SETUP message) and the screen is updated.

If the channel state is changed to CS\_DISCONNECTED then the call duration is calculated (if any) and the screen status is also updated.

## The `check_inbound()` function

The following section describes the `check_inbound()` function which waits for an inbound call on the E1 channel and once a call has been received it will spawn the IVR task that will then play messages and take DTMF digits etc. The IVR task that is spawned is specified in the DEMO.CFG file and by default is the IN\_IVR\_TASK.TES program supplied with the demo application.

After spawning the IVR task the program then goes into a loop alternatively waiting for a message from the IVR task or for a hangup signal on the E1 channel.

The first thing the function does is check whether the channel is an outbound only channel and if so it simply returns:

```
// Return immediately if this is an outbound only channel
if(in_out_flag eq DIR_OUT_ONLY)
    return 0;
endif
```

Then the function goes into a loop waiting for an inbound call to arrive at the channel. If the channel is inbound then the `CCwait()` function will block waiting for an inbound call. There is also code here to handle the case of a bi-directional channel in which case the `CCstate()` call is used to poll the channel to see if there is an outbound call and the function will return back to the main loop if not:

```
// If this is an inbound channel then we can block task and wait
if(in_out_flag eq DIR_IN_ONLY)
    // Loop waiting for incoming call
    while(1)
        debug(serv_name & " About to CCWait (FOREVER)");
        x=CCwait(port,channel,CC_WAIT_FOREVER,&event);
        debug(serv_name & " CCWait() returned with event=" & event);

        if(x<0)
            if(event eq CS_INCOMING_CALL_DET)
                break;
            else
                applog(serv_name & ": port=",port," channel=",channel," got unexpected event=",event);
            endif
            // prevent tight loop
            task_sleep(1);
        endif
    endwhile
    # otherwise we poll for incoming calls
else

    # Check to see if there is an incoming call.
    if (CCstate(port,channel) <> CS_INCOMING_CALL_DET)
        return 0;
    endif
endif
```

If an inbound call was detected by the channel state becoming CS\_INCOMING\_CALL\_DET then the DID and CLI are extracted, the screen updated and the call is accepted. In this version of the demo there is no attempt to inspect the CLI or DID make a decision about which IVR task to spawn or whether to reject the call. Instead the call is always accepted with the `CCaccept()` function:

```
// Get ANI and DNIS .
```

```

errctl(1);
CCgetparm(port,channel,CP_ORIGINATING_ADDR,&ANI);
errctl(0);
CCgetparm(port,channel,CP_DESTINATION_ADDR,&did);

debug(serv_name & " DID=" & did & " CID=" & ANI);
scr_print(line,"","DID=" & did & " CID=" & ANI & "","","");

// In DEMO we don't check DID or ANI we just answer the call
x=CCaccept(port, channel);
if(x<0)
    errlog(serv_name & " Port=",port," channel=",channel," CCaccept failed.. releasing the call");
    CCdisconnect(port, channel, LC_CALL_REJECTED);
    wait_idle_then_release();

# re-enable inbound calls
debug(serv_name & "Doing Enable in on port=" & port & " channel=" & channel);
x=CCenablein(port,channel);

// Resetting the screen
if(in_out_flag eq DIR_IN_ONLY)
    scr_stl(line,"Wait");
    scr_print(line,"","Waiting For Line Seizure....","");
else if(in_out_flag eq DIR_OUT_ONLY)
    scr_stl(line,"Idle");
else
    scr_stl(line,"InOut");
endif endif

return -1;
endif

```

Once the call has been accepted then the IVR task specified in the DEMO.CFG (passed down as an argument from DEMO.TES) is spawned, and the screen status is updated. If the program could not spawn the IVR application then the channel is reset:

```

##### Spawn INBOUND IVR TASK #####
child_pid=task_spawn(ivr_task,line,port,channel,vox_chan);

// If we couldn't spawn the child task then release the call ..
if(child_pid<0)
    errlog(serv_name & " Port=",port," channel=",channel," CCaccept failed.. releasing the call");
    CCdisconnect(port, channel, LC_CALL_REJECTED);
    wait_idle_then_release();

# re-enable inbound calls
debug(serv_name & "Doing Enable in on port=" & port & " channel=" & channel);
x=CCenablein(port,channel);

// Resetting the screen
if(in_out_flag eq DIR_IN_ONLY)
    scr_stl(line,"Wait");
    scr_print(line,"","Waiting For Line Seizure....","");
else if(in_out_flag eq DIR_OUT_ONLY)
    scr_stl(line,"Idle");
else
    scr_stl(line,"InOut");
endif endif

return -1;
endif

scr_st2(line,"INV",ivr_task);

```

Next the function calls the *await\_msgs()* function, which loops waiting for either a hangup signal on the E1 channel or a message from the IVR task. In this case the messages received from the IVR task are passed using the Telecom Engine Internal messaging functions from the CXMSG.DLL library. Here is the code for this loop:

```
// Loop forever waiting for messages and/or hangup signal
while(1)
    // check for disconnect
    a_x=CCstate(port,channel);
    if(a_x eq CS_REMOTE_DISCONNECT)
        task_hangup(a_pid);
        debug(serv_name & " Detected hangup on Line " & line & " Issued task hangup on pid=" & a_pid);
    endif

    a_msg=msg_get(0);
    // IF we didn't get anything for one hour this looks suspicious (IVR task hung?)
    // In this version just set the status to "—" in red as an alert
    if(a_msg streq "")
        if(sys_tmsecs() >= 3600)
            scr_st2(line,12,"—");
            sys_tmstart();
        endif
        sleep(3);
        continue;
    endif

    // Extract fields from received message..
    strtok("", "");
    a_cmd=strtok(a_msg, ",");
    a_cause=strtok(a_msg, ",");
    switch(a_cmd)
        // Hangup (and go blocking?? (Depends on protocol))
        case GEN_HANGUP:
            debug(serv_name & " Received GEN_HANGUP message on Line " & line);
            CCdisconnect(port, channel, a_cause,1);
            a_hflag=1;
        case GEN_RESTART:
            debug(serv_name & " GENERIC: received GEN_RESTART message on Line " & line);
            # Send restart acknowledgment...
            msg_put(msg_pid(), GEN_ACK);
            if(a_hflag eq 0)
                CCdisconnect(port, channel, a_cause,1);
                a_hflag=1;
            endif
        endif

        # Clear any DTMF digits
        if(vox_chan)
            SMClrtones(vox_chan);
        endif

        // Reset the scbus routing in case it was changed...
        CCunlisten(port,channel);
        if(vox_chan > 0)
            SMinlisten(vox_chan);
            SMListen(vox_chan,CCgetslot(port,channel));
        endif

        wait_idle_then_release();

        # Do we need to re-enable inbound calls?
        if(in_out_flag <> DIR_OUT_ONLY)
            debug(serv_name & "Doing Enable in on port=" & port & " channel=" & channel);
```

```
        CGenablein(port,channel);
    endif

    // Resetting the screen
    if(in_out_flag eq DIR_IN_ONLY)
        scr_stl(line,"Wait");
        scr_print(line,"","Waiting For Line Seizure....","");
    else if(in_out_flag eq DIR_OUT_ONLY)
        scr_stl(line,"Idle");
    else
        scr_stl(line,"InOut");
    endif endif

    return 0;

default:
    errlog(serv_name & ": REceive invalid protocol message='" & a_msg & "'");
endswitch
task_sleep(1); # prevent tight looping
endwhile
```

The two messages that can be received are:

GEN\_HANGUP - this causes the channel to be disconnected

GEN\_RESTART - this causes the channel to be released and made ready to accept a new inbound call.

The *wait\_idle\_then\_release()* function is described above in the description of the *check\_outbound()* function.

-O-

## OUT\_IVR\_TASK.TES

### OUT\_IVR\_TASK program description

The OUT\_IVR\_TASK.TES program is the client program that takes control of an outbound channel and causes a outbound call to be dialled on that channel using the simplified internal TCP/IP protocol described above in the CHANTASK.TES program description.

After initialisation the program simply pauses a random amount of time between 10 and 90 seconds before connecting to the CHANTASK task that is in charge of the channel and issuing a request for an outbound call to be initiated on the channel. Once connected the program will again pause a random amount of time between 10 and 90 seconds before issuing a hangup request. The program will then go back to the beginning and start the sequence again.

Therefore for outbound channels the demo application will continuously dial out on those channels at random intervals until the program is terminated.

After some trivial initialisation the first part of the code retrieves the arguments passed down from the CHANTASK program, which provides information about which channel the task is in control of:

```
// Get the passed arguments..  
serv_name=arg(0);  
line=arg(1);  
port=arg(2);  
channel=arg(3);  
vox_chan=arg(4);
```

Next the program retrieves the process ID of the parent CHANTASK task so that the TCP/IP port can be calculated (the listening port will be 7000 + taskID). It then obtains a random number and pauses for this amount of seconds:

```
// Get the task ID of parent (used to get the listen port of channel task)  
parent_pid=task_parentid();  
  
// Pause for a random number of seconds between 1 and 90  
rnd_seed();  
  
int pause_secs;  
pause_secs=rand(10,90);  
  
debug(serv_name & "About to pause for " & pause_secs & " seconds..  
  
// Pause for this number of seconds  
task_sleep(pause_secs*10);
```

The *rnd\_seed()* and *rand()* functions are simple pseudo random number generator functions found in the *common* function directory.

Next the program calls the outbound call setup sequence:

```
// Now initiate the call setup sequence..  
x=out_setup();  
if(x<0)  
    // Force ourselves into onsignal  
    task_hangup(task_getpid());  
endif
```

The *out\_setup()* function initiates the setup sequence which consists of connecting to the parent CHANTASK task that is in charge of the outbound channel, and exchanging the C\_PS\_SETUP and C\_PS\_SETRESP messages. A more detailed description of the *out\_setup()* function will be given later. Note that if an error occurs then the *task\_hangup()* function is called which forces the program to jump onto the onsignal function to clear down the call.

After the setup sequence has completed then the dial sequence is started through the *out\_dial()* function:

```
// Now send the dial request to the channel task
x=out_dial("0123456789","0987654321");
if(x < 0)
    task_hangup(task_getpid());
endif
```

This *out\_dial()* simply sends the C\_PS\_DIAL message to the CHANTASK task (the *out\_dial()* function will be described later). Again, any error results in the *task\_hangup()* function forcing the program to jump into the onsignal function.

Next a loop is entered waiting for call progress messages to be received from the CHANTASK task showing the outbound call in progress. The *out\_progress()* function simply waits for TCP/IP messages using the TCPRead() function and will be described later. The loop will be broken once the C\_PS\_CONNECTED message is received to indicate that the outbound call has been answered. If a C\_PS\_DISCONNECT message is received then *task\_hangup()* forces a jump to onsignal to clear down the call.

```
// Loop getting call progress from outbound channel
while(1)
    x=out_progress(&cp_type,&cp_data,&cp_date,&cp_time,&cp_duration);
    if(x < 0)
        task_hangup(task_getpid());
    endif

    if(cp_type eq C_PS_CONNECTED)
        break;
    else if(cp_type eq C_PS_DISCONNECTED)
        task_hangup(task_getpid());
    endif endif

    // Prevent tight loop
    task_sleep(1);
endwhile
```

The obtains another random number of seconds and then loops waiting for this number of seconds to expire or a C\_PS\_DISCONNECT message to be received over the TCP/IP connection from the CHANTASK task:

```
// Get a random number of seconds to pause before hanging up...
pause_secs=rand(10,90);

tmr_start();
// Loop waiting for hangup indication or timeout
while(1)
    // check for timer..
    if(tmr_secs() > pause_secs)
        task_hangup(task_getpid());
    endif

    x=out_progress(&cp_type,&cp_data,&cp_date,&cp_time,&cp_duration);
```

```
if(x < 0)
    task_hangup(task_getpid());
endif

    if(cp_type eq C_PDISCONNECTED)
        task_hangup(task_getpid());
    endif

    // Prevent tight loop
    task_sleep(1);
endwhile
endmain
```

The *onsignal* function (which is jumped to whenever as *task\_hagup()* function is called) provides a single restart point for the program, when the outbound call is released and the program is restarted ready to initiate another dial on the outbound channel. The *out\_release()* carries out the release of the outbound call and will be described later:

```
// This will be jumped to as soon as hangup is received..
onsignal

    // Release the outbound call
    out_release();

    // end the application
    applog("Restarting the application");
    restart;

endonsignal
```

The following sections describe in more detail the following functions used above:

*out\_setup()*  
*out\_dial()*  
*out\_progress()*  
*out\_release()*



## out\_setup() function description

The *out\_setup()* function is responsible for establishing a TCP/IP connection to the CHANTASK task and to exchange the C\_PS\_SETUP and C\_PS\_SETRESP messages. Although the exchange of these messages is largely redundant in the demo application (since we are not connecting two conversations together) it is still useful to look at the function code to understand how the socket library works:

First the *Sconnect()* call is used to make a TCP/IP connection and will return either a negative error code or a connecting socket. The function will return immediately, even though the connection might not have been fully established, so the *Scheck()* function is used to check for when the connection is fully established (or an error occurs).

```
#####
# Setup the Outbound call by making socket connection and then sending
# the initial setup message to exchange information about h.100
# timeslots etc
#####
func out_setup()
int rc;
var out_command:10;
var out_sess:10;

#####
# Now Initiate the dial sequence. The channel task is always waiting on
# port 6000+task_id

out_progress=0;

debug(serv_name & " SETUP About to connected to port=" & (PORT_OFFSETS+parent_pid));
// Make a socket connection to channel control task on port=6000+task_id
out_sock=Sconnect("localhost",PORT_OFFSETS+parent_pid);

// Loop waiting for connection or error
while(1)
    // Check if socket is ready for write (connection complete)
    x=Scheck(out_sock,1);
    if(x eq 1)
        break;
    endif

    // Check for error
    x=Scheck(out_sock,2);
    if(x eq 1)
        errlog(serv_name & ": Error could not connect to channel task on port=" &
(PORT_OFFSETS+parent_pid));
        return -1;
    endif
    // prevent tight loop to allow windows to receive messages..
    task_sleep(1);
endwhile

if(out_sock < 0)
    errlog(serv_name & ": failed to make socket connection to channel control task on
port=",PORT_OFFSETS+parent_pid);
    return -1;
endif

debug(serv_name & " SETUP connected to port=" & (PORT_OFFSETS+parent_pid));

// If we get here then we have made a socket connection..
```

```
out_progress=1; // socket connection made (from this point jump to onsignal to force cleardown)..
```

The next part of the function sends the C\_PS\_SETUP message and then waits for the response using the *TCPsend()* and *TCPrecv()* functions:

```
// Send setup request... format is:
// C_PS_SETUP,<sessionID>,<Bus type>,<Bus Timeslot>,<offered vox chan>
out_msg=C_PS_SETUP & "," & TS_TYPE_LOGICAL & ",-1,0";

rc=TCPsend(out_sock,out_msg);
if(rc < 0)
    errlog(serv_name & ": Failed to send SETUP message: err=",rc);
    // Force jump to onsignal to clear down..
    task_hangup(task_getpid());
endif

out_progress=2; // Sent setup request..

tmr_start();
out_msg="";
out_omsg="";
while(1)
    rc=TCPread(out_sock,&out_msg,&out_omsg,0);

    // Check for loss of connection
    if(rc < 0)
        errlog(serv_name & ": Socket disconnect during SETUP exchange: err=",rc);
        return -1;
    // If there was a message then extract the command
    else if(rc eq 1)
        debug(serv_name & ": SETUP: RECEIVED MESSAGE=" & out_msg);
        strtok("","");
        out_command=strtok(out_msg,",");
        break;
    endif endif

##### CHECK FOR TIMEOUT IF NO MESSAGE #####
if(tmr_secs() > 15)
    errlog(serv_name,": SETUP - Timeout waiting for SETUPRESP from channel task: line=",line);
    return -1;
endif
task_sleep(1);
endwhile
```

The next step of the function is to check that the correct response has been received and to return to the main *check\_inbound()* function:

```
switch(out_command)
    case C_PS_SETRESP:
        debug(serv_name & "SETUP: GOT SETUP RESP - msg = " & out_msg);

        // Extract the data fields
        // Get the return code
        out_setuprc=strtok(out_msg,",");

        // Check for setup error of some kind..
        if(out_setuprc <> 0)
            errlog(serv_name,"SETUP Failed result=" & out_setuprc);
            return -2;
        endif
```

```
    out_progress=3;    // Set flag to indicate set-up complete

    // Setup succeeded - extract rest of fields

    // Get the bus timeslot and type (E.g. LOGICAL, SCBUS etc)
    out_slottype=strtok(out_msg, ",");
    out_slot=strtok(out_msg, ",");
    out_voxchan=strtok(out_msg, ",");
    out_scodefer=strtok(out_msg, ",");

    debug("SETUP: obtained slottype=" & out_slottype);
    debug("SETUP: obtained slot    =" & out_slot);
    debug("SETUP: obtained voxchan =" & out_voxchan);

    return 1;
default:
    errlog(serv_name, ": SETUP", "Getting nonsense while waiting for SETRESP! msg=" & out_msg);
    return -1;
endswitch

endfunc    // End of the out_setup() function
```

-O-

## out\_dial() function description

The *out\_dial()* function sends the C\_PS\_DIAL message over the socket connection to initiate an outbound call on the channel.

The code is as follows:

```
func out_dial(dest_telno,orig_telno)
int rc;
    out_msg = C_PS_DIAL & "," & dest_telno & "," & orig_telno & "," & NT_NATIONAL & "," & 1 & "," & "00";
    rc = TCPsend(out_sock,out_msg);

    if(rc < 0)
        errlog(serv_name & ": Failed to send DIAL message: err=",rc);
        // Force jump to onsignalto clear down..
        task_hangup(task_getpid());
    endif
    out_progress=4;    // Dial request has been sent...
    return 0;
endfunc
```

-O-

## out\_progress() function description

The *out\_progress()* function simply waits for a call progress message, extracts the time-stamp and event data then returns this information to the calling program. All arguments passed to this function are pointers to variables to contain the results:

```
func out_progress(px_event,px_data,px_ldate,px_ltime,px_duration)
int rc;

    // Check for a call progress messages
    rc = TCPread(out_sock,&out_msg,&out_omsg,0);
    if(rc < 0)
        errlog(serv_name,": PROGRESS - Error reading socket error=",rc);
        return -1;
    else if(rc eq 1)
        strtok("", "");
        *px_event=strtok(out_msg, ",");
    else
        return 0;
    endif endif

    out_lastmsg=out_msg;
    switch(*px_event)
    case C_PS_ALERTING:
        *px_data=strtok(out_msg, ",");
        *px_ldate=strtok(out_msg, ",");
        *px_ltime=strtok(out_msg, ",");
        debug("XXXPROGRESS: check_inbound(): GOT ALERTING: data code=" & *px_data);
        return 1;
    case C_PS_PROCEEDING:
        debug("XXXPROGRESS: check_inbound(): GOT PROCEEDING");
        *px_data=strtok(out_msg, ",");
        *px_ldate=strtok(out_msg, ",");
        *px_ltime=strtok(out_msg, ",");
        return 1;

    case C_PS_PROGRESSING:
        debug("XXXPROGRESS: check_inbound(): GOT PROGRESSING");
        *px_data=strtok(out_msg, ",");
        *px_ldate=strtok(out_msg, ",");
        *px_ltime=strtok(out_msg, ",");
        return 1;

    case C_PS_CONNECTED:
        debug("XXXPROGRESS: check_inbound(): GOT CONNECTED");
        *px_data=strtok(out_msg, ",");
        *px_ldate=strtok(out_msg, ",");
        *px_ltime=strtok(out_msg, ",");
        out_progress=5; # We are connected
        return 2;
    case C_PS_DISCONNECTED:
        debug("XXXPROGRESS: check_inbound(): GOT DISCONNECTED");
        *px_data=strtok(out_msg, ",");
        *px_ldate=strtok(out_msg, ",");
        *px_ltime=strtok(out_msg, ",");
        *px_duration=strtok(out_msg, ",");

        return 1;
    case C_PS_TASKFAILURE:
        debug("XXXPROGRESS: check_inbound(): GOT TASKFAILURE");
        *px_data=strtok(out_msg, ",");
        *px_ldate=strtok(out_msg, ",");
```

```
    *px_ltime=strtok(out_msg,",");  
  
    return 1;  
default:  
    errlog(serv_name,": PROGRESS Received invalid message=" & out_msg);  
  
    return 0;  
endswitch  
end
```

-O-

## out\_release() function description

The *out\_release()* function relies on the global *out\_progress* variable which is maintained by all of the other *out\_xxxx()* functions as follows. Depending on which state the outbound call has reached then function will either close the socket connection or will send as C\_PS\_RELEASE message then close the socket connection:

```
// this function hangs up and releases socket etc.
func out_release()
int rc;

switch(out_progress)
case 0:
    return 0;
    // We have connected to channel task
case 1:
    Sclose(out_sock);

    // We have sent setup request..
case 2:
    Sclose(out_sock);

    // We have received setup response
case 3:
    Sclose(out_sock);

    // We have sent dial request
case 4:
    // Need to send release request to clear down call
    out_msg = C_PS_RELEASE & "," & NORMAL_CLEARING;
    rc = TCPsend(out_sock,out_msg);
    Sclose(out_sock);
    // Connection established..
case 5:
    // Need to send release request to clear down call
    out_msg = C_PS_RELEASE & "," & NORMAL_CLEARING;
    rc = TCPsend(out_sock,out_msg);
    Sclose(out_sock);
endswitch

out_progress=0;
return 0;
endfunc
```

-O-

# IN\_IVR\_TASK.TES

## IN\_IVR\_TASK.TES

The IN\_IVR\_TASK.TES is the program that is spawned when an inbound call is received. It simply plays a voice file (DEMO.VOX) and then sends a request to the CHANTASK.TES task to hangup the call and reset the channel ready to take another call.

After some trivial initialisation the program first reads the arguments that have been passed to it from the CHANTASK task:

```
// Get the passed arguments..  
serv_name=arg(0);  
line=arg(1);  
port=arg(2);  
channel=arg(3);  
vox_chan=arg(4);
```

It then simply plays the voice file DEMO.VOX, pauses for 20 seconds, then sends a message to the controlling CHANTASK task to hangup the channel:

```
// Play a voice file..  
SMplay(vox_chan,"DEMOVOX");  
  
// Pause for 20 seconds  
sleep(200);  
  
// Get the Process ID of parent task  
parent_pid=task_parentid();  
  
// Send hangup request to parent task  
msg_put(parent_pid,GEN_HANGUP);  
  
// Force ourselves into onsignal  
task_hangup(task_getpid());
```

The final *task\_hangup()* call forces the program to jump to the *onsignal* function which requests that the channel be released and reset ready for another call:

```
// This will be jumped to as soon as hangup is received..  
onsignal  
// Get the Process ID of parent task  
parent_pid=task_parentid();  
  
// Send GEN_RESTART message  
msg_put(parent_pid,GEN_RESTART);  
  
// Wait for Ack message  
msg=msg_get(60);  
  
// end the application  
stop;  
endonsignal
```

Notice that the messages are sent to the CHANTASK task using internal Telecom Engine



messages frmo the CXMSG.DLL library.

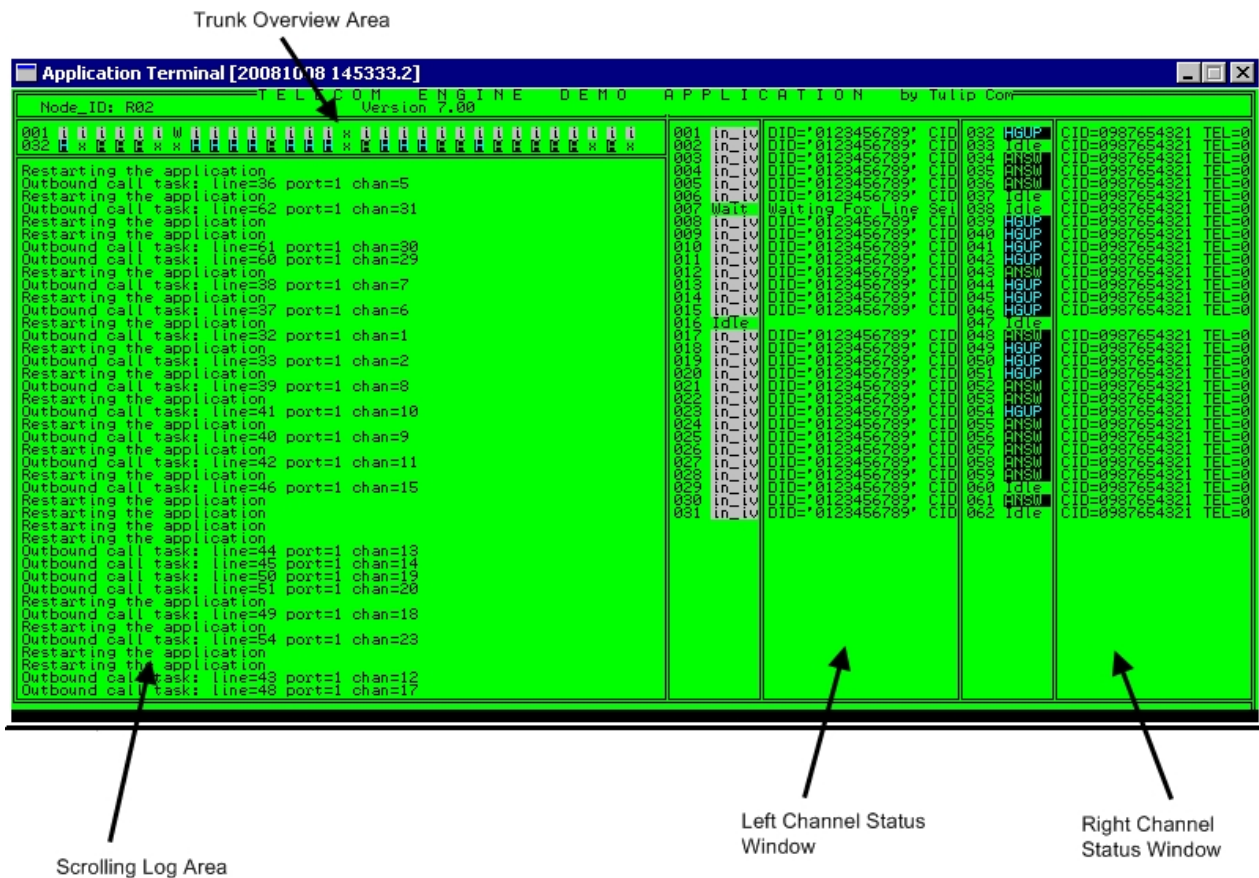
-0-

# SCRDEMO.TES

## SCRDEMO.TES program description

The SCRDEMO.TES program is in charge of maintaining the application terminal screen. There are a number of utility function provided with the DEMO application and that reside in the *common* subdirectory that work with the SCRDEMO.TES program to allow the screen status to be updated.

The SCRDEMO splits the screen into four separate areas shown in the screen shot below:



The Trunk overview area allows a complete quick glance summary showing what is happening on every channel of each trunk. Each line of the Trunk overview area shows the logical channel number for the first channel on the trunk followed by a single character channel status indicator (which will be the first character of the channel status from the left or right channel status windows). By appropriate use of colour this can provide a very useful indicator of what is happening on each channel of the trunk. In the DEMO program, inbound channels with active calls have the colour black on whitewhereas the outbound channels have various colours on a black background (E.g. Red on black for RING, green on black for ANSWER, cyan on black for HANGUP etc).

The left and right channel status windows show more detailed channel status information about what is happening on each channel and are individually scrollable by the SCRDEMO.TES program. The channel status window consists of three parts (from left to right):

- Channel number
- Five character channel status

## c) Twenty character channel status detail

In the DEMO program the channels are split 50/50 between the left and right channel status windows and the number of channels will thus depend on the DEMO.CFG file.

The Scrolling Log Area is the only part of the screen that is written to directly by other programs apart from the SCRDEMO.TES program through the `aplog()`, `errlog()`, `tracelog()` function calls provided by the CXTERM.DLL library.

The functions supplied with the DEMO program that allow for the screen to be updated are as follows:

```
scr_print(line,colour,detail_text>window_id)
scr_stat(stat_text)
scr_stat1(line,stat_text)
scr_stat2(line,colour,stat_text)
```

`scr_print()` writes to the detailed status part of the status window for the specified logical channel in the specified colour. The `window_id` argument is not used by the SCRDEMO.TES program, and is for future use.

`scr_stat()` writes to the 5 character abbreviated status part of the status area for the current logical channel (defined by the global `line` variable passed as an argument). It uses the default screen colour.

`scr_stat1()` allows the logical line number to be specified rather than using the current one specified by the `line` global variable. It uses the default screen colour.

`scr_stat2()` allows both the logical line number to be specified as well as the colour that should be used.

The way these functions communicate with the SCRDEMO.TES program is through a set of global arrays declared by the SCRDEMO.TES program using the global array functions from the CXGLB.DLL library.

The code for the `scr_print()` function is as follows:

```
# Function: scr_print
# synopsis: include "screen.inc"
#          scr_print(a_line,a_attr,a_msg,a_window)
#          a_line      - The line number
#          a_attr       - The colour (or "") for default colour
#          a_msg        - The message to write
#          a_window     - The the window number to write to (unused in SCRDEMO.TES)
func scr_print(a_line,a_attr,a_msg,a_window)
    task_defersig("");
    array_set("line_text",a_line,a_msg);
    array_set("line_color",a_line,a_attr);
    array_set("line_txflag",a_line,1);
    task_defersig("");
end
```

The code for the `scr_stat2()` function is as follows:

```
#          scr_stat: Put short (up to 5 characters) status field for line
#
```

```

#      ARGUMENTS
#      a_line      The line number
#      a_attr      Colour to use
#      a_msg       Message string

func scr_stat2(a_line,a_attr,a_msg)
    task_defersig("");
    array_set("line_stat",a_line,a_msg);
    array_set("line_attr",a_line,a_attr);
    array_set("line_stflag",a_line,1);
    task_defersig("");
end

```

Notice that both functions use the task\_defersig() function to prevent a hangup signal from interrupting the code block before it has completed. There are then four global arrays that can be set to update the screen:

"line\_text" - This array holds the detail text for a particular logical line  
 "line\_color" - This array holds the colour attribute that is to be used for the detail text display  
 "line\_txtflg" - Set this to 1 to indicate to the SCRDEMO.TES that the detail text for this logical line has changed  
 "line\_stat" - This array holds the abbreviated 5 character status for the logical line  
 "line\_attr" - This array holds the colour attribute that is to be used for the status display  
 "line\_stflag" - Set this to 1 to indicate to the SCRDEMO.TES that the status test for this logical line has changed

From the above code it can probably be guessed that the SCRDEMO.TES simply polls these arrays looking for any elements in the "line\_txtflg" or "line\_stflag" set to 1. If any are found then the respective status or detail text on the screen is updated and the "line\_txtflg" or "line\_stflag" array element is set back to 0.

There is probably not much to be gained from showing the entire code from the SCRDEMO.TES program since much of the code is dedicated to calculating the offsets on the screen where the channel status and/or detail should be written. Instead it is probably more useful to just look at the important parts of the program.

After some trivial initialisation the first thing the SCRDEMO program does is to declare the global arrays described above:

```

## Allocate the screen arrays..
array_dim("line_stat",1024,10);
array_dim("line_attr",1024,8);
array_dim("line_text",1024,80);
array_dim("line_color",1024,8);
array_dim("line_stflag",1024,1); # change flag for status
array_dim("line_txflag",1024,1); # change flag for text

```

The array\_dim(name,num\_elements,length) function allocates a dynamic global array with the number of elements given by *num\_elements* where the length of each element is given by *length*. As can be seen above, the maximum number of logical channels that can be handled by the SCRDEMO.TES program is 1024.

Next the SCRDEMO.TES retrieved the arguments passed down to it from the DEMO.TES program:

```

# Get the startup parms
bsv_parm1=arg(1);
bsv_parm2=arg(2);
bsv_parm3=arg(3);

```

```

strtok("", "");
trunk_area_depth=strotok(bsv_pam1, "", "");
if(trunk_area_depth > 32)
    trunk_area_depth=32;
else if(trunk_area_depth < 1)
    trunk_area_depth=5;
endif endif

# fix the line depth
line_depth=LINE_DEPTH;

strtok("", "");
lline_start=strotok(bsv_pam2, "", "");
lline_tot=strotok(bsv_pam2, "", "");
strtok("", "");
rline_start=strotok(bsv_pam3, "", "");
rline_tot=strotok(bsv_pam3, "", "");

```

The arguments passed down are as follows:

*arg1*=<number of trunks>

*arg2*=<start line of left channel status win>, <number lines in left win>

*arg3*=<start line of right channel status win>, <number lines in right win>

*arg2* and *arg3* contain two values each separated by commas so the *strotok()* function is used to extract the individual values.

After this the program enters the main loop where it alternatively waits for internal telecom engin messages to arrive and/or refreshes the screen by iterating over the above arrays to update the screen:

```

# Now loop waiting for commands
while(1)
    #applog("Waiting for message");
    r_msg=msg_get(1);
    if(r_msg streq "")
        goto refresh;
    endif

    r_cmd=substr(r_msg,1,2);

    switch(r_cmd)
    case SCR_INIT:
        init_screen();
        # The calling task will be waiting for a response so give it one!
        msg_put(msg_pid(), "!");
    case SCR_TRUNK:
        # Get the trunk channel start and channel range
        strtok("", "");
        tot_trunks++;
        trunk_start[tot_trunks]=strotok(substr(r_msg,4), "", "");
        trunk_chans[tot_trunks]=strotok(substr(r_msg,4), "", "");

        init_trunk(tot_trunks);

        msg_put(msg_pid(), "!");
    case SCR_LEFT:
        curr_scroll=LEFT_SIDE; # Left screen
        next_screen(LEFT_SIDE);

    # Right select
    case SCR_RIGHT:
        curr_scroll=RIGHT_SIDE; # Left screen
        next_screen(RIGHT_SIDE);

```

```
case SCR_BOTH:
    curr_scroll=BOTH_SIDES; # both sides of screen
    next_screen(BOTH_SIDES);
case SCR_GOTO:
    goto_line=substr(r_msg,4,3);
    #applog("Goto line=" & goto_line);
    # Which side is it?
    if(goto_line >= lline_start and goto_line <= (lline_start+lline_tot-1))
        lcurr_page=(goto_line-lline_start)/line_depth+1;
        #applog("Calc new lcurr_page=" & lcurr_page);
        draw_left();

        else if(goto_line >= rline_start and goto_line <= (rline_start+rline_tot-1))
            rcurr_page=(goto_line-rline_start)/line_depth+1;
            #applog("Calc new rcurr_page=" & rcurr_page);
draw_right();
    endif endif
# Toggle screen formats
case SCR_STYLE:
    # do nothing..

case SCR_NEXT:
    #applog("SCRDRV2: Next received");
    next_screen(curr_scroll);
case SCR_PREV:
    prev_screen();
case SCR_REDRAW:
    redraw_screen();
endswitch

# This is the refresh screen part..
refresh:
    timer_refresh();
endwhile
```

The messages that can be received by the SCRDEMO.TES program allow for the screen to be initialised and then for another task to accept commands from the keyboard and to scroll the left and right status windows (or both) or to go to a specific line number in the left or right status windows.

There is no keyboard interface task provided with this version of the DEMO program and then only message that is used is the SCR\_INIT message which is sent from the DEMO.TES program at startup and causes the screen to be initialised.

-0-

# COMMAND.TES

## COMMAND.TES program description

The COMMAND.TEX program allows for the terminal console to be placed in 'command' mode and accept input from the keyboard.

In order to enter 'command' mode the user would hit the colon (":") key, then type in the command followed by the <ENTER> key. To escape from 'command' mode the user would hit the <ESC> key.

The commands that have been provided in the COMMAND.TEX program are simply to allow the right hand side of the screen to be scrolled if there are more channels specified in the DEMO.CFG file than can be simultaneously displayed.

The commands accepted by the COMMAND.TEX program are as follows:

l <ENTER>	- Scroll the left hand list of channels
r <ENTER>	- Scroll the right hand list of channels
b <ENTER>	- Scroll both left and right hand lists of channels
g [channel] <ENTER>	- Goto the given [channel] number (ie scroll co this channel is at the top of list)
<ENTER>	- Scroll the left, right or both lists depending on the last command entered from above
<ESC>	- Escape from command mode

These commands will only take effect if the list of channels exceeds the height of the screen (i.e there are at least 4 E1s running under the DEMO application).

In the source file (COMMAND.TES) you will see that after some initialisation the program enters a loop waiting for the colon ":" key to be press to put the terminal into 'command' mode. For this the program simply calls the term\_kbget() function to receive keys pressed on the keyboard. If a ":" is hit the program obtains the last row of the terminal screen from the term\_size() function and then enters a second loop accepting input from the keyboard throught the term\_kbedit() function call. The source snippet for this can be seen below:

```
# Loop forever
while(1)
    input_str=term_kbget();
    if(input_str streq ":")
        # Get the screen size..
        term_size(&scr_rows,&scr_cols);
        if(scr_cols > 127)
            line_width=126;
        else
            line_width=scr_cols-1;
        endif
        input_str="";

    # Loop accepting commands
    while(1)
        term_cur_pos(scr_rows-1,0);
        term_print(ljust(":", " ", line_width));
        input_str=term_kbedit(scr_rows-1,1,scr_cols-1,input_str,0);
        # Get the character that terminated the input (E.g. ESC or Enter)
        c=term_kbgetx();
        term_kbget();

        etc..
```

The key that caused the `term_kbedit()` to terminate is retrieved by a call to `term_kb_getx()` and if it was the ESC key then we break out of the inner loop to take the terminal out of command mode.

Otherwise the `input_str` variable has been returned from the `term_kbedit()` function it is then parsed and the command and parameters are extracted to an array called `parms[]`:

```

if(c eq ESC)
    term_cur_pos(scr_rows-1,0);
    term_print(ljust("", " ",line_width+1));
    break;
else
    # Strip off any question mark at end
    if(substr(input_str,length(input_str),1) streq "?")
        input_str=substr(input_str,1,length(input_str)-1);
    endif
    # Get each parmater in turn
    i=1;
    for(p=1;p<=MAX_PARMS;p++)
        parms[p]="";
    endfor

    break_next=0;
    for(p=1;p <= MAX_PARMS;p++)
        # skip spaces
        for(; i <= length(input_str); i++)
            if(substr(input_str,i,1) strneq " " and
substr(input_str,i,1) strneq "`t")
                break;
            endif
        endfor

        # If rest of input contained only spaces
        if(i > length(input_str))
            break;
        endif

        # extract next parm ...
        for(;i <= length(input_str);i++)
            c=substr(input_str,i,1);
            if(c streq " " or c streq "`t")
                break;
            endif
            parms[p]=parms[p] & c;
        endfor
        if(break_next eq 1)
            break;
        endif
    endfor

```

Once the command and parameters have been parsed from the input string then the program can execute the commands. For the DEMO application the only commands are to allow the screen to be scrolled by sending internal messages to the SCRDEMO.TEXT program. The messages that can be sent are defined in SCREEN.INC and are as follows:

```

const SCR_NEXT   = "03"; # Scroll left, right or both screens (depends on last command)
const SCR_LEFT   = "06"; # Select and scroll left screen
const SCR_RIGHT  = "07"; # Select and scroll right screen
const SCR_BOTH   = "08"; # Scroll both
const SCR_GOTO   = "09"; # Goto a certain line

```

These messages are sent to the SCRDEMO.TEXT task using the `msg_put()` function as shown in the following code snippet:



```
        # If there was no input (only ENTER pressed) then just send the
SCR_NEXT command
        if(parms[1] streq "")
            input_str="";
            msg_put("SCREEN",SCR_NEXT);
            continue;
        endif

        # We have got a command
        p--;
        switch(parms[1])
            case "l":
                msg_put("SCREEN",SCR_LEFT);
            case "r":
                msg_put("SCREEN",SCR_RIGHT);
            case "b":
                msg_put("SCREEN",SCR_BOTH);
            case "g":
                msg_put("SCREEN",SCR_GOTO & " " & parms[2]);
            default:
                input_str=input_str & "?";
                continue;
        endswitch
        input_str="";
    endif # esc not pressed
endwhile # End loop accepting commands
endif # End if ":"
endwhile
endmain
```

-O-

# Index

## - C -

CHANTASK.TES program description 13  
COMMAND.TES program description 47  
Compiling the applications 8

## - D -

DEMO.TES program description 10

## - I -

Introduction 5  
IN\_IVR\_TASK.TES 40

## - O -

out\_dial() function description 36  
OUT\_IVR\_TASK program description 30  
out\_progress() function description 37  
out\_release() function description 39  
out\_setup() function description 33

## - R -

Running the demo application 9

## - S -

SCRDEMO.TES program description 42

## - T -

The check\_inbound() function 26  
The check\_outbound() function 17

