CXPIKA

Host Media Processing (HMP) Library
for the Telecom Engine

Table of Contents

Introduction

The CXPIKA.DLL library provides the functionality for the PIKA host Media Processing (HMP) range of products. This includes the PIKA Analog FXO Trunk and Analogue FXS Station boards; the PIKA digital PCI and PCIe boards; and the on-host VOIP (SIP) functionality.

The CXPIKA.DLL libary is written using the high level PIKA GrandPrix API and many of the functions will map one-to-one to the GrandPrix library equivalent. However much of the difficult programming tasks such as multi-threading, event handling and developing under the asynchronous state-machine model will be allieviated by programming under the Telecom Engine.

The CXPIKA library contains both the call control functionality required for making and receiving calls, as well as the media processing functionality for playing speech prompts, receiving DTMF digits, recording messages etc.

All the call control functions have function names with the prefix: PKCall.. (E.g. PKCallAnswer(), PKCallAccept() etc), whereas the media processing functions have the prefix: PKChan... (E.g. PKChanPlay(), PKChanRecord() etc).

For all functions both call control and media processing (PKCallXxx() and media processing PKChanXxx()), the functions take the group number and channel number as the first two arguments. Note that channel numbers run from 1 up to the maximum number of channels in the group, whereas group numbers start from 0.

-o-

A Simple Example

Probably the best way to show the basic library functions and the library calling conventions is to provide a simple example.

The example below simply waits for an incoming call on the first channel of the first *group*, then plays a message and receives some DTMF. The concept of a *group* is defined by the PIKA software as a collection of channels possibly associated with an E1 port on Digital boards, or otherwise a logical collection of channels such as VOIP or Analogue channels. Group numbers start from 0, whereas channel numbers start from 1.

```
$include "pika.inc"

int group, chan, x, event;
var filename:64;
var tone:1;

main
    // Variables to hold incoming address information
    var From:127;
    var To:127;
    var Display:127;

    // hard-code these for this example
    group=0
    chan=1;
    filename="hello.vox";

   // Turn on trace for call controlfunctions and events
    PKCallTrace(group,chan,2);
    // Turn on trace for media processing functions and events
    PKChanTrace(group,chan,2);

    // Loop waiting for events
    while(1)
       // Wait until next event received
       x=PKCallWait(group,chan,PK_WAIT_FOREVER,&event);
       if(x > 0)
           applog("group=",group," chan=",chan," received event=",event);

           // Look for Incoming call event
           if(event eq PKX_EVENT_GROUP_INCOMING_CALL)
               // Force jump to onsignal on DISCONNECT
               PKCallUseSignal(group,channel);

               // When we get INCOMING CALL then retreive the call info
               x=PKCallGetInfo(group,chan);

               // Now get the parameters that we want (To, From, Display)
               x=PKCallGetParm(group,chan,PK_PARM_TO,&From);
               x=PKCallGetParm(group,chan,PK_PARM_FROM,&To);
               x=PKCallGetParm(group,chan,PK_PARM_DISPLAY,&Display);

               applog("INFO FROM   =",From," To=",To," Display=",Display);

               // Accept the call
               x=PKCallAccept(group,chan);

           // Look for Call Accepted Event...
           else if(event eq PKX_EVENT_CALL_ACCEPTED)
               // Answer the call once we get Call Accepted event
               x=PKCallAnswer(group,chan);
           // When we get call Answered event then we break the loop
```

```
            else if(event eq PKX_EVENT_CALL_ANSWERED)
                break;
            endif endif endif
        else if(x eq 0)
            applog("group=",group," chan=",chan," No event recieved");
        endif endif
        // Prevent tight loop (to allow windows events to be processed)
        task_sleep(1);
    endwhile

    // Call answered
    applog("group=",group," chan=",chan," Call Answered...");

    // Simply loop forever playing DEMO.VOX
    // until DISCONNECT causes jump to onsignal..
    while(1)
        // Play a prompt
        x=PKChanPlay(group,chan,filename);

        // Wait for up to 3 DTMF digits..
        x=PKChanWaitDTMF(group,chan,3,40,40);

        // Retrieve the entered DTMF digits
        input=PKChanGetDTMF(group,chan);

        applog("PKChanGetDTMF returned input=",input);

        // Prevent tight loop (to allow windows events to be processed)
        task_sleep(1);
    endwhile

endmain

//*********************************************
//******* THIS IS THE ONSIGNAL FUNCTION ********
//*********************************************
onsignal

int state;

   applog("group=",group," chan=",chan," IN ONSIGNAL!");

   // Disconnect our end of the call..
   x=PKCallHangup(group,chan,0);
   // Wait for channel to go idle
   while(1)
       x=PKCallState(group,chan,&state);
       applog("group=",group," chan=",chan," state=",state);
       if(state eq PKX_CALL_STATE_IDLE)
           // Release the call resources..
           x=PKCallRelease(group,chan);
           break;
       endif
       // Prevent tight loop (to allow windows events to be processed)
       task_sleep(1);
   endwhile

   // Restart the program to wait for another call..
   restart;
endonsignal
```

The program should be fairly self explanatory but I will describe the key parts of the program below.

The “pika.inc” file is provided with the library and defines all the constants that are used with the library such as PKX_EVENT_GROUP_INCOMING_CALL, PK_EVENT_CALL_ACCEPTED

etc.

In the *main* routine, after initializing some variables and switching on function and event trace (with PKChanTrace() and PKCallTrace()) the program then enters a loop waiting for events to be received from the underlying PIKA software. The PKCallWait *(group,channel,timeout_100ms,&event)* function call will wait for the specified timeout (in 10ths of a second) for an event . If the timeout is defined as -1 (PK_WAIT_FOREVER) then the call will not return until an event is received or it is aborted by a PKCallWaitAbort() call.

The first event we expect to receive here is the PKX_EVENT_GROUP__INCOMING_CALL to indicate that an incoming call has been received for this *group* and *channel*. Once a PKX_EVENT_GROUP__INCOMING_CALL event has been received then we retrieve some call information using the PKCallGetInfo() and PKCallGetParm*()* functions, then we accept the call using PKCallAccept() function. After the PKX_EVENT_CALL_ACCEPTED has been received we answer the call with PKCallAnswer*()*. Once the PKX_EVENT_CALL_ANSWERED event has been received then the call is now answered and live and so the program breasks from the loop to carry out some media functions (such as play and record etc).

It should be noted that after the PKX_EVENT_GROUP_INCOMING_CALL event is detected the program calls PKCallUseSignal*()* to force and immediate jump to the *onsignal* function if a PKX_EVENT_CALL_DISCONNECT event is received thereafter. This provides a mechanism to have a single exit point for the program when the Disconnect event can be handled and the call can be cleared down.

After this the program enters another loop where some calls to the Media processing functions are made to play a voice prompt and to receive some DTMF digits. This loop will continue indefinitely until the caller disconnects the phone whereby the program will immediately jump to the *onsignal* function to cleardown the call and restart the application ready for another call.

First a voice prompt is played to the caller using the PKChanPlay*(group,chan,filename)* function after which the application waits for some DTMF input using the PKChanWaitDTMF *(group,chan,num_dig,first_delay10ths, inter_delay10ths)* function.
The PKChanWaitDTMF*()* function puts the task into a blocking state until one of the terminating conditions is met. The terminating condition could be that the requested number of digits has been received (*num_dig)* or the timeout waiting for the first digit (*first_delay10ths)* was exceeded, or the inter-digit timeout was exceeded (*inter_delay10ths)*. PKChanWaitDTMF*()* will then copy any digits received into the internal digit buffer for the voice channel. The next call PKChanGetDTMF*(group,chan)* returns any digits that have been copied to the internal digit buffer for the specified channel.

The application then checks if a tone was received and if so will use the received DTMF to make the name of a prompt file which is then played using the SMplay() function.

Once a PKX_EVENT_CALL_DISCONNECT event occurs and the program is forced into the the *onsignal* function, the call is disconnected using the PKCallHangup*(group,channel,cause)* call and the application goes into a loop waiting for the channel to return to the PKX_CALL_STATE_IDLE state before releasing the call with PKCallRelease*(group,channel)* and restarting the program to wait for the next call.

Notice that here the program uses the PKCallState*()* function to wait for the channel to enter the PKX_CALL_STATE_IDLE state. It would also be valid to wait for the channel to receive the PKX_EVENT_CHANNEL_READY event by calling the PKChanLastEvent*()* function ..

-o-

Example to Handle Multiple Channels

The best way to simulataneously handle multiple inbound calls is to have a master program 'spawn' a separate channel control task for each of the inbound channels. The channel control task will then receive the group and channel number from the master task and then wait for an inbound call on that channel.

The following code will spawn 20 channel control tasks to simultaneously wait for inbound calls (this is sufficient for the evaluation version of the PIKA HMP software which provides a maximum of 20 SIP channels).

master.tes:

```
main
int group, chan;

     // Assume group=0 for this example
     group=0;

     // Turn on trace for this group
     PKGroupTrace(group,2);

     // spawn 20 channel control tasks to receive inbound calls
     for(chan=1;chan <=20;chan++)

         // spawn the task 'pika_in.tex' which receives
         // the group and channel as arguments..
         task_spawn("pika_in",group,chan);

     endfor
endmain
```

pika_in.tes:

```
// This defines all the constants for the cxpika.dll library
$include "pika.inc"

// define some global variables..
int group, chan;

main
    var filename:127;
    var From:127;
    var To:127;
    var Display:127;

    // receive the group and channel numbers as passed
    //from the master.tes program
    group=task_arg(1);
    chan=task_arg(2);
    filename="helloworld.vox";

    // Loop waiting for events
    while(1)
       // Wait until next event received
       x=PKCallWait(group,chan,PK_WAIT_FOREVER,&event);
       if(x > 0)
           applog("group=",group," chan=",chan," received event=",event);

           // Look for Incoming call event
           if(event eq PKX_EVENT_GROUP_INCOMING_CALL)
              // Forces the Disconnect signal to cause
              // immediate jump to onsignal function
```

```
            PKCallUseSignal(group,channel);

            // When we get INCOMING CALL then retreive the call info
            x=PKCallGetInfo(group,chan);

            // Now get the parameters that we want (To, From, Display)
            x=PKCallGetParm(group,chan,PK_PARM_TO,&From);
            x=PKCallGetParm(group,chan,PK_PARM_FROM,&To);
            x=PKCallGetParm(group,chan,PK_PARM_DISPLAY,&Display);

            // Display these to application log
            applog("INFO FROM   =",From," To=",To," Display=",Display);

            // Accept the call
            x=PKCallAccept(group,chan);

         // Look for Call Accepted Event...
         else if(event eq PKX_EVENT_CALL_ACCEPTED)
             // Answer the call once we get Call Accepted event
             x=PKCallAnswer(group,chan);
         // When we get call Answered event then we break the loop
         else if(event eq PKX_EVENT_CALL_ANSWERED)
             break;
         endif endif endif
      else if(x eq 0)
         applog("group=",group," chan=",chan," No event recieved");
      endif endif
      // Prevent tight loop (to allow windows events to be processed)
      sleep(1);
   endwhile

   // Call answered
   applog("group=",group," chan=",chan," Call Answered...");

   // Simply loop forever playing HELLOWORLD.VOX until
   // DISCONNECT causes jump to onsignal..
   while(1)
      // Play a prompt (DEMO.VOX)
      x=PKChanPlay(group,chan,filename);

      // Wait for up to 3 DTMF digits..
      x=PKChanWaitDTMF(group,chan,3,40,40);

      // Retrieve the entered DTMF digits
      input=PKChanGetDTMF(group,chan);
      applog("PKChanGetDTMF returned input=",input);

       // Prevent tight loop (to allow windows events to be processed)
      task_sleep(1);
   endwhile

endmain

//*********************************************
//******* THIS IS THE ONSIGNAL FUNCTION ********
//*********************************************
onsignal

int state;

   applog("group=",group," chan=",chan," IN ONSIGNAL!");

   // Disconnect our end of the call..
   x=PKCallHangup(group,chan,0);
   // Wait for channel to go idle
   while(1)
       x=PKCallState(group,chan,&state);
       applog("group=",group," chan=",chan," state=",state);
```

```
       if(state eq PKX_CALL_STATE_IDLE)
           // Release the call resources..
           x=PKCallRelease(group,chan);
           break;
       endif
       // Prevent tight loop (to allow windows events to be processed)
       task_sleep(1);
   endwhile

   // Restart the program to wait for another call..
   restart;
endonsignal
```

-o-

Terminating Events

Many of the media processing functions such as *PKChanPlay()* and *PKChanRecord()* will cause the calling task to block until the function completes with a terminating event (unless *PKChanMode()* is called to allow non-blocking functionality).

The list of blocking functions for which terminating events apply are listed below:

PKChanPlay(vox_chan,filename[,encoding,sample_rate])
PKChanPlayh(vox_chan,filehandle[bytes,encoding,sample_rate])
PKChanRecord(vox_chan,filename,[timeout_ms,silence_ms,encoding,sample_rate,beep])
PKChanRecordh(vox_chan,filehandle,[timeout_ms,silence_ms,encoding,sample_rate,beep])
PKChanWaitDTMF(vox_chan,max_tones,first_delay10ths,inter_delay10ths[,term_digits])
PKChanPlayTone(vox_chan,toneid,duration_ms)
PKChanPlayDigits(vox_chan,digit_str,[inter_delay_ms,dig_dur_ms])

The underlying PIKA function calls for all of the above functions require that a PKX_TTermCond structure be passed to the function. This structure is as follows:

```
typedef struct {
  PK_U32 digitMask;
  PK_INT maxDigits;
  PK_INT timeout;
  PK_INT initialSilenceTimeout;
  PK_INT silenceTimeout;
  PK_INT nonSilenceTimeout;
  PK_INT interDigitTimeout;
} PKX_TTermCond;
```

Each Channel maintains its own global copy of this structure which is passed to each of the above blocking media processing functions (plus any others that are specified by the specific function arguments).

On start-up the only global termination condition to be set is the maxDigits field which is set to 1. This means, by default, all of the above asyncronous functions will terminate when a single DTMF digit is received (however this may be overridden by a function specific argument where relevant).

Some of the values in this channel specific global structure can be set using the PKChanTermDTMF(), PKChanTermTimeout() and PKChanTermNonSil() functions. All the other fields are set depending on the function and the possible termination conditions that are valid for that function.

For example, for the PKChanWaitDTMF *(vox_chan,max_tones,first_delay10ths,inter_delay10ths[,term_digits])* function, prior to making the call the global, channel specific **PKX_TTERMCond** structure is copied to a local structure, then the *maxDigits, initialSilenceTimeout, InterDigitTimeout* and *digitMask* fields are overwritten by the values specified by the *max_tones,first_delay10ths,inter_delay10ths and term_digits* arguments passed to the function.

Below is a description of each of the fields in the PKX_TTermCond structure and the functions that each is relevant to:

Members	Description
digitMask	Terminate the media function when one of the digits specified in the mask is received (All media functions).
maxDigits	Terminate the media function when the specified number of digits have been received (All media functions).
timeout	Terminate the media function when the specified amount of time, in milliseconds, has passed (All media functions).
initialSilenceTimeout	Terminate the media function when the specified amount of initial silence (no digits for CollectDigits or no voice for Record), in milliseconds, has passed (Record and CollectDigits media functions).
silenceTimeout	Terminate the media function when the specified amount of silence, in milliseconds, has passed after receiving voice (Record media function).
nonSilenceTimeout	Terminate the media function when the specified amount of non-silence (digits), in milliseconds, has been received (All media functions).
interDigitTimeout	Terminate the media function when the specified amount of silence, in milliseconds, after receiving a digit has passed (CollectDigits media function).

When a blocking media processing function returns the reason for the termination of the function is passed back as the function return value. Below is shown the list of termination reason values as defined in the **pika.inc** header file:

```
# Terminating events
#define TERM_ERROR        -1
#define TERM_TONE         1
#define TERM_MAXDTMF      2
#define TERM_TIMEOUT      3
#define TERM_INTERDELAY   4
#define TERM_SILENCE      5
#define TERM_ABORT        6
#define TERM_EODATA       7
#define TERM_PLAYTONE     8
#define TERM_PLAYDIGITS   9
#define TERM_PLAYCPTONE   10
#define TERM_INITSILENCE  11
#define TERM_NONSIL       12
```

The following table gives a description of each of these terminating events and the function for which they apply:

Event Name	Description	Applies to functions
TERM_ERROR	An error of some kind was encountered	ALL
TERM_TONE	The function was	ALL

	terminated by a DTMF digit listed in the set of specific DTMF digits	
TERM_MAXDTMF	The total number of DTMF digits requested has been received	ALL
TERM_TIMEOUT	A timeout has occurred	ALL
TERM_INTERDELAY	The specified inter-digit delay timeout has occurred	PKChanWaitDTMF();
TERM_SILENCE	The specified period of silence has occurred	PKChanRecord(); PKChanRecordh();
TERM_ABORT	The function was aborted by SMabort()	ALL
TERM_EODATA	End of file or data has been reached.	PKChanPlay(); PKChanPlayh(); PKChanRecord(); PKChanRecordh();
TERM_PLAYTONE	The specified tone has finished playing	PKChanPlayTone()
TERM_PLAYDIGITS	All specified digits have been played	PKChanPlayDTMF()
TERM_INITSILENCE	Initial silence period exceeded (Record and WaitDTMF)	PKChanWaitDTMF(); PKChanRecord(); PKChanRecordh();
TERM_NONSILENCE	Period of non-silence exceeded	ALL

Note: When called in blocking mode (the default mode for a channel), all of the above functions will also be terminated whenever a hangup signal is received that causes a jump to the *onsignal* function. However there is no specific terminating event code for this type of termination since the return value from the function can never be retrieved when a hangup signal is received, since the program execution will immediately jump to the *onsignal* routine.

When in non-blocking mode (as specified by the *PKChanBlockMode*() function), the above functions will not automatically be terminated by a jump to the *onsignal* function. When in non-blocking mode the asyncronous media operation will continue uninterrupted and must be manually aborted using the *PKChanabort()* function (or the application must manually wait for the function to complete by looping on the *PKChanBusyState*() function call to wait for the state to return to 0 to indicate that the asynchronous media function has completed..

-o-

Blocking and Non-blocking Mode

All of the above *asyncronous media functions* can be called in both blocking or non-blocking mode as specified by a call to PKChanBlockMode*(group,chan,block_mode)*.

Under normal circumstances any call to the above blocking functions will not return until the media function has completed or been terminated by a terminating event. For example a call to PKChanPlay() will not return until the play has completed or has been interupted by a terminating condition such as a DTMF digit being received.

However sometime it is useful to allow the function to return immediately whilst the media processing task is completed in the background so that the program can carry on with some other task. For example, for a database look-up that might take a significant period of time it is useful to be able to start playing the music but for the PKChanPlay() function to return immediately so that the database look-up can be carried out whilst the music is still playing. Once the database lookup has completed then the program can abort the play and continue.

In order to allow this the voice channel can be put into non-blocking mode using the PKChanBlockMode*()* function. If the *block_mode* argument is set to a non zero value then any calls to the above *asyncronous media functions* will return immediately whilst the play, record, play tone etc proceeds in the background.

In this case it is up to the application to ensure that the current *asyncronous media functions* has finished before attempting to call one of the other blocking speech functions. To do this there is a function PKChanBusystate*(group,chan)* which returns the current function that running on the channel at the present time or 0 if there are no speech functions currently running.

NOTE: if *block_mode* is set to 1 then the PKChanBlockMode*()* call will only apply to the next blocking speech function call. Once that function has completed then the channel mode will be set back to blocking mode. If a non-zero value other than 1 is given then the channel will stay in non-blocking mode until a call to *PKChanBlockMode()* is made again with *block_mode* set to 0 to put the channel back into non-blocking mode.

There are three constants defined in ACULAB.INC for this purpose as shown below:

```
const PK_MODE_BLOCKING                =0;
const PK_MODE_NONBLOCKING_ONCEONLY    =1;
const PK_MODE_NONBLOCKNG              =2;
```

For example, the following code extract will play some music in the background whilst a database look-up occurs. Once the database lookup has completed the application will abort the music and wait for the channel to return to idle.

```
    // Prevent DTMF tones from interrupting playback
    PKChanTermDTMF(group,chan,0);

    // Play "Please wait while we look up the information"
    PKChanPlay(group,chan,"PLSWAIT.VOX");

    // Change the mode to play in the background (non-blocking) for the next speech function only
(non-blocking_flag=1)
    PKChanBlockMode(group,chan,PK_MODE_NONBLOCKING_ONCEONLY);
    // Play music in the backgroundwhile information is retrieved
```

```
    PKChanPlay(group,chan,"MUSIC.VOX");

    // Do the data retrieval whilst music is playing
    data_retrieval_func();

    // Abort the music
    PKChanAbort(group,chan);
    // Loop waiting for chan state to return to 0 (should only take milliseconds..)
    while(PKChanBusyState(group,chan))
          task_sleep(1);
    endwhile

    // Allow DTMF tones to interrupt SMplay() etc again..
    PKChanTermDTMF(group,chan,1);

...

etc

// Remember that is a jump to onsignal occurs in non-blocking mode then the play will continue
// in the background and it is up to the programmer to abort and/or wait for the play to finish
    onsignal
          // Check if hangup received during music playback (or other non-blocking operation
          if(PKChanBusyState(group,chan))
                PKChanAbort(group,chan);
                // Loop waiting for chan state to return to 0 (should only take milliseconds..)
          while(PKChanBusyState(group,chan))
                task_sleep(1);
          endwhile
          endif

    ..
    endonsignal
```

-o-

Call Control Library Quick Reference

PKGroupTrace(group, tracelevel);
PKCallTrace(group, channel, tracelevel);
PKCallWait(group,channel,timeout_100ms,&pState);
PKCallWaitAbort(group,channel);
PKCallLastEvent(group, channel, &pEvent);
PKCallState(group, channel, &pState);
PKCallAccept(group, channel);
PKCallAnswer(group, channel);
PKCallReject(group, channel);
PKCallHangup(group, channel,cause);
PKCallRelease(group, channel);
PKCallGetInfo(group, channel);
PKCallGetParm(group, channel,parm_id,&pValue);
PKCallSetParm(group, channel,parm_id,Value);
PKCallClrParm(group, channel);
PKCallMake(group, channel);
PKCallUseSignal(group, channel[,flag]);
PKCallEarlyMedia(group, channel);
PKCallSendInfo(group, channel,info_str1[,info_str2,[...]]);
PKCallMediaBridge(group1, channel1,group2,channel2[,recapture]);

-o-

Media Processing Library Quick Reference

PKChanTrace(group, channel, tracelevel)
PKChanState(group, channel, &pState)
PKChanLastEvent(group, channel, &pEvent)
term_code=PKChanPlay(group,channel,filename[,encoding,sample_rate])
term_code=PKChanPlayh(group,chan,filehandle[bytes,encoding,sample_rate])
term_code=PKChanRecord
(group,chan,filename,max_time_ms,max_silence_ms,[encoding[,sample_rate[,beep]]])
term_code=PKChanRecordh
(group,chan,file_handle,max_time_ms,max_silence_ms,[encoding[,sample_rate[,beep]]])
PKChanTermDTMF(group, channel,max_digits[,digit_mask])
PKChanTermTimeout(group, channel,timeout_ms)
PKChanTermNonSil(group, channel,timeout_ms)
PKChanToneCtl(group, channel,DTMF_Detect, Tone_detect)
term_code=PKChanWaitDTMF
(group,channel,max_tones,first_delay10ths,inter_delay10ths[,term_digits])
PKChanClearDTMF(group, channel)
digits=PKChanGetDTMF(group, channel[,max_digits)
PKChanAbort(group, channel)
functionID=PKChanBusyState(group, channel);
PKChanBlockMode(group, channel,block_mode);

-o-

PIKA HMP Call Control Function Reference

PKGroupTrace

Synopsis:

PKGroupTrace(group, tracelevel)

Arguments:

group – The logical group number.
tracelevel – 0 turns trace off, 1-Trace function calls, 2-Also trace events

Description: This function switches on or off the tracing of all function and events for a particular *group*. If *tracelevel* is set to 1 then only function calls are traced, whereas if the *tracelevel* is set to 2 then both function calls and events are traced.

If group trace is switched on the both Call and Channel functions (and events) will be traced for this group.

Trace will be written to the Telecom Engine trace log.

Returns: 0 upon success or -1 if a bad group or channel was provided.

-o-

PKCallTrace

Synopsis:

PKCallTrace(group, channel, tracelevel)

Arguments:

group – The logical group number.
channel – The channel number.
tracelevel – 0 turns trace off, 1-Trace function calls, 2-Also trace events

Description: This function switches on or off the tracing of all call control functions and events (PXK_EVENT_CALL_xxx). If *tracelevel* is set to 1 then only function calls are traced, whereas if the *tracelevel* is set to 2 then both function calls and Events are traced. Trace will be written to the Telecom Engine trace log.

Returns: 0 upon success or -1 if a bad group or channel was provided.

-o-

PKCallWait

Synopsis:

PKCallWait(group,channel,timeout_100ms,&pState)

Arguments:

group – The logical group number.

channel – The channel number.

pState – Pointer to a variable to receive the ID of the event that terminated the PKCallWait() call.

Description: This function will wait for an event to be received on a particular *group* and *channel*. It will return either when an event is detected on the channel or the timeout has expired (or if it is aborted by PKCallWaitAbort()). The timeout is specified in 100ms units (tenths of a second) after which the function will return if no event has been received. If -1 (PK_WAIT_FOREVER) is specified for the timeout then the call will wait forever for an event.

The function takes a pointer to a variable which will hold the ID of the event received.

Note that the function keeps an internal track of events on a channel and if the state of a channel has changed since the last time it was called then it will return immediately with the current state of the channel. This is to prevent events from being missed in between calls to PKCallWait() but it means that the programmer should always check the returned state in a loop to ensure that the expected event is received. For example:

```
// loop waiting for incoming call
while(1)
    x=PKCallWait(group,chan,PK_WAIT_FOREVER,&event);
    // Make sure the event we want is received..
    if(x > 0)
      if(event eq PK_EVENT_GROUP_INCOMING_CALL)
        // Accept the call
        PKCallAccept(group,chan);
      else if(event eq PK_EVENT_CALL_ACCEPTED)
        // Answer the call
        PKCallAnswer(group,chan);
      else if(event eq PK_EVENT_CALL_ANSWERED)
        break;
      endif endif endif
    endif
    task_sleep(1);
endwhile


// Play a vox file to caller
PKChanPlay(group,chan,filename);

// Hangup the call
PKCallHangup(group,chan,PK_CAUSE_NORMAL);

etc..
```

Returns: The function will return 0 if the timeout has expired without receiving an event (or if the function was aborted by a PKCallWaitAbort() call). It will return 1 if the function terminated because an event was detected (and the event ID will be set in the variable pointed to by *pState*). It will return -1 is a bad group or channel was given.

-o-

PKCallWaitAbort

Synopsis:

PKChanWaitAbort(group, channel)

Arguments:

group – The logical group number.
channel – The channel number.

Description: This function will abort any currently active call to PKCallWait() that is blocking and waiting for an event to arrive. This function would typically be called from another task to stop a PKCallWait() in order that some other event could be handled (such as graceful shutdown of the system).

The PKCallWait() function will return with a return value of 0 to indicate that no event was actually received.

Returns: 0 upon success or a negative error code.

-o-

PKCallLastEvent

Synopsis:

PKCallLastEvent(group, channel, &pEvent)

Arguments:

group – The logical group number.
channel – The channel number.
pEvent – Pointer to the variable that will receive the last event

Description: This function sets the variable pointed to by the *pEvent* argument to the value of the last event recieved by the active call on on the specified *group* and *channel*. The event values returned can be one of the following as defined in the **pika.inc** include file:

```
const PKX_EVENT_GROUP_INCOMING_CALL             =0x6100;
const PKX_EVENT_GROUP_INCOMING_TRANSFER                   =0x6101;
const PKX_EVENT_CALL_ACCEPTED               =0x6180;
const PKX_EVENT_CALL_ANSWERED                   =0x6181;
const PKX_EVENT_CALL_DIALING                =0x6182;
const PKX_EVENT_CALL_PROCEEDING                 =0x6183;
const PKX_EVENT_CALL_ALERTING               =0x6184;
const PKX_EVENT_CALL_CONNECTED                  =0x6185;
const PKX_EVENT_CALL_DROPPED                =0x6186;
const PKX_EVENT_CALL_DISCONNECTED               =0x6187;
const PKX_EVENT_CALL_ANALYSIS_DETECTED_MODEM_FAX          =0x6188;
const PKX_EVENT_CALL_ANALYSIS_DETECTED_ANSWERING_MACHINE      =0x6189;
const PKX_EVENT_CALL_ANALYSIS_DETECTED_LIVE_PERSON        =0x618A;
const PKX_EVENT_CALL_ANALYSIS_DETECTED_SIT_MESSAGE        =0x618B;
const PKX_EVENT_CALL_ANALYSIS_DETECTED_SILENCE       =0x618C;
const PKX_EVENT_CALL_ANALYSIS_DETECTED_UNKNOWN            =0x618D;
 const PKX_EVENT_CALL_ANALYSIS_DETECTED_ANSWERING_MACHINE_END   =0x618E;
const PKX_EVENT_CALL_HELD                   =0x6190;
const PKX_EVENT_CALL_RESUMED                =0x6191;
const PKX_EVENT_CALL_EARLY_MEDIA                =0x6192;
const PKX_EVENT_CALL_TRANSFER_COMPLETED              =0x6193;
const PKX_EVENT_CALL_TRANSFER_FAILED            =0x6194;
const PKX_EVENT_CALL_INFO_UPDATED                =0x6195;
const PKX_EVENT_CALL_TASK_FAILED                =0x61ff;
```

Returns: 0 upon success or -1 if a bad group or channel was provided.

-o-

PKCallState

Synopsis:

PKCallState(group, channel, &pState)

Arguments:

group – The logical group number.
channel – The channel number.
pState – Pointer to a variable that will hold the returned state

Description: This function sets the variable pointed to by the *pState* argument to the current state of the channel as returned from the PIKA PKX_CALL_GetState() function on the specified *group* and *channel.* The state values returned can be one of the following as defined in the **pika.inc** include file:

```
const  PKX_CALL_STATE_IDLE=0;      // Call is idle or non-existent.
const  PKX_CALL_STATE_INITIATED=1;   // Outgoing call was started but dialing has not commenced
yet.
const  PKX_CALL_STATE_DIALING=2;    // Outgoing call is dialing address information.
const  PKX_CALL_STATE_PROCEEDING=3;    // Outgoing call has transmitted all necessary address
information.
const  PKX_CALL_STATE_ALERTING=4;    // Incoming or outgoing call is alerting (ringing).
const  PKX_CALL_STATE_DETECTED=5;    // Incoming call detected and address information is being
retrieved (internal state only).
const  PKX_CALL_STATE_OFFERED=6;     // Incoming call is being offered to the user application.
const  PKX_CALL_STATE_CONNECTED=7;    // Incoming or outgoing call is connected.
const  PKX_CALL_STATE_DISCONNECTED=8;   // Incoming or outgoing call is disconnected.
const  PKX_CALL_STATE_TRANSFER_OFFERED=9;  // Incoming transfer is being offered to the user
application.
const  PKX_CALL_STATE_BLOCKING_COLLECT_CALL=10;    // Attempting to block an incoming collect
call
```

Returns: 0 upon success or -1 if a bad group or channel was provided.

-o-

PKCallAccept

Synopsis:

PKCallAccept(group, channel)

Arguments:

group – The logical group number.
channel – The channel number.

Description: This function accepts the incoming call on the specified *group* and *channel* after a PKX_EVENT_GROUP_INCOMING_CALL event has been received. This function will result in a PKX_ EVENT_CALL_ACCEPTED event being generated.

This function maps to the following PIKA function:

```
PCK_CALL_Accept(TPikaHandle callHandle );
```

Returns: This function returns 0 upon success or a negative error code.

-o-

PKCallAnswer

Synopsis:

PKCallAnswer(group, channel)

Arguments:

group – The logical group number.
channel – The channel number.

Description: This function answers the incoming call on the specified *group* and *channel* after a PKX_EVENT_GROUP_INCOMING_CALL or PKX_EVENT_CALL_ACCEPTED event has been received. This function will result in a PKX_ EVENT_CALL_ANSWERED event being generated.

This function maps to the following PIKA function:

```
PCK_CALL_Answer(TPikaHandle callHandle );
```

Returns: This function returns 0 upon success or a negative error code.

-o-

PKCallReject

Synopsis:

PKCallRejct(group, channel)

Arguments:

group – The logical group number.
channel – The channel number.

Description: This function rejects the incoming call on the specified *group* and *channel* after a PKX_EVENT_GROUP_INCOMING_CALL event has been received. Example reasons for rejecting a call are:

- Called party number is unknown to the application,
- Application is not ready to handle any more incoming calls, and
- Calling party is on the application's blocked list.

Do not call PKCallAccept() or PKCallAnswer() prior to calling this function. Use the PKCallHangup() function to terminate the call after those functions are called.

Notes:

- Applications must respond to the PKX_EVENT_GROUP_INCOMING_CALL event with one of three functions as soon as possible: PKCallAccept(), PKCallAnswer(), or PKCallReject(). No other function calls are allowed. Failure to respond to the event in a timely fashion (time depends on the protocol, but is typically just a few seconds), results in the incoming call being dropped by the remote side.
- Do not call the PKCallRelease() function after rejecting a call. The PKCallReject() function automatically releases the call resources.

This function maps to the following PIKA function:

```
PCK_CALL_Reject(TPikaHandle callHandle);
```

Returns: This function returns 0 upon success or a negative error code.

-o-

PKCallHangup

Synopsis:

PKCallHangup(group, channel,cause)

Arguments:

group – The logical group number.
channel – The channel number.
cause - The handug cause code

Description: This function performs a local side call hangup on the specified *group* and *channel.* The *cause* argument can be set to one of the following values as defined in he **pika.inc** header file:

```
const  PKX_CALL_DROP_CAUSE_NORMAL=0;
const  PKX_CALL_DROP_CAUSE_BUSY=1;
const   PKX_CALL_DROP_CAUSE_FASTBUSY=2;
const   PKX_CALL_DROP_CAUSE_REORDER=3;
const   PKX_CALL_DROP_CAUSE_NOANSWER=4;
const   PKX_CALL_DROP_CAUSE_NODIALTONE=5;
const  PKX_CALL_DROP_CAUSE_RESET=6;
const   PKX_CALL_DROP_CAUSE_TRANSFER=7;
const   PKX_CALL_DROP_CAUSE_REJECTED=8;
const   PKX_CALL_DROP_CAUSE_FAILED=9;
const   PKX_CALL_DROP_CAUSE_NOT_FOUND=10;
const  PKX_CALL_DROP_CAUSE_UNAUTHORIZED=11;
const   PKX_CALL_DROP_CAUSE_ADDRESS_CHANGED=12;
const   PKX_CALL_DROP_CAUSE_ADDRESS_INCOMPLETE=13;
const   PKX_CALL_DROP_CAUSE_CONGESTION=14;
const  PKX_CALL_DROP_CAUSE_BAD_REQUEST=15;
const   PKX_CALL_DROP_CAUSE_NETWORK_TIMEOUT=16;
const   PKX_CALL_DROP_CAUSE_NOT_IMPLEMENTED=17;
const   PKX_CALL_DROP_CAUSE_NOT_ACCEPTABLE=18;
const   PKX_CALL_DROP_CAUSE_RESOURCE_UNAVAILABLE=19;
const   PKX_CALL_DROP_CAUSE_SERVICE_UNAVAILABLE=20;
const   PKX_CALL_DROP_CAUSE_NETWORK_OUT_OF_ORDER=21;
const   PKX_CALL_DROP_CAUSE_SESSION_TIMER_EXPIRY=22;
const   PKX_CALL_DROP_CAUSE_UNKNOWN=23;
```

This function operates asynchronously. The PKX_EVENT_CALL_DROPPED event is raised if the function completes successfully. The PKX_EVENT_CALL_TASK_FAILED event is raised if the function fails for some reason.

The PKCallRelease() function must be called after the PKX_EVENT_CALL_DROPPED event is received to clean up the call resources.

This function maps to the following PIKA function:

PCK_CALL_Drop(`TPikaHandle callHandle,PK_TCallDropCause cause`);

Returns: This function returns 0 upon success or a negative error code.

-o-

PKCallRelease

Synopsis:

PKCallRelease(group, channel)

Arguments:

group – The logical group number.
channel – The channel number.

Description: This function releases the call on the specified *group* and *channel* and frees up the call handle.

This function operates synchronously. It must be the last function executed on a call. The PKX_EVENT_CALL_DISCONNECTED or PKX_EVENT_CALL_DROPPED event must be received prior to calling this function (i.e the call state should be PK_CALL_STATE_IDLE as returned by the PKCallState() function).

Note: Do not use this function on an incoming call handle that has been rejected with the PKCallReject() function. The PKCallReject() function releases the call resources automatically.

This function maps to the following PIKA function:

```
PCK_CALL_Release(TPikaHandle callHandle);
```

Returns: This function returns 0 upon success or a negative error code.

-o-

PKCallGetInfo

Synopsis:

PKCallGetInfo(group, channel)

Arguments:

group – The logical group number.
channel – The channel number.

Description: This function is used to retreive the addressing information for the call on the specified *group* and *channel* usually after a PKX_GROUP_EVENT_INCOMING_CALL event has been received (It can be used on outgoing calls but its use here is somewhat redundant). A copy of the **PKX_TCallInfo** structure is stored in the internal channel array and the the individual addressing parameters can then be retreived using the PKCallGetParm() function.

This function maps to the following PIKA function:

```
PCK_CALL_GetInfo(TPikaHandle callHandle );
```

Returns: This function returns 0 upon success or a negative error code.

-o-

PKCallGetParm

Synopsis:

PKCallGetParm(group, channel,parm_id,&pValue)

Arguments:

group – The logical group number.
channel – The channel number.
parm_id - The ID of the addressing parameter to retrieve
pValue - Pointer to the variable that will hold the returned addressing value..

Description: This function retrieves the specific addressing parameter value of the parameter specified by *parm_id* for the channel specified by *group* and *channel*. This function should be called after the addressing information structure has been retrieved by a call to the PKCallGetInfo () function. The value of the parameter will be returned in the variable pointed to by the *pValue* argument.

The *parm_id* argument should be set to one of the following values as defined in the **pika.inc** header file:

```
const PK_PARM_TYPE_TO                  =0;
const PK_PARM_TYPE_FROM           =1;
const PK_PARM_TYPE_FORWARDEDFROM     =2;
const PK_PARM_TYPE_DISPLAY             =3;
const PK_PARM_TYPE_CUSTOM              =4;
const PK_PARM_TYPE_CATEGORY       =5;
const PK_PARM_TYPE_TOLLCATEGORY        =6;
const PK_PARM_TYPE_NUMRESTRICTED =7;
```

Returns: This function returns 0 upon success or a negative error code.

-o-

PKCallSetParm

Synopsis:

PKCallSetParm(group, channel,parm_id,Value)

Arguments:

group – The logical group number.
channel – The channel number.
parm_id - The ID of the addressing parameter to retrieve
Value - The value to set the parameter to .

Description: This function allows the addressing and other channel parameters to be set prior to making an outbound call on the specified *group* and *channel*. The *parm_id* field specifies which parameter is to be set and the *Value* argument is the value to set that parameter to.

The *parm_id* argument should be set to one of the following values as defined in the **pika.inc** header file:

```
const PK_PARM_TYPE_TO                   =0;
const PK_PARM_TYPE_FROM           =1;
const PK_PARM_TYPE_FORWARDEDFROM     =2;
const PK_PARM_TYPE_DISPLAY              =3;
const PK_PARM_TYPE_CUSTOM               =4;
const PK_PARM_TYPE_CATEGORY       =5;
const PK_PARM_TYPE_TOLLCATEGORY         =6;
const PK_PARM_TYPE_NUMRESTRICTED =7;

const PK_PARM_TYPE_TIMEOUT              =16;
const PK_PARM_TYPE_CA_ENABLE        =17;
const PK_PARM_TYPE_CA_TYPE          =18;
const PK_PARM_TYPE_CA_DEBOUNCEON        =19;
const PK_PARM_TYPE_CA_DEBOUNCEOFF     =20;
const PK_PARM_TYPE_CA_LIVEWORDS   =21;
const PK_PARM_TYPE_CA_MAXDURATION     =22;
const PK_PARM_TYPE_CA_SPEECHENDTIME   23;
const PK_PARM_TYPE_CA_LIVEMAXTIME =24;
const PK_PARM_TYPE_CA_LIVETOTTIME =25;
const PK_PARM_TYPE_CA_ANSWMACHINEENDTIME=26;
```

The first group of parameters (from 0 to 7) relate to the fields of the **PKX_TCallInfo** structure passed in the PIKA PKX_CALL_Make() function, whereas the second group of parameters (from 16 onwards) relate to the fields of the
PKX_TCallSettings structure passed in the PIKA PKX_CALL_Make() function..

Before making a call, at the very least it is necessary to set the PK_PARM_TYPE_TO field to the destination address.

Returns: This function returns 0 upon success or a negative error code.

-o-

PKCallClrParm

Synopsis:

PKCallClrParm(group, channel)

Arguments:

group – The logical group number.
channel – The channel number.

Description: This function clears the internal **PKX_TCallInfo** and **PKX_TCallSettings** structures related to the given *group* and *channel.* This function can be called prior to calling the PKCallSetParm() function to clear any values previous set by calls to this function.

Returns: This function returns 0 upon success or a negative error code.

-o-

PKCallMake

Synopsis:

PKCallMake(group, channel)

Arguments:

group – The logical group number.
channel – The channel number.

Description: This function initiates an outbound call on the *group* and *channel*. Before making this call the addressing information for the call should be set up using the PKCallSetParm() function with, at the very least, the PK_PARM_TYPE_TO parameter being set to define the destination address..

```
// Set up the outbound call parameters..
PKCallClrParm(group,channel);
 PKCallSetParm(group,channel,PK_PARM_TYPE_TO,"123@192.168.2.6");
 PKCallSetParm(group,channel,PK_PARM_TYPE_FROM,"192.168.2.3");
 PKCallSetParm(group,channel,PK_PARM_TYPE_DISPLAY,"Joe Bloggs");

// Initiate the outbound call
x=PKCallMake(group,channel);
```

This function operates asynchronously. The PKX_EVENT_CALL_CONNECTED or PKX_EVENT_CALL_DISCONNECTED event is raised if the function completes successfully. The PKX_EVENT_CALL_TASK_FAILED event is raised if the function fails for some reason.

This function maps to the following PIKA function:

```
PCK_CALL_Make(TPikaHandle callHandle,PKX_TCallInfo * info,PKX_TCallSettings * setting);
```

Returns: This function returns 0 upon success or a negative error code.

-o-

PKCallUseSignal

Synopsis:

PKCallUseSignal(group, channel[,flag])

Arguments:

group – The logical group number.
channel – The channel number.
[flag] – Set to 1 (default) to cause task to jump to *onsignal* if a PKX_EVENT_CALL_DISCONNECTED event is received. Set to 0 to stop task jumping to *onsignal*.

Description: This function allows the current Telecom Engine task to be associated with a *group* and *channel* in such a way that if a call on the specified *port* and *channel* receives a PKX_EVENT_CALL_DISCONNECTED event then the task will be forced to jump immediately to its *onsignal* function. The default value of *flag* if it is not specified is 1. To clear the association between the task and a *port* and *channel* so that it will no longer jump to the *onsignal* function upon receiving a PKX_EVENT_CALL_DISCONNECTED event then the *flag* should be set to 0.

If this call is made when the last event received on the *group* and *channel* was the PKX_EVENT_CALL_DISCONNECTED (i.e the Disconnect signal had already arrived) then this will cause the program to immediately jump to its *onsignal* function.

This function cannot be called unless there is a valid call handle on the specified *group* and *channel* (i.e. after a PKX_EVENT_GROUP_INCOMING_CALL has been received or an outbound call initiated with a PKCallMake() function call).

Returns: This function returns 0 upon success or a negative error code. code.

-o-

PKCallEarlyMedia

Synopsis:

PKCallEarlyMedia(group, channel)

Arguments:

group – The logical group number.
channel – The channel number.

Description: This function informs the remote party on the call specified by *group* and *channel* that audio information is present on the channel.

This function maps to the following PIKA function:

PCK_CALL_EalryMedia(`TPikaHandle callHandle`);

Returns: This function returns 0 upon success or a negative error code.

-o-

PKCallSendInfo

Synopsis:

PKCallSendInfo(group, channel,info_str1[,info_str2,[...]])

Arguments:

group - The logical group number.
channel - The channel number.
info_str1 - The information string
[info_str2[, info_str3...]] - Optional additional information strings that will be concatenated to info_str1

Description: This function informs the remote party on the call specified by *group* and *channel,* of additional protocol specific information as specified by the information strings info_str1, info_str2 etc. Only SIP channels are currently supported.

For SIP channel, issuing this function results in an INFO message being sent to the remote call party. The info buffer parameter is broken up into "name=value" fields separated by a vertical bar "|" character. Values with a vertical bar in them must be enclosed in double quotes. The following field names are supported:

Field Name	Description
headers	Additional message headers
type	Type of payload being included
payload	Payload contents

Note that at if multiple information strings are specified then these strings are concatenated together to make a single information string that will be sent to the remote party. Since these strings are concatenated without modification care should be taken to leave spaces between the end of one string and the start of the next if necessary.

This function maps to the following PIKA function:

```
PCK_CALL_Info(TPikaHandle callHandle,PKCHAR * Info);
```

Returns: This function returns 0 upon success or a negative error code.

-o-

PKCallMediaBridge

Synopsis:

PKCallMediaBridge(group1, channel1,group2,channel2[,recapture])

Arguments:

group1 – The first logical group number.
channel1 – The first channel number.
group2 – The second logical group number.
channel2 – The second channel number.
[recapture] - Option recapture flag (default=0)

Description: This function will join the media paths between two calls specified by group1, channel2 and group2,channel2, freeing up the local channel resources. The call signaling will still be controlled by the application.

The *recapture* flag defines whether the media should be pushed out to the network (0) or returned back to the application (1).

This function maps to the following PIKA function:

PCK_CALL_MediaBridge(`TPikaHandle callHandle,TPikaHandle OthercallHandle, PK_BOOL recapture`);

This function will not interfere with the signaling path of either call. Only the media path is changed.

Media bridging is only valid for calls that use SIP channels.

The media connection between the two calls will be broken when pushing the media to the network. This connection will NOT be re-established when recapturing the media to the application. The application will be responsible for calling PKX_CHANNEL_FullDuplexConnect if desired.

Returns: This function returns 0 upon success or a negative error code.

-o-

PIKA HMP Media Processing Function Reference

PKChanTrace

Synopsis:

PKChanTrace(group, channel, tracelevel)

Arguments:

group – The logical group number.
channel – The channel number.
tracelevel – 0 turns trace off, 1-Trace function calls, 2-Also trace events

Description: This function switches on or off the tracing of all channel (Media) function and events (PXK_EVENT_CHANNEL_xxx).. If *tracelevel* is set to 1 then only function calls are traced, whereas if the *tracelevel* is set to 2 then both function calls and events are traced. Trace will be written to the Telecom Engine trace log.

Returns: 0 upon success or -1 if a bad group or channel was provided.

-o-

PKChanState

Synopsis:

PKChanState(group, channel, &pState)

Arguments:

group – The logical group number.
channel – The channel number.
pState – Pointer to a variable that will hold the returned state

Description: This function sets the variable pointed to by the *pState* argument to the current state of the channel as returned from the PIKA PKX_CHANNEL_GetState() function on the specified *group* and *channel.* The state values returned can be one of the following as defined in the **pika.inc** include file:

```
const PKX_CHANNEL_STATE_DOWN=0;
const PKX_CHANNEL_STATE_READY=1;
const PKX_CHANNEL_STATE_IN_USE=2;
```

Returns: 0 upon success or -1 if a bad group or channel was provided.

-o-

PKChanLastEvent

Synopsis:

PKChanLastEvent(group, channel, &pEvent)

Arguments:

group – The logical group number.
channel – The channel number.
pEvent – Pointer to the variable that will receive the last event

Description: This function sets the variable pointed to by the *pEvent* argument to the value of the last channel event recieved on the specified *group* and *channel*. The event values returned can be one of the following as defined in the **pika.inc** include file:

```
const PKX_EVENT_CHANNEL_DOWN           =0x6020;
const PKX_EVENT_CHANNEL_READY          =0x6021;
const PKX_EVENT_CHANNEL_IN_USE         =0x6022;
const PKX_EVENT_CHANNEL_PHONE_OFFHOOK       =0x6023;
const PKX_EVENT_CHANNEL_PHONE_ONHOOK        =0x6024;
const PKX_EVENT_CHANNEL_PHONE_HOOKFLASH         =0x6025;
const PKX_EVENT_CHANNEL_HOOKFLASH           =0x6025;
const PKX_EVENT_CHANNEL_DONE_RINGING        =0x6026;
const PKX_EVENT_CHANNEL_DTMF_START         =0x6040;
const PKX_EVENT_CHANNEL_DTMF           =0x6040;
const PKX_EVENT_CHANNEL_DTMF_END          =0x6043;
const PKX_EVENT_CHANNEL_TONE_ON          =0x6041;
const PKX_EVENT_CHANNEL_TONE_OFF          =0x6042;
const PKX_EVENT_CHANNEL_DONE_TONE_GENERATION    =0x6045;
const PKX_EVENT_CHANNEL_DONE_PLAY         =0x6046;
const PKX_EVENT_CHANNEL_STOPPED_TONE_GENERATION  =0x6047;
const PKX_EVENT_CHANNEL_STOPPED_PLAY        =0x6048;
const PKX_EVENT_CHANNEL_STOPPED_RECORD       =0x6049;
const PKX_EVENT_CHANNEL_STOPPED_COLLECT_DIGITS  =0x604a;
const PKX_EVENT_CHANNEL_DIGIT_BUFFER_FULL    =0x604b;
const PKX_EVENT_CHANNEL_TERM_DIGIT_MASK      =0x6050;
const PKX_EVENT_CHANNEL_TERM_MAX_DIGITS      =0x6051;
const PKX_EVENT_CHANNEL_TERM_TIMEOUT        =0x6052;
const PKX_EVENT_CHANNEL_TERM_SILENCE_TIMEOUT   =0x6053;
const PKX_EVENT_CHANNEL_TERM_INTERDIGIT_TIMEOUT =0x6054;
const PKX_EVENT_CHANNEL_TERM_NONSILENCE_TIMEOUT  =0x6055;
const PKX_EVENT_CHANNEL_TERM_INITSILENCE_TIMEOUT=0x6056;
const PKX_EVENT_CHANNEL_DATAREADY_PLAY       =0x6060;
const PKX_EVENT_CHANNEL_UNDERFLOW_PLAY       =0x6061;
const PKX_EVENT_CHANNEL_DATAREADY_RECORD      =0x6062;
const PKX_EVENT_CHANNEL_OVERFLOW_RECORD       =0x6063;
const PKX_EVENT_CHANNEL_FULL_DUPLEX_CONNECT     =0x6064;
const PKX_EVENT_CHANNEL_HALF_DUPLEX_CONNECT     =0x6065;
const PKX_EVENT_CHANNEL_FULL_DUPLEX_DISCONNECT  =0x6066;
const PKX_EVENT_CHANNEL_HALF_DUPLEX_DISCONNECT       =0x6067;
const PKX_EVENT_CHANNEL_SPEECH_ON           =0x6068;
const PKX_EVENT_CHANNEL_SPEECH_OFF          =0x6069;
const PKX_EVENT_CHANNEL_SPEECH_TONE_ON          =0x606A;
const PKX_EVENT_CHANNEL_SPEECH_TONE_OFF         =0x606B;
const PKX_EVENT_CHANNEL_CONTROL_PLAY        =0x606C;
const PKX_EVENT_CHANNEL_DONE_FAX            =0x6070;
const PKX_EVENT_CHANNEL_FAX_STARTED             =0x6071;
const PKX_EVENT_CHANNEL_FAX_TRAINING_SUCCESS    =0x6072;
const PKX_EVENT_CHANNEL_FAX_DOCUMENT_BEGIN          =0x6073;
const PKX_EVENT_CHANNEL_FAX_DOCUMENT_END        =0x6074;
const PKX_EVENT_CHANNEL_FAX_PAGE_BEGIN          =0x6075;
```

```
const PKX_EVENT_CHANNEL_FAX_PAGE_END            =0x6076;
const PKX_EVENT_CHANNEL_FAX_PAGE_SUCCESS      =0x6077;
const PKX_EVENT_CHANNEL_FAX_DISCONNECTING     =0x6078;
const PKX_EVENT_CHANNEL_FAX_DISCONNECTED     =0x6079;
const PKX_EVENT_CHANNEL_FAX_TRAINING       =0x607A;
const PKX_EVENT_CHANNEL_FAX_TRAINING_FAILED =0x607B;
const PKX_EVENT_CHANNEL_FAX_CONTROL_FRAME     =0x607D;
const PKX_EVENT_CHANNEL_STOPPED_FAX             =0x607E;
const PKX_EVENT_CHANNEL_TASK_FAILED       =0x60ff;
```

Returns: 0 upon success or -1 if a bad group or channel was provided.

-o-

PKChanPlay

Synopsis:

term_code=PKChanPlay(group,channel,filename[,encoding,sample_rate])

Arguments:

group – The logical group number.
channel – The channel number.
filename – The filename of the voice prompt to play
[encoding] – The audio encoding type of voice prompt file to play
[sample_rate] – The sample rate of the voice prompt file.

Description: This function plays the speech file specified by *filename* on the given *group* and *channel*.

If the *encoding* and *sample_rate* are not specified then the last encoding and sample_rate used on the channel are assumed. At start-up the default encoding and sample_rate is set to and the sample rate is 6000.

Otherwise the *encoding* and *samperate* can be specifed from one of the types defined in the **pika.inc** header file as follows:

```
const PKX_AUDIO_ENCODING_LINEAR_8BIT          =0x00100000;
const PKX_AUDIO_ENCODING_LINEAR_16BIT         =0x00200000;
const PKX_AUDIO_ENCODING_LINEAR_16BIT_INTEL   =0x00200000;
const PKX_AUDIO_ENCODING_LINEAR_16BIT_MOTOROLA=0x20000000;
const PKX_AUDIO_ENCODING_MU_LAW               =0x00400000;
const PKX_AUDIO_ENCODING_A_LAW                =0x00800000;
const PKX_AUDIO_ENCODING_ADPCM_4BIT_PIKA      =0x01000000;
const PKX_AUDIO_ENCODING_ADPCM_4BIT_DIALOGIC  =0x02000000;
const PKX_AUDIO_ENCODING_ADPCM_3BIT           =0x04000000;
const PKX_AUDIO_ENCODING_GSM_610_MS           =0x10000000;
```

The *sample_rate* is the sample rate in bits per second of the speech file. The valid sample rates are as follows:

```
const PKX_AUDIO_SAMPLING_RATE_4KHZ            =0x00010000;
const PKX_AUDIO_SAMPLING_RATE_6KHZ            =0x00020000;
const PKX_AUDIO_SAMPLING_RATE_8KHZ            =0x00040000;
const PKX_AUDIO_SAMPLING_RATE_11KHZ           =0x00080000;
```

Note that under normal circumstances the Telecom Engine will block the calling task until the playback is terminated by a terminating event of some kind. This may be the presence of a DTMF digit in the DTMF digit buffer for the channel, the end of the file, a call to PKChanAbort() or any other terminating event.

The reason for the function terminating is returned as the return value of the function and may be one of the following values:

```
# PKChanPlay() Terminating events
const PK_TERM_ERROR       -1;
const PK_TERM_TONE        =1;
const PK_TERM_MAXDTMF      =2;
const PK_TERM_TIMEOUT      =3;
```

```
const PK_TERM_ABORT      =6;
const PK_TERM_EODATA       =7;
const PK_TERM_NONSIL       =12;
```

(see Terminating events)

Also whenever a jump to the *onsignal* function occurs (for example caused by a hangup signal after a call to PKCallUseSignal()) and if the function is playing in blocking mode (see Blocking and non-blocking mode) then the PKChanPlay() will automatically be aborted (i.e the speech playback will be stopped). If playing in non-blocking mode then it is up to the application to abort the play and/or wait for it to complete.

Returns: Returns either an error code (E.g. if the file could not be opened) or the reason for the function termination.

-o-

PKChanPlayh

Synopsis:

term_code=PKChanPlayh(group,chan,filehandle[bytes,encoding,sample_rate])

Arguments:

group – The logical group number.
channel – The channel number.
filehandle – A file handle returned from a call to *sys_fhopen()*
[bytes] – Number of bytes to play from the file
[encoding] – The encoding type of voice prompt file to play
[sample_rate] – The sample rate of the voice prompt file.

Description: This function is similar to the PKChanPlay*()*function except that it takes a file handle as returned from the *sys_fhopen()* function in the Telecom Engine standard system library (CXSYS.DLL). It is up to the application to open the file first and to ensure that the file handle is released after use.

If the optional argument *bytes* is specified then the function will terminate with the event TERM_EODATA once the specified number of *bytes* has been played from the file. If *bytes* is omitted or set to 0 then the function will continue playing from the file until the end of the file is reached or another terminating event causes the function to finish.

Just as for the PKChanPlay() function the PKChanPlayh() will return the reason for the termination of the function, which will be one of the following values:

```
# PKChanPlay() Terminating events
const PK_TERM_ERROR       -1;
const PK_TERM_TONE        =1;
const PK_TERM_MAXDTMF      =2;
const PK_TERM_TIMEOUT      =3;
const PK_TERM_ABORT       =6;
const PK_TERM_EODATA       =7;
const PK_TERM_NONSIL       =12;
```

(see Terminating events)

For *encoding* and *sample_rate* values see PKChanPlay().

If the function is playing in blocking mode then a jump to *onsignal* will cause the playback to be aborted. In non-blocking mode the playback will continue even after a jump to *onsignal* and it is then up to the application to abort the playback and/or wait for it to complete.

Example:

```
int fh;
// Open the prompt file..
fh=sys_fhopen("HELLO.VOX","rs");
if(fh < 0)
 errlog("Error opening file: err=",fh);
   task_hangup(task_getpid());
endif

// If we get here then the file is open so play it

PKChanPlayh(group,chan,fh,PKX_AUDIO_ENCODING_LINEAR_8BIT,PKX_AUDIO_SAMPLING_RATE_8
```

```
KHZ);

      // Close the file handle after use..
       sys_fhclose(fh);
```

Returns: Returns either an error code (E.g. if the file could not be opened) or the reason for the function termination.

-o-

PKChanRecord

Synopsis:

term_code=PKChanRecord(group,chan,filename,max_time_ms,max_silence_ms,[encoding[,sample_rate[,beep]]])

Arguments:

group – The logical group number.
channel – The channel numbernel
filename - The filename to record to
[max_time_ms] – Number of millisecond seconds to record
[silence_ms] – Number of milliseconds of silence to end recording
[encoding] – The audio encoding type of voice prompt file to play
[sample_rate] - Optional sample rate

Description: This function records to the given *filename* on the specified *group* and *chan*. The number of millisecods *to* to record is specified by the *max_time_ms* argument. The recording will be terminated if the number of millisecods of silence is detected as specified by *silence_ms*.

If the *encoding* and *sample_rate* are not specified then the last encoding and sample_rate used on the channel are assumed. At start-up the default encoding and sample_rate is set to and the sample rate is 6000.

Otherwise the *encoding* and *samperate* can be specifed from one of the types defined in the **pika.inc** header file as follows:

```
const PKX_AUDIO_ENCODING_LINEAR_8BIT          =0x00100000;
const PKX_AUDIO_ENCODING_LINEAR_16BIT         =0x00200000;
const PKX_AUDIO_ENCODING_LINEAR_16BIT_INTEL   =0x00200000;
const PKX_AUDIO_ENCODING_LINEAR_16BIT_MOTOROLA=0x20000000;
const PKX_AUDIO_ENCODING_MU_LAW               =0x00400000;
const PKX_AUDIO_ENCODING_A_LAW                =0x00800000;
const PKX_AUDIO_ENCODING_ADPCM_4BIT_PIKA      =0x01000000;
const PKX_AUDIO_ENCODING_ADPCM_4BIT_DIALOGIC  =0x02000000;
const PKX_AUDIO_ENCODING_ADPCM_3BIT           =0x04000000;
const PKX_AUDIO_ENCODING_GSM_610_MS           =0x10000000;
```

The *sample_rate* is the sample rate in bits per second of the speech file. The valid sample rates are as follows:

```
const PKX_AUDIO_SAMPLING_RATE_4KHZ            =0x00010000;
const PKX_AUDIO_SAMPLING_RATE_6KHZ            =0x00020000;
const PKX_AUDIO_SAMPLING_RATE_8KHZ            =0x00040000;
const PKX_AUDIO_SAMPLING_RATE_11KHZ           =0x00080000;
```

Note that under normal circumstances the Telecom Engine will block the calling task until the recording is terminated by a terminating event of some kind. This may be the presence of a DTMF digit in the DTMF digit buffer for the channel, or the maximum number of seconds has been reached or the maximum duration of silence has been detected etc

The reason for the function terminating is returned as the return value of the function and may be one of the following values:

```
const PK_TERM_ERROR       -1;
const PK_TERM_TONE        =1;
const PK_TERM_MAXDTMF      =2;
const PK_TERM_TIMEOUT      =3;
const PK_TERM_SILENCE      =5;
const PK_TERM_ABORT       =6;
const PK_TERM_EODATA       =7;
const PK_TERM_INITSILENCE  =11;
const PK_TERM_NONSIL       =12;
```

(See Terminating events)

Also whenever a jump to the *onsignal* function occurs (for example caused by a hangup signal after a call to PKCallUseSignal()) and if the function is playing in blocking mode (see Blocking and non-blocking mode) then the PKChanRecord() will automatically be aborted (i.e the recording to file will be stopped). If playing in non-blocking mode then it is up to the application to abort the recording and/or wait for it to complete for some other termination reason.

Returns: Returns either an error code (E.g. if the file could not be opened) or the reason for the function termination.

-o-

PKChanRecordh

Synopsis:

term_code=PKChanRecordh(group,chan,file_handle,max_time_ms,max_silence_ms,[encoding[,sample_rate[,beep]]])

Arguments:

group – The logical group number.
channel – The channel numbernel
filename - The filename to record to
[max_time_ms] – Number of millisecond seconds to record
[silence_ms] – Number of milliseconds of silence to end recording
[encoding] – The audio encoding type of voice prompt file to play
[sample_rate] - Optional sample rate

Description: This function is similar to the PKChanRecord() function except that it takes a file handle to a file that has previously been opened by a call to *sys_fhopen()* function in the Telecom Engine standard system library (CXSYS.DLL). It is up to the application to open the file first and to ensure that the file handle is released after use.

The number of millisecods to record is specified by the *max_time_ms* argument. The recording will be terminated if the number of millisecods of silence is detected as specified by *silence_ms*.

If the *encoding* and *sample_rate* are not specified then the last encoding and sample_rate used on the cahnnel are assumed. At start-up the default encoding and sample_rate is set to and the sample rate is 6000. See PKChanRecord() for valid *encoding* and *sample_rate* values.

Note that under normal circumstances the Telecom Engine will block the calling task until the recording is terminated by a terminating event of some kind. This may be the presence of a DTMF digit in the DTMF digit buffer for the channel, or the maximum number of seconds has been reached or the maximum duration of silence has been detected etc

The reason for the function terminating is returned as the return value of the function and may be one of the following values:

```
const PK_TERM_ERROR       -1;
const PK_TERM_TONE        =1;
const PK_TERM_MAXDTMF      =2;
const PK_TERM_TIMEOUT      =3;
const PK_TERM_SILENCE      =5;
const PK_TERM_ABORT       =6;
const PK_TERM_EODATA       =7;
const PK_TERM_INITSILENCE  =11;
const PK_TERM_NONSIL       =12;
```

(See Terminating events)

Also whenever a jump to the *onsignal* function occurs (for example caused by a hangup signal after a call to PKCallUseSignal()) and if the function is playing in blocking mode (see Blocking and non-blocking mode) then the PKChanRecord() will automatically be aborted (i.e the recording to file will be stopped). If playing in non-blocking mode then it is up to the application to abort the recording and/or wait for it to complete for some other termination reason.

Returns: Returns either an error code (E.g. if the file could not be opened) or the reason for the function termination.

-o-

PKChanTermDTMF

Synopsis:

PKChanTermDTMF(group, channel,max_digits[,digit_mask])

Arguments:

group – The logical group number.
channel – The channel number.
max_digits – The number of DTMF digits that will cause a termination event
[digit_mask]- Optional string of terminating DTMF digits.

Description: This function allows the channel specific DTMF termination conditions for the specified *group* and *channel* to be set. The *max_digits* argument specifies the maximum number of DTMF digits that need to be received before an asynchronous media function will be terminated by a PKX_EVENT_CHANNEL_TERM_MAX_DIGITS event. At startup this is set to 1 so that a single digit will terminate all asynchronous media processing functions.

The *digit_mask* argument is a string specifying the set of DTMF digits that will cause an asynchronous media processing function to terminate with a PKX_EVENT_CHANNEL_TERM_DIGIT_MASK event. The set of digits that can be specified in the digit_mask can be one or more of the following: "1234567890*#ABCD"

Below are some examples:

The following example stops DTMF digits from interrupting a PKChanPlay():

```
// Prevent DTMF from interrupting the voice file playback
PKChanTermDTMF(group,channel,0);
x=PKChanPlay(group,channel,"MUSIC.VOX");
// Turn DTMF interruption back on... by setting max_digits to 1
PKChanTermDTMF(group,channel,1);
```

The following example allows the PKChanRecord() function to be terminated only by '*' or '#' digit.

```
// Set the digit mask so only '*' or '#' will interrupt the recording
PKChanTermDTMF(group,channel,0,"*#');
x=PKChanRecord(group,channel,"RECORDING.VOX");
// Turn DTMF interruption back on... by setting max_digits to 1 and the digit_mask to "" (empty string)
PKChanTermDTMF(group,channel,1,"");
```

Note that even when asynchronous function termination by DTMF is disabled using this function the DTMF digits are still detected and placed in the internal PIKA buffer and can be later retrieved by the PKChanGetDTMF() function. to stop DTMF digits from being detected at all use the PKChanToneCtl() function.

Note: The PKChanWaitDTMF() function overrides both of these global channel settings with the *max_digits* and *digit_mask* arguments passed directly into the function.

Returns: 0 upon success or -1 if a bad group or channel was provided.

-o-

PKChanTermTimeout

Synopsis:

PKChanTermTimeout(group, channel,timeout_ms)

Arguments:

group – The logical group number.
channel – The channel number.
timeout_ms – Timeout in millisecond before terminating asynchronous function.

Description: This function allows the channel specific Timeout termination condition for the specified *group* and *channel* to be set. If this function is called with a timeout_ms value greater than 0, then all asynchronous media processing functions will be terminated with a PKX_EVENT_CHANNEL_TERM_TIMEOUT terminating event after the specified time has been reached.

Note: For the PKChanRecord(), PKChanRecordh() functions this global timeout value is overriden by the *timeout_ms* argument passed directly into the function.

Example: In the following code the PKChanWaitDTMF() function will terminate after 10 seconds even if non of the function specific terminating events (such as the first_delay10ths or interdigit_delay10ths) has occurred:

```
// Set the maximum time for digits to be entered to 10 secondsfile playback
PKChanTermTimeout(group,channel,10000);
x=PKChanWaitDTMF(group,channel,6,40,40);
// Turn global timeout termination off again
PKChanTermTimeout(group,channel,0);
```

Returns: 0 upon success or a negative error code.

-o-

PKChanTermNonSil

Synopsis:

PKChanTermNonSil(group, channel,timeout_ms)

Arguments:

group – The logical group number.

channel – The channel number.

timeout_ms – Timeout in milliseconds of non silence before terminating an asynchronous function.

Description: This function allows the channel specific non-silence timeout termination condition for the specified *group* and *channel* to be set. If this function is called with a *timeout_ms* value greater than 0, then all asynchronous media processing functions will be terminated with a PKX_EVENT_CHANNEL_TERM_NONSILENCE terminating event if non-silence is detected on the channel for this amount of time.

Returns: 0 upon success or a negative error code.

-o-

PKChanToneCtl

Synopsis:

PKChanToneCtl(group, channel,DTMF_Detect, Tone_detect)

Arguments:

group	– The logical group number.
channel	– The channel number.
DTMF_detect	- Turns DTMF detection on (1) or off (0)
Tone_detect	- Turns Tone detection on (1) or off (0)

Description: This function turns on or off DTMF and/or Tone detection on the specified *group* and *channel*. Upon startup both DTMF and Tone detection os switched on, but either one of these can be disabled or enabled with this call.

This function maps to the following PIKA function:

```
PK_STATUS PK_API PKX_CHANNEL_SetConfig(TPikaHandle channelHandle,PKX_TChannelSettings *
channelSettings);
```

Returns: 0 upon success or a negative error code.

-o-

PKChanWaitDTMF

Synopsis:

term_code=PKChanWaitDTMF(group,channel,max_tones,first_delay10ths,inter_delay10ths[,term_digits])

Arguments:

group – The logical group number.
channel – The channel number.
max_tones – The maximum number of tones to receive
first_delay10ths – The time to wait for the first digit to be entered (in 10ths of a second)
inter_delay10ths – The maximum time between digits (in 10ths of a second)
[term_digits] – Optional argument specifying a string of DTMF digits that would terminate the input

Description: This function allows the application to block waiting for DTMF input to match the specified terminating conditions defined by the *max_tones*, *first_delay10ths*, *inter_delay10ths* and *term_digits* arguments.

The *max_tones* argument specifies the maximum number of DTMF tones to receive before terminating the PKChanWaitDTMF() function with a PK_TERM_MAXDTMF return value. Note that if the internal PIKA DTMF buffer already holds the number of tones specified by *max_tones* then the function will terminate immediately and return with PK_TERM_MAXDTMF.

The first_delay10ths specifies the maximum time (in 1/10ths second) that the function will wait for the first input tone to be received. If this timeout is exceeded then the function will terminate with a PK_TERM_INITSILENCE return code. IF there are already one or more digits in the internal PIKA buffer then the *first_delay10ths* expires immediately and the *inter_delay10ths* timer is started. If *first_delay10ths* is set to 0 value then the function will terminate immediately with PK_TERM_INITSILENCE if there was not already a digit in the internal PIKA buffer.

The *inter_delay10ths* timer is started after the first digit has been received and specifies the maximum time allowed between all successive received digits. If the *inter_delay10ths* timer is exceeded then the function will be terminated with a PK_TERM_INTERDELAY return value.

The optional *term_digits* argument allows the input to be terminated upon receipt of one of a set of DTMF digits specified as a string of digits. For example if the input is to be terminated by either a '*' or '#' digit then the *term_digits* string should be set to "*#". As soon as either of these digits is received then the function terminates with a PK_TERM_TONE event. The full set of digits that can be specifed by the term_digits argument is: "1234567890*#ABCD".

Note that this function simply waits for the terminating condition specified by the arguments to be met. All received digits are held in the PIKA internal buffers until retreived by the PKChanGetDTMF() function call.

Therefore the PKChanWaitDTMF() and PKChanGetDTMF() functions work in conjunction with each other. The PKChanWaitDTMF() function sets conditions for which tones to wait for and the conditions which will cause the PKChanWaitDTMF() to terminate. A call to PKChanGetDTMF() will copy any DTMF digits it received into the channel specific digit buffer and then return them to the calling task.

Examples:

```
// This will wait for upto 4 digits to be received with the first and inter digit delay set to 4 seconds each
x=PKChanWaitDTMF(group,chan,4,40,40);
//  Get the digits received from the foreground buffer..
tones=PKChanGetDTMF(group,chan);


// This will wait for upto 4 digits to be received unless a * or # is received, with the first and inter digit delay
set to 4 seconds each
x=PKChanWaitDTMF(group,chan,4,40,40,"*#");
//  Get the digits received from the foreground buffer..
tones=PKChanGetDTMF(group,chan);


// This will return immediately with PK_TERM_INITSILENCE unless there is already a DTMF digit already in
the background buffer (first_delay10th set to 0)
x=PKChanWaitDTMF(group,chan,1,0,0);
//  Get the digits received from the foreground buffer..
tones=PKChanGetDTMF(group,chan);
```

Returns: Returns the terminating event or a negative error code.

-o-

PKChanClearDTMF

Synopsis:

PKChanClearDTMF(group, channel)

Arguments:

group – The logical group number.
channel – The channel number.

Description: This function turns removes all DTMF digits from the channel specified by *group* and *channel*. It will remove the DTMF digits from both the PIKA internal buffer as well as the channel specific buffer maintained by the CXPIKA.DLL library.

This function maps to the following PIKA function:

```
PK_STATUS PK_API PKX_CHANNEL_ClearDigits(TPikaHandle channelHandle);
```

Returns: 0 upon success or a negative error code.

-o-

PKChanGetDTMF

Synopsis:

digits=PKChanGetDTMF(group, channel[,max_digits)

Arguments:

group	– The logical group number.
channel	– The channel number.
[max_digits]	- Optional argument to specify the maximum number of digits to retrieve.

Description: This function copies all of the DTMF digits from the PIKA internal DTMF buffer to the channel specific buffer maintained by the CXPIKA.DLL library for the specified *group* and *channel*. If the option *max_digits* argument is given then this number of DTMF digits from the channel specific buffer will be returned to the calling task (or less than *max_digits* if not that many are in the buffer). If max_digits is not specified then all of the DTMF digits that are in the channel specific buffer will be returned to the calling task.

This function maps to the following PIKA function:

```
PK_STATUS PK_API PKX_CHANNEL_GetDigits(TPikaHandle channelHandle,PK_CHAR *
buffer, PK_INT num_digits);
```

Returns: Upon success this function returns the set of DTMF digits from the channel specific DTMF digit buffer, otherwise it will return a negative error code.

-o-

PKChanAbort

Synopsis:

PKChanAbort(group, channel)

Arguments:

group – The logical group number.
channel – The channel number.

Description: This function will abort any asyncronous media processing function that is currently executing on the specified *group* and *channel.* The relevant asyncronous function will then terminate immediately and return a value of PK_TERM_ABORT.

This function maps to the following PIKA function:

```
PK_STATUS PK_API PKX_CHANNEL_Stop(TPikaHandle channelHandle);
```

Returns: 0 upon success or a negative error code.

-o-

PKChanBusyState

Synopsis:

functionID=PKChanBusyState(group, channel)

Arguments:

group – The logical group number.
channel – The channel number.

Description: This function will return a value indicating which *asyncronous media function* is currently running on the specified group and channel, otherwise it will return 0 if there is no *asyncronous media function* currently active.

The set of values that can be returned from this function are defined in **pika.inc** as follows:

```
const PK_MTF_NONE          =0;
const PK_MTF_PLAY          =1;
const PK_MTF_RECORD        =2;
const PK_MTF_WAITTONE      =3;
const PK_MTF_PLAYTONE      =4;
const PK_MTF_PLAYDIGITS    =5;
const PK_MTF_PLAYCPTONE    =6;
```

Returns: The ID of the asyncronous function currently executing on the channel, or 0 if the channel is idle or a negative error code.

-o-

PKChanBlockMode

Synopsis:

PKChanBlockMode(group, channel,block_mode)

Arguments:

group	– The logical group number.
channel	– The channel number.
block_mode	- 0=Blocking, 1=Single non-blocking call, 2=Indefinite Non-blocking

Description: This function allows for asynchronous media functions (such as PKChanPlay(), PKChanRecord() etc) to be carried out in either blocking or non-blocking mode. All of the media functions that operate in this way are known as *asyncronous media functions*.

In blocking mode the calling Telecom Engine task will *block* until the *asyncronous media functions* has completed after which the function will return with the terminating event that caused the *asyncronous media functions* to complete.

The set of *asyncronous media functions* for which this function allies to is as follows:

term_code=PKChanPlay(group,channel,filename[,encoding,sample_rate])
term_code=PKChanPlayh(group,chan,filehandle[bytes,encoding,sample_rate])
term_code=PKChanRecord
(group,chan,filename,max_time_ms,max_silence_ms,[encoding[,sample_rate[,beep]]])
term_code=PKChanRecordh
(group,chan,file_handle,max_time_ms,max_silence_ms,[encoding[,sample_rate[,beep]]])
term_code=PKChanWaitDTMF
(group,channel,max_tones,first_delay10ths,inter_delay10ths[,term_digits])

In non-blocking mode the function will return immediately and the media function will continue playing in the background while the Telecom Engine task continues processing. In this case it is up to the program to wait for the operation to complete (using PKChanBusyState()) or to specifically abort the speech operation using PKChanAbort().

See Blocking and Non-blocking Mode for more information**.**

If the *block_mode* argument is set to 0 then this causes all subsequent *asyncronous media functions* to block the calling Telecom Engine task (this is the default behaviour).

If the *block_mode* flag is set to 1 then the next *asyncronous media functions* will operate in non-blocking mode. After this media function has completed then the channel mode reverts back to *blocking,* so that further calls to *asyncronous media functions* will block the calling task as normal.

If the *block_mode* flag is set to 2 then the channel will be placed into non-blocking mode indefinitely so that all subsequent *blocking speech functions* will operate in non-blocking mode.
A call to PKChanBLockMode() with the *block_mode* flag set to 0 will be required to set the channel back to blocking mode again.

Note that when in *blocking* mode a jump to onsignal will cause the *asyncronous media functions* to

be immediately aborted with a call to PKX_CHANNEL_Stop(). However in *non-blocking* mode a jump to onsignal will not automatically abort the *asyncronous media function* and it is thus up to the program to call PKChanAbort() to abort the function (or else to use PKChanBusyState() to wait for the function to finish).

Returns: 0 upon success or a negative error code.

-o-

Index

- I -

- A -

- T -

- B -

- P -

- C -

- P -

- M -

- E -

- P -