TB640 User's guide

Document number
9000-00002-2H

January 2008

This page in intentionally left blank

Table of contents

Figures

Tables

INTRODUCTION

This User’s Guide is intended for the users of the TelcoBridges adapters. It will explain the concepts behind the API and how it can be used to build complex applications. Some telecommunications knowledge is expected from the reader as well as a basic understanding of the “C” language.

This manual is divided into sections that are constructed to be read in a linear fashion. It is recommended to read the User’s Guide entirely before developing an application. This manual can be seen as the preface to the TBX API Reference Guide and the TB640 Msg API.

1 OVERVIEW

The TelcoBridges API family is intended to give full flexibility to the user while providing accessibility and control on all the adapter functions. The API calls allow you to configure the TelcoBridges adapters, allocate and define resources, connect them together, initiate and accept calls and manage the adapter. It is based around a message passing mechanism that is asynchronous by nature. Three types of messages are available to the application designer: requests, responses and events. Requests are used to send commands and they can be sent by the user application or by the adapter. A request is always associated with a response which will carry the result of the execution of the request. The third type of message is used for asynchronous notification of events.

The transport mechanism for the messages that are exchanged between the application and the adapters is transparent as seen by the application. The discovery of which adapter is available to the application is automatically done by the library and information can be retrieved by a simple API call. The application can then use the list of adapters returned to decide which one to attach to. An adapter that has its two ethernet ports available can be attached to using network redundancy mode. In network redundancy mode, the application will continue to communicate seamlessly with the adapter even if one of the networks fails.

Two types of resources are offered by the TelcoBridges adapters: channels resources and voice processing resources. Channels resources are physical entry/exit points that can connect to the exterior world (trunk, H.110, etc.). They must be allocated by the application before being used specifying the exact location (trunk, stream, timeslot, channel…) of the resource to be allocated. Voice processing resources are used to modify the contents of the voice channels handled by the adapter (tone detectors, conferencing, play, record, etc.).

Interaction between the signaling entities and the user application are done using the same messaging interface. Requests for call setup and teardowns are originated by the application while request for incoming call setup comes from the adapter. The signaling protocol is configurable on a trunk per trunk basis and multiple different signaling protocols can be used simultaneously on different trunks. Moreover, this can be done live without restarting or rebooting the TelcoBridges adapter.

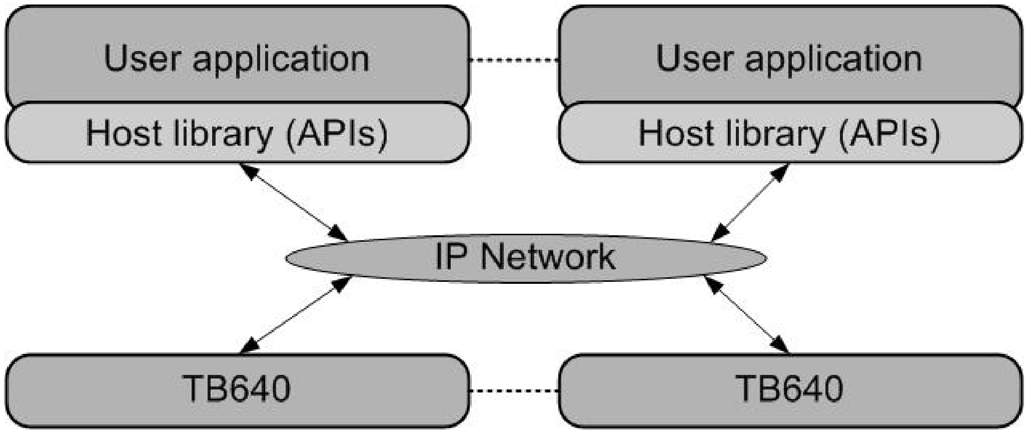


Figure 1: High-level Software view

2 MESSAGES

All the communication between the user application and the adapter is done via the use of messages. Three types of messages are available to the user application: requests, responses and events. All three types of messages can be sent and received by the application. Requests are always followed by an associated response. Response is sent back to the application which sent the request, not any other application. Events are sent by the TB640 to all applications attached to the TB640. Messages sent using the TelcoBridges API are inherently of an asynchronous nature. Facilities are provided to operate in a synchronous manner. Filters are used by the application to select which messages should be processed.

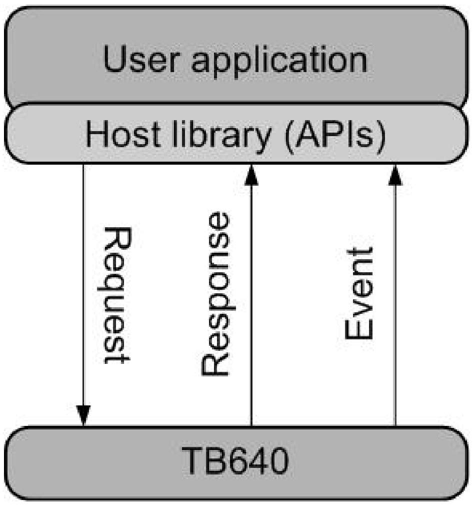


Figure 2: Message types

2.1 Asynchronous messaging

This API is designed to allow applications to get full advantage of the parallel processing capability between the host and the adapters. It is possible, for example, to send multiple requests in batch and wait for responses in a separate thread or only when the application is ready to do so. Note that responses are not sent back to the application using callback functions. The application will never get interrupted by the API. The application decides when it is time to wait for the responses or events. The application should use multi-threading to do operations in parallel as much as possible and to obtain best performance. Responses can be matched with requests using implicit filtering services, however explicit filtering is more appropriate when performing asynchronous message exchange. The implicit filtering should be seen as a service to implement synchronous kind of call and explicit filtering service to implement asynchronous call. The application can use the **TBX_MSG_USER_CONTEXTx_SET** and **TBX_MSG_USER_CONTEXTx_GET** macros to write / read user context fields to be able to match response and request messages.

Note: for easy usage, a macro named **TBX_FORMAT_MSG_HEADER** is used to set the appropriate fields of a message before sending it to the board.

2.1.1 Why asynchronous messaging?

The TB640 adapters handle a high number of resources. Each resource is independent from the others. Because of the very high number of resources, it is not feasible for a host application to dedicate one thread per resource. Instead, we recommend using few threads (typically one thread per host processor) to handle state changes of big number of resources.

Of course, when one thread is responsible for multiple resources, everything must be handled asynchronously. For example, after sending a "allocate resource" request to an adapter, the thread must be immediately available to send more requests or process incoming events/responses from any resource. Eventually, the response from the "allocate resource" request will be received and the state machine for the corresponding resource can go on.
Using this coding scheme will allow taking advantage of 100% of the CPU power available on the adapters or host CPUs.
It is very important to understand that it's illegal to perform any synchronous operation within such an asynchronous state machine. In fact, the host thread must always be available to process incoming responses and events. Using purely asynchronous code allows the thread to handle any task while a particular resource is waiting for a response. Also, if a resource dies, it will not affect other resource's state machines.

2.1.2 Zero loss of responses and events

To simplify coding of asynchronous applications, the TB640 API has been developed to provide zero loss of responses and events.
Every request will have one, and exactly one, response returned. The response may indicate success or failure. Thus a state machine does not need to use any timers or complex management[1]. Reception of the response is assured unless the host application receives the event TBX_MSG_ID_API_NOTIF_ADAPTER_REMOVED, which indicates communication between host and adapter has been interrupted.
In summary, after sending a request, the possible situations are:

- Response (success) is received. State machine can continue for this resource.
- Response (failure) is received. State machine should cancel for this resource.
- Adapter is disconnected. State machine should "stall" for every resource until adapter is reconnected. When adapter is reconnected, host application should list resources and connections to resynchronize its state machine with the adapter for all resources.

2.2 Synchronous messaging

This API has not been designed to perform synchronous calls but it is an easy task to do in the application code to perform call that behaves synchronously. The application can use implicit message filtering method to block and retrieve responses using standard **TBXReceiveMsg** call just after request has been sent using the standard **TBXSendMsg** call. Code examples in this document that use the implicit message filtering method are illustrating the synchronous messaging strategy.

2.2.1 Dangers of synchronous messaging

It is highly recommended to develop applications using asynchronous state machines. When using ISDN, CAS or SS7 stacks, not only is it recommended, it is mandatory.
Using an asynchronous state machine allows the application to process changes on a resource while results of operations on other resources are pending. This allows using the full potential of the host and adapters. Applications using this scheme will easily get 10 times the performance of applications that process everything synchronously.
Keep in mind that, within an asynchronous state machine, synchronous calls are illegal.

2.3 Filters

Filters are an important concept in the TelcoBridges API. They are used by the application to indicate to the API which messages it wants to receive. The application can create as many explicit filters as it wishes to. Each filter has an associated queue (FIFO) where the incoming messages that match the filter are queued. The application can then retrieve theses messages at its convenience.

[1] Some signaling stacks (ISDN, CAS, SS7) may require the host to implement some timeouts for specific state machine states. For applications using these stacks, please refer to the appropriate section of this document. For any other state machines, no state machine timeouts are required as long as the adapter remains connected to the host.

In a more detailed manner, when a message is received by the API from the adapter, the API scans the list of filters that were registered by the application to find all the ones that match. It then creates multiple copies of the message and queues all of them for later retrieval by the application. After having put the message on the filter's queue, the API will then check if any thread is blocked waiting for a message on this filter and awaken one of them. Only one thread must be waiting on a particular filter or else the API will choose the one to schedule in a random fashion.

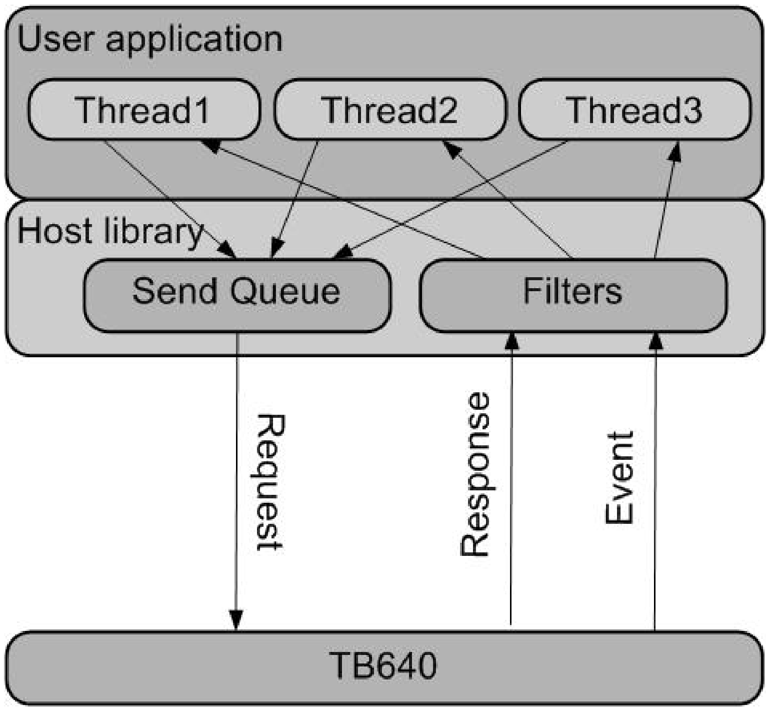
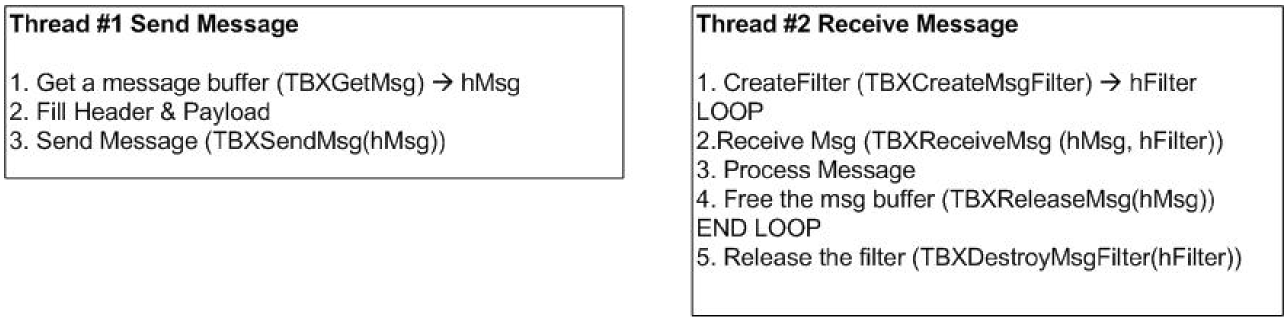


Figure 3: Message type and filters

2.3.1 Explicit filters

When a request is sent to the adapter by the application using the **TBXSendMsg** call, the application can tell the API (using a **NULL** pointer for **out_phFilter** argument) that no implicit filter is required. The application can create its own filters with **TBXCreateMsgFilter** call and use its own strategy to retrieve and match asynchronous responses with previously sent request. This brings out another powerful aspect of the filter mechanisms: the user application can create multiple filters corresponding to certain selected messages. At this point, the library allows the user application to create filters that traps messages from a specific adapter, a certain type (request/response/event), a specific message ID, specific message group (e.g. all trunk related messages), specific user context(s) or any combination of any of those. This mechanism allows a user application to redirect incoming events to a specific flow of processing without having to do a special lookup sequence.

The application has the possibility to set whatever value in the two user context fields in the header of each message sent using the **TBX_MSG_USER_CONTEXT1_SET** or **TBX_MSG_USER_CONTEXT2_SET** macros. Every response header will contain the user context fields that match the request's user context field(s) initialized by the application. If desired, the application can also use one of the context to store a value for the filtering (such as the thread ID that would process the response) and use the other one to store a structure pointer for fast dereferencing.



Thread #1 Send Message

```
1. Get a message buffer (TBXGetMsg) → hMsg
2. Fill Header & Payload
3. Send Message (TBXSendMsg(hMsg))
```

Thread #2 Receive Message

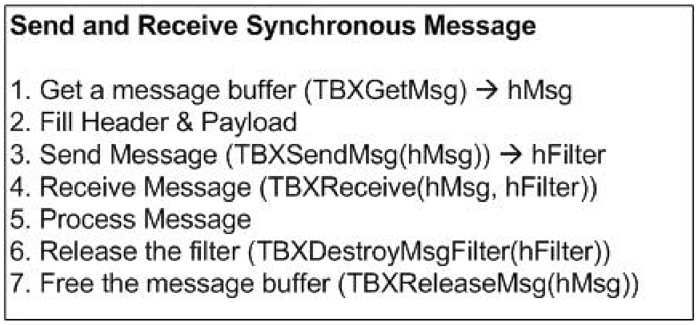
```
1. CreateFilter (TBXCreateMsgFilter) → hFilter
LOOP
2.Receive Msg (TBXReceiveMsg (hMsg, hFilter))
3. Process Message
4. Free the msg buffer (TBXReleaseMsg(hMsg))
END LOOP
5. Release the filter (TBXDestroyMsgFilter(hFilter))
```

Figure 4: Asynchronous Messaging (explicit filtering)

2.3.2 Implicit filters

When a request is sent to the adapter by the application using the **TBXSendMsg** call, the application can ask the API (**non NULL** pointer for **in_pFilterHandle** argument) to create an implicit filter that will match the expected response for the request sent. Note that an arbitrary number of operations can be executed between the call that sends the message and the one that receives the response, including sending other messages to the same adapter. As the internally generated messages are inherently asynchronous, the application cannot take for granted that multiple requests will be retired in order. Although, the application can use the implicit filter to tell implicitly to the API which response the application wants to receive exclusively in a synchronous kind of behavior. If a response finds a matching implicit filter, then only this implicit filter will receive the message. Thus, the response will not be received by any other explicit filter that would have otherwise received this message.

IMPORTANT Note #1: The application is responsible to destroy the implicit filter when it is not used anymore. Failure to do so will probably end-up in the application leaking buffers and requiring enormous amount of memory out of the system.
IMPORTANT note #2: The response from a request will always be received (unless the **TBX_MSG_ID_API_NOTIF_ADAPTER_REMOVED** event is received). A timeout in a **TBXReceiveMsg** using an implicit filter does not mean the response is lost, only that it has not been received in the timeout period specified in **TBXReceiveMsg**. Thus we recommend using an infinite duration in the **in_unTimeoutMSec** parameter (-1) or a do loop until this message is received.



Send and Receive Synchronous Message

```
1. Get a message buffer (TBXGetMsg) → hMsg
2. Fill Header & Payload
3. Send Message (TBXSendMsg(hMsg)) → hFilter
4. Receive Message (TBXReceive(hMsg, hFilter))
5. Process Message
6. Release the filter (TBXDestroyMsgFilter(hFilter))
7. Free the message buffer (TBXReleaseMsg(hMsg))
```

Figure 5: Synchronous Messaging (implicit filtering)

2.4 Code example #1: Create an explicit message filter to retrieve event messages from all accessible adapters

Here is a code example showing how to create an explicit message filter to retrieve event messages from all accessible adapters:

…

```
TBX_RESULT_API          APIResult;
TBX_FILTER_PARAMS       aFilterParams [1];
TBX_FILTER_HANDLE       hFilter;
```

…

```
/* Create an explicit message filter to retrieve all event messages from all
accessible adapters */
aFilterParams[0].un32StructVersion = 1;
aFilterParams[0].FilterMask = TBX_FILTER_MSG_TYPE;
aFilterParams[0].MsgTypeMask = TBX_MSG_TYPE_EVENT;
APIResult = TBXCreateMsgFilter (
  in_hLib,        /* Library handle returned by TBXOpenLib call */
  (sizeof(aFilterParams) / sizeof(TBX_FILTER_PARAMS)),
  aFilterParams,
  &hFilter );

/* At this point in the code, FilterHandle could be used as argument by
TBXReceiveMsg, TBXWaitMsg and TBXFlushMsgs calls to retrieve event messages
from all accessible adapters */
```

…

```
/* Destroy the explicit message filter */
TBXDestroyMsgFilter ( in_hLib, hFilter );

/* At this point, hFilter should not be used anymore */
```

…

2.5 Code example #2: Create an explicit message filter to retrieve response messages from a specific adapter

Here is a code example showing how to create an explicit message filter to retrieve response messages from a specific adapter:

…

```
TBX_RESULT_API          APIResult;
TBX_FILTER_PARAMS       aFilterParams [1];
TBX_FILTER_HANDLE       hFilter;
```

…

```
/* in_hLib is returned by TBXOpenLib call */
/* in_hAdapter is returned by TBXGetAdaptersList call */

/* Create an explicit message filter to retrieve all event messages from all
accessible adapters */
aFilterParams[0].un32StructVersion = 1;
```

```
aFilterParams[0].FilterMask =
  ( TBX_FILTER_ADAPTER_HANDLE | TBX_FILTER_MSG_TYPE );
aFilterParams[0].hAdapter = in_hAdapter;
aFilterParams[0].MsgTypeMask = TBX_MSG_TYPE_RESPONSE;
APIResult = TBXCreateMsgFilter (
  in_hLib,
  (sizeof(aFilterParams) / sizeof(TBX_FILTER_PARAMS)),
  &hFilter );

/* At this point, hFilter could be used as argument by TBXReceiveMsg,
TBXWaitMsg and TBXFlushMsgs calls to retrieve response messages from the
specified adapter */
…
/* Destroy the explicit message filter */
TBXDestroyMsgFilter ( in_hLib, hFilter );

/* At this point, hFilter must not be used anymore */
…
```

2.6 Code example #3: Send request message to attach specific adapter (explicit filtering method)

Here is a code example showing how to send request message to attach specific adapter that do not use the automatic implicit filtering creation (explicit filtering method):

```
…
TBX_RESULT_API                APIResult;
TBX_MSG_HANDLE                hMsg;
TBX_REQ_ADAPTER_OP_ATTACH *   pRequestAttach;

/* in_hLib is returned by TBXOpenLib call */
/* in_hAdapter is returned by TBXGetAdaptersList call */
/* in_pUserContext is a user defined field that can be used to match response
and request messages when using explicit message filtering */

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Get message buffer */
      APIResult = TBXGetMsg (
        in_hLib,
        sizeof ( TB640_MSG_ADAPTER_OP_ATTACH ),
        &hMsg );

}

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Initialize message header */
      TBX_FORMAT_MSG_HEADER (
        hMsg,
        TB640_MSG_ID_ADAPTER_OP_ATTACH,
        TBX_MSG_TYPE_REQUEST,
        Sizeof (TB640_MSG_ADAPTER_OP_ATTACH)
```

```
          in_hAdapter,
          in_pUserContext,
          0);

        /* Set the message payload */
        pRequestAttach =
          (PTB640_REQ_CONN_OP_ATTACH)TBX_MSG_PAYLOAD_POINTER( hMsg );
        pRequestAttach->un32MsgVersion = 1;
        pRequestAttach->un32KeepAliveTimeoutSec = 1;
        pRequestAttach->fEnableAutoReattach = TBX_TRUE;

        /* Send the attach request to specific adapter. Note that the last
        argument is NULL. This call will NOT return handle on an implicit filter.
        An explicit filter created with TBXCreateMsgFilter call must be used to
        retrieve the response. */
        Result = TBXSendMsg (
          in_hLib,
          &hMsg,
          NULL );

        /* At this point, the application can use TBXReceiveMsg, TBXWaitMsg and
        TBXFlushMsg calls to handle the response message. An explicit filter must
        have been created before sending the request. */
}
```

2.7 *Code example #3: Send request message to attach specific adapter (implicit filtering method)*

Here is a code example showing how to send request message to attach specific adapter that use the automatic implicit filtering creation (implicit filtering method):

```
…
TBX_RESULT_API                 APIResult;
TBX_MSG_HANDLE                 hMsg;
TB640_REQ_ADAPTER_OP_ATTACH *  pRequestAttach;
TBX_FILTER_HANDLE              hFilter;

/* Initialize local variables */
hFilter = 0;

/* in_hLib is returned by TBXOpenLib call */
/* in_hAdapter is returned by TB640GetAdaptersList call */

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
        /* Get message buffer */
        APIResult = TBXGetMsg (
          in_hLib,
          sizeof( TB640_MSG_ADAPTER_OP_ATTACH ),
          &hMsg );
}

if ( TBX_RESULT_SUCCESS( APIResult ) )
```

```
{
      /* Initialize message header */
      TBX_FORMAT_MSG_HEADER (
        hMsg,
        TB640_MSG_ID_ADAPTER_OP_ATTACH,
        TBX_MSG_TYPE_REQUEST,
        Sizeof (TB640_MSG_ADAPTER_OP_ATTACH)
        in_hAdapter,
        0,
        0);

      /* Set the message payload */
      pRequestAttach =
        (PTB640_REQ_CONN_OP_ATTACH)TBX_MSG_PAYLOAD_POINTER( hMsg );
      pRequestAttach->un32MsgVersion = 1;
      pRequestAttach->un32KeepAliveTimeoutSec = 1;
      pRequestAttach->fEnableAutoReattach = TBX_TRUE;

      /* Send the attach request to specific adapter. Note that the last
      argument is non NULL. This call will return handle on an implicit filter.
      This filter can be used to retrieve the response to this request. */
      APIResult = TBXSendMsg (
        hLib,
        &hMsg,
        &hFilter );

      /* At this point, the application can use TBXReceiveMsg, TBXWaitMsg and
      TBXFlushMsg calls with the implicit hFilter filter to handle the response
      message. */
}

…

if (hFilter != 0)
{
      /* Destroy the implicit message filter */
      TBXDestroyMsgFilter ( in_hLib, hFilter );
}

…
```

2.8 Code example #4: Receive response message

Here is a code example showing how to use implicit or explicit filter to retrieve response message:
…

```
TBX_RESULT_API                  APIResult;
TB640_RESULT                    Result;
TBX_MSG_HANDLE                  MsgHandle;
TBX_MSG_ADAPTER_OP_ATTACH *     pMsgAttach;
TBX_RSP_ADAPTER_OP_ATTACH *     pResponseAttach;

/* in_hLib is returned by TBXOpenLib call */
/* in_hFilter is returned by TBXCreateMsgFilter call */
/* Use implicit filter returned by TBXSendMsg call or explicit filter created
with TBXCreateMsgFilter call to retrieve response message. Call blocks for
maximum 5 second and will return as soon as a response message match the
characteristics of the filter. Consider that the explicit filter has been
created to trig on response type of messages */
APIResult = TBXReceiveMsg (
  in_hLib,
  in_hFilter,
  5000,
  &hMsg );

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Log message identification information */
      printf("Adapter handle = %X\n", TBX_MSG_ADAPTER_HANDLE_GET( hMsg ));
      printf("Message type = %d\n", TBX_MSG_TYPE_GET( hMsg ));
      printf("Message ID = %d\n", TBX_MSG_ID_GET( hMsg ));
      printf("Serial number = %X\n", TBX_MSG_SERIAL_NUMBER_GET( hMsg ));
      printf("User context1 = %X\n", TBX_MSG_USER_CONTEXT1_GET( hMsg ));
      printf("User context2 = %X\n", TBX_MSG_USER_CONTEXT2_GET( hMsg ));
      printf("Payload length = %d\n", TBX_MSG_PAYLOAD_LENGTH_GET( hMsg ));
      printf("Payload maximum length = %d\n",
        TBX_MSG_PAYLOAD_LENGTH_GET( hMsg ));
      printf("Payload pointer = %X\n", TBX_MSG_PAYLOAD_POINTER( hMsg ));

      switch ( TBX_MSG_ID_GET( hMsg ) )
      {
            case TB640_MSG_ID_ADAPTER_OP_ATTACH:
            {
                  /* Retrieve the message payload */
                  pResponseAttach = (PTB640_RSP_ADAPTER_OP_ATTACH)
                    TB640_MSG_PAYLOAD_POINTER( hMsg );
                  Result = pResponseAttach->Result;

                  if ( TBX_RESULT_SUCCESS( Result ) )
                  {
                        printf( "SUCCESS: Attach adapter" );
                  }
                  else
                  {
                        printf( "FAILURE: Attach adapter" );
                  }
            }
            break;

            default:
            {
                  printf ("Message with identifier = %X is not handle by this
                  program.\n", TBX_MSG_ID_GET ( hMsg ));
            }
      }

      /* Release message buffer */
      APIResult = TBXReleaseMsg(
        in_hLib,
        hMsg );

      /* At this point, the hMsg handle must not be used anymore. */
}
```

…

3 HOST LIBRARIES

3.1 Linking with the HostLibs

Depending on what host you're running, and what type of libraries you're using, you will have to link with one of the following libraries.

Host	TBX Host Library	Stream Library	StreamServer Library
Win32 Multi Threaded	tbx.lib	streamlib.lib	tbstreamserver_lib.lib
Win32 Multi Threaded DLL	tbx_md.lib	streamlib_md.lib	tbstreamserver_lib_md.lib
Solaris 8/9	libtbxhost.a	libtbxstream.a	libtbstreamserver.a

3.2 TBX HostLib Debug API

To enable debug functionality, define TBX_API_DEBUG in your makefile or include "tbx_api_debug.h" instead of "tbx_api.h"

The API library in debug mode checks validity of every handle given as argument to API calls. In the event an error is detected, the library prints an error message telling what file and what line number the error was originated.

4 INITIALIZATION

First of all, the application can, at anytime, call **TBXGetLibInfo** to get information about the library. It could be interesting for the application to get information on the library before starting the initialization sequence. Therefore **TBXGetLibInfo** is typically the first call to be done by the application to the API library. Although this call provides valuable information, it isn't required to perform other API calls and could be done later or even not at all. The **TBXGetLibInfo** call returns the version of the library, the build number, the available transport layer types to exchange messages with adapters and other information.

Secondly, the application must call **TBXOpenLib** to initialize the library and get access to other available API calls. At completion of this call the library starts to monitor available adapters and dispatch messages coming from or sent to the different adapters. Therefore the application could have messages sent to it. For example, event messages could be sent to indicate the arrival or departure of adapters. At this point in the initialization sequence, the application can create message filters with **TBXCreateMsgFilter** call and start to receive messages with **TBXReceiveMsg** call. Note that adapter handles are unknown at this point. Therefore, the message filter cannot specify a particular adapter. Important things that a filter should detect at this point are **TBX_MSG_ID_API_NOTIF_ADAPTER_ADDED** and **REMOVED** events. A thread can be started to receive those events with **TBXReceiveMsg**, using the filter created.

Thirdly, the application can call **TBXGetAdaptersList** to get the list of available adapters. This function returns you with a handle for each adapter that are alive and that can be attached to the library. Each of these adapter handles is unique throughout a system (even with other applications). It is possible to get more information (adapter name, serial number, slot and shelf id…) on every available adapter by using **TBXGetAdapterInfo**. At this point in the initialization sequence, the application can start to receive messages from specific adapters with **TBXReceiveMsg** call and send requests to specific adapter with **TBXSendMsg** call that do not require attachment privilege.

Finally, the application can send a **TB640_MSG_ADAPTER_OP_ATTACH** message to get full control on specific adapter. Once attached to the adapter, the application will start receive events sent by the adapter (assuming the application has allocated filters that match the sent events). When not attached to the adapter, the application can still send request to the adapter and receive the responses, but will not receive any events sent by the adapter. When attaching to an adapter, it is the responsibility of the application to properly synchronize with adapter's resources and states to recover from dropped events while the application was not attached to the adapter.

By default, the host library will connect to the adapter using network redundancy mode (assuming both Ethernet interfaces of the adapter are connected). To modify the network redundancy parameters (including disable network redundancy), the application can call TBXConfigureNetworkRedundancy. The host library will send events type TBX_MSG_ID_API_NOTIF_ADAPTER_ETH_[UP/DOWN] to keep the application informed of Ethernet ports availability. At any time, the application can call the function TBXGetNetworkRedundancyState to query the network redundancy state.

4.1 Code example #5: Initialize the library and attach specific adapter using the implicit filter

Here is a code example showing how to initialize the library and attach the first adapter of the adapters list using the implicit filter to retrieve response message in synchronous like behavior:

```
#define ADAPTER_MAX_COUNT 16

…

TBX_RESULT_API                 APIResult;
TB640_RESULT                   AttachResult;
```

```
TBX_LIB_INFO                    LibInfo;
TBX_LIB_PARAMS                  LibParams;
TBX_LIB_HANDLE                  hLib;
TBX_UINT                        unAdapterHandleCount;
TBX_ADAPTER_HANDLE              ahAdapter[ADAPTER_MAX_COUNT];
TBX_ADAPTER_HANDLE              hAdapter;
TBX_FILTER_HANDLE               hFilter;
TBX_MSG_HANDLE                  hMsg;
TBX_REQ_ADAPTER_OP_ATTACH *     pRequestAttach;
TBX_RSP_ADAPTER_OP_ATTACH *     pResponseAttach;
TBX_UINT                        unIndex;

/* Initialize local variables */
hFilter = 0;

…

/* Get library information */
LibInfo.ui32StructVersion = 1;
APIResult = TBXGetLibInfo ( &LibInfo );
if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Print library information */
      printf( “Library Version %d.%d.%d, Build number %d\n”,
        LibInfo.Version.un8Major,
        LibInfo.Version.un8Minor,
        LibInfo.Version.un16PatchLevel,
        LibInfo.Version.un32BuildNumber );
      printf( “Ethernet transport is %s\n”,
        LibInfo.Transport & TBX_TRANSPORT_ETHERNET ?
        “available” : “not available” );

      /* Set library parameters */
      LibParams.un32StructVersion = 1;
      LibParams.un32AutoDetectionDelayMin = 500;
      LibParams.un32WatchdogTimeoutSec = 0;
      LibParams.MsgRxThreadPrio = TBX_MSG_RX_THREAD_PRIO_HIGHER;
      LibParams.un32SendFifoSize = TBX_SEND_FIFO_SIZE_DEFAULT;

      if ( LibInfo.Transport & TBX_TRANSPORT_ETHERNET )
      {
            /* Use Ethernet to transport messages */
            LibParams.Transport = TB640_TRANSPORT_ETHERNET;
      }
      else
      {
            /* No transport is available or unknown transport type */
            APIResult = TBX_RESULT_FAIL;
      }
}

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Open library */
      APIResult = TBXOpenLib( &LibParams, &hLib );
}
```

```
if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* At this point in the initialization sequence, the application can
      create message filters using the TBXCreateMsgFilter call that do not
      specify the adapter and start to receive messages from all adapters with
      TBXReceiveMsg call. */

      /* Get adapters list */
      APIResult = TBXGetAdaptersList (
        hLib,
        ADAPTER_MAX_COUNT,
        &unAdapterHandleCount,
        &ahAdapter[0] );

      if ( unAdapterHandleCount == 0 )
      {
            /* No adapter found */
            APIResult = TBX_RESULT_FAIL;
      }
}

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* At this point in the initialization sequence, the application can
      scan the adapters list and retrieve information for each adapter (name,
      serial number, slot and shelf id…) to select a specific adapter. */
      for (unIndex = 0; unIndex < ADAPTER_MAX_COUNT; unIndex++)
      {
            APIResult = TBXGetAdapterInfo (hLib, ahAdapter [unIndex],
            &AdapterInfo);
            if ( TBX_RESULT_SUCCESS (APIResult) &&
                 (strcmp (AdapterInfo.szAdapterName, “TB00”) == 0))
            {
                  /* The adapter has been found. */
                  hAdapter = ahAdapter [unIndex];
                  break;
            }
      }

      if (unIndex == ADAPTER_MAX_COUNT)
      {
            /* Adapter not found */
            result = TBX_RESULT_API_NOT_FOUND;
      }
}

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Get message buffer */
      APIResult = TBXGetMsg (
        hLib,
        sizeof( TB640_MSG_ADAPTER_OP_ATTACH ),
        &hMsg );
}
```

```
if ( TBX_RESULT_SUCCESS( APIResult ) )
{
	/* At this point in the initialization sequence, the application can
	create message filters with TBXCreateMsgFilter call and start to receive
	messages from specific adapter or all adapters with TBXReceiveMsg call.
	The application can also send requests to specific adapter that do not
	require attachment privilege. */

	/* Get message buffer */
	APIResult = TBXGetMsg (
	  hLib,
	  sizeof( TB640_MSG_ADAPTER_OP_ATTACH ),
	  &hMsg );
}

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
	/* Initialize message header */
	TBX_FORMAT_MSG_HEADER (
	  hMsg,
	  TB640_MSG_ID_ADAPTER_OP_ATTACH,
	  TBX_MSG_TYPE_REQUEST,
	  Sizeof (TB640_MSG_ADAPTER_OP_ATTACH)
	  hAdapter,
	  0,
	  0);

	/* Set the message payload */
	pRequestAttach = (PTB640_REQ_ADAPTER_OP_ATTACH)
	  TBX_MSG_PAYLOAD_POINTER( hMsg );
	pRequestAttach->un32MsgVersion = 1;
	pRequestAttach->un32KeepAliveTimeoutSec = 1;
	pRequestAttach->fEnableAutoReattach = TBX_TRUE;

	/* Send the attach message to first adapter in the list. Note that the
	last argument is non NULL. This call will return handle on an implicit
	filter. This filter can be used to retrieve the response to this request.
	*/
	APIResult = TBXSendMsg (
	  hLib,
	  hAdapter,
	  &hMsg,
	  &hFilter );
}

/* Insert code here for the operations to be done in parallel */
/* Multi-threading is recommended to obtain best performance and take full
advantage of the asynchronous capabilities */

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
	/* Use implicit filter returned by TBXSendMsg call to retrieve response
	message. Call blocks for maximum 5 second. */
	APIResult = TBXReceiveMsg (
	  hLib,
        hFilter,
        5000,
        &hMsg );
}

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Retrieve the message payload */
      pResponseAttach = (PTB640_RSP_ADAPTER_OP_ATTACH)
        TBX_MSG_PAYLOAD_POINTER( hMsg );
      AttachResult = pResponseAttach->Result;

      if ( TBX_RESULT_SUCCESS( AttachResult ) )
      {
            printf( "SUCCESS: Attach adapter\n" );
      }
      else
      {
            printf( "FAILURE: Attach adapter\n" );
      }

      /* Release message buffer */
      APIResult = TBXReleaseMsg(
        hLib,
        hMsg );
}

if (hFilter != 0)
{
      /* Destroy the implicit message filter */
      TBXDestroyMsgFilter ( in_hLib, hFilter );
}

/* At this point in the initialization sequence, the application can create
message filters with TBXCreateMsgFilter call and receive messages from specific
adapter or all adapters with TBXReceiveMsg call. The application can also send
requests to specific adapter without any restriction for the attached adapter.
*/

…
```

4.2 Code example #6: Un-initialize the library and detach specific adapter using the implicit filter

Here is a code example showing how to un-initialize the library and detach a specific adapter using the implicit filter to retrieve response message in synchronous like behavior:

…

```
TBX_RESULT_API                APIResult;
TB640_RESULT                  DetachResult;
TBX_MSG_HANDLE                hMsg;
TBX_FILTER_HANDLE             hFilter;
```

```
TB640_REQ_ADAPTER_OP_DETACH * pRequestDetach;
TB640_RSP_ADAPTER_OP_DETACH * pResponseDetach;

/* Initialize local variables */
hFilter = 0;

…

/* in_hLib is the library handle returned by TBXOpenLib call
in_hAdapter is an adapter handle returned by TBXGetAdaptersList call */

/* Get message buffer */
APIResult = TBXGetMsg(
  in_hLib,
  sizeof( TB640_MSG_ADAPTER_OP_DETACH ),
  &hMsg );

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Initialize message header */
      TBX_FORMAT_MSG_HEADER (
        hMsg,
        TB640_MSG_ID_ADAPTER_OP_DETACH,
        TBX_MSG_TYPE_REQUEST,
        Sizeof (TB640_MSG_ADAPTER_OP_DETACH)
        in_hAdapter,
        0,
        0);

      /* Set the message payload */
      pRequestDetach = (PTB640_REQ_ADAPTER_OP_DETACH)
        TBX_MSG_PAYLOAD_POINTER( hMsg );
      pRequestDetach->un32MsgVersion = 1;

      /* Send the attach message to first adapter in the list. Note that the
      last argument is non NULL. This call will return handle on an implicit
      filter. This filter can be used to retrieve the response to this request.
      */
      APIResult = TBXSendMsg (
        in_hLib,
        &hMsg,
        &hFilter );
}

/* Insert code here for the operations to be done in parallel */
/* Multi-threading is recommended to obtain best performance and take full
advantage of the asynchronous capabilities */

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Use implicit filter returned by TBXSendMsg call to retrieve response
      message. Call blocks for maximum 5 second. */
      APIResult = TBXReceiveMsg (
        in_hLib,
        hFilter,
        5000,
        &hMsg );
}

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Retrieve the message payload */
      pResponseDetach =
        (PTB640_RSP_ADAPTER_OP_DETACH)TBX_MSG_PAYLOAD_POINTER( hMsg);
      DetachResult = pResponseDetach->Result;

      if ( TBX_RESULT_SUCCESS( DetachResult ) )
      {
            printf( "SUCCESS: Detach adapter\n" );
      }
      else
      {
            printf( "FAILURE: Detach adapter\n" );
      }

      /* Release message buffer */
      APIResult = TBXReleaseMsg(
        in_hLib,
        hMsg );
}

if (hFilter != 0)
{
      /* Destroy the implicit message filter */
      TBXDestroyMsgFilter ( in_hLib, hFilter );
}

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Close library */
      APIResult = TBXCloseLib( in_hLib );
}

…
```

5 LINE INTERFACES / SERVICES

This section gives an overview and examples on how to allocate trunks using the new APIs. Theses APIs will be available on all **TelcoB**ridges Inc. products including TB640 boards. The new APIs are available starting from the Release 93.

Using the New APIs, The allocation of a trunk requires at least two steps. The first step is the allocation of the trunk's (Line Service) parent and the second step is the trunk (Line Service) allocation. For example, on TB640 board, to allocate an E1 trunk, we must allocate a line interface of type E1 then allocate the trunk itself. It should be noted, that the legacy APIs trunk allocation are supported but we strongly recommend using the new APIs to allocate the trunks.
Please note, that the term trunk and Line Service are used interchangeably in this document.

5.1 Line Interfaces

Each line interface is associated with a physical interface and it describes parameters like encoding, payload type, Line Length and other properties specific to this interface.
The line interfaces are the parents of the line services and must be allocated first. The table below depicts all line interfaces and theirs payload types.

Line Interface	Valid Index	Payload Type	TB640-DS3	TB640-STM1	TB640
STM1_OPT	0-1	AU3/AU4		√	
OC3	0-1	STS3		√	
DS3	0-2	DS3	√		
T1	0-63	T1			√
E1	0-63	E1			√
J1	0-63	J1			√

Table 1 Line interfaces

The following figures demonstrate the hierarchy between the line interfaces and underlying line services (covered in section 5.2). Figure 6 shows an example of a TB640-STM1 blade configured for an SDH optical network. Figure 7 shows an example of a TB640-STM1 blade configured for a SONET optical network. Figure 8 shows an example of a TB640-DS3 blade configuration and Figure 9 shows an example of a TB640 blade configuration. Note that these figures are examples and do <u>not</u> cover all configuration setups that are possible with the TelcoBridges' products.

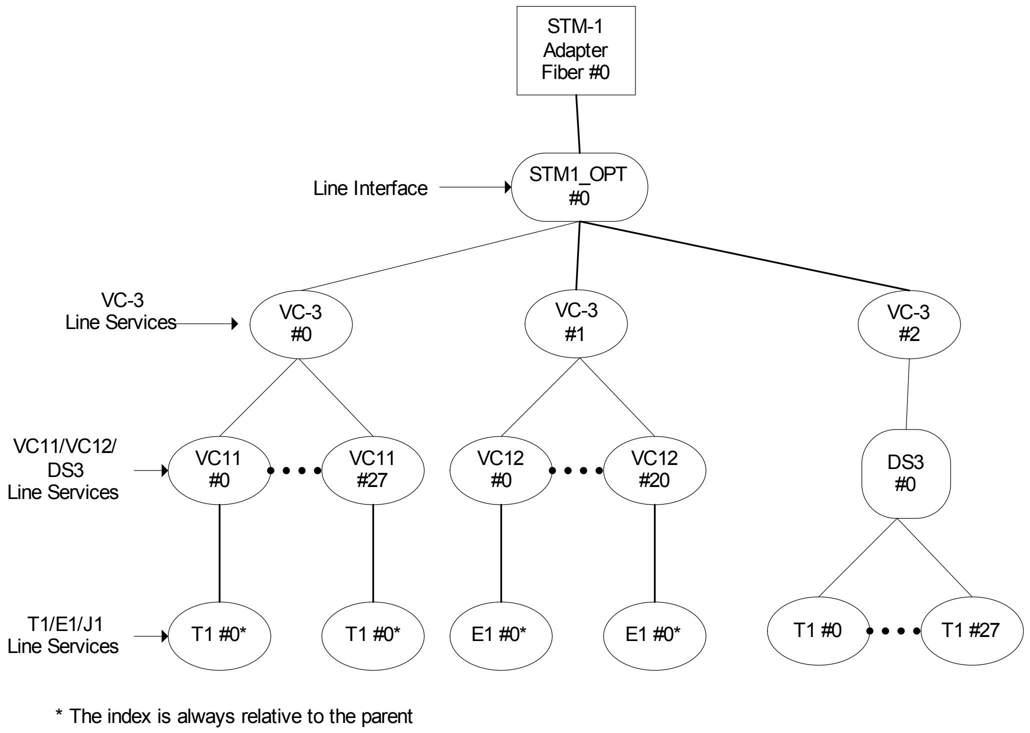


Figure 6 TB640-STM1 Line Interfaces/Services Tree (SDH configuration)

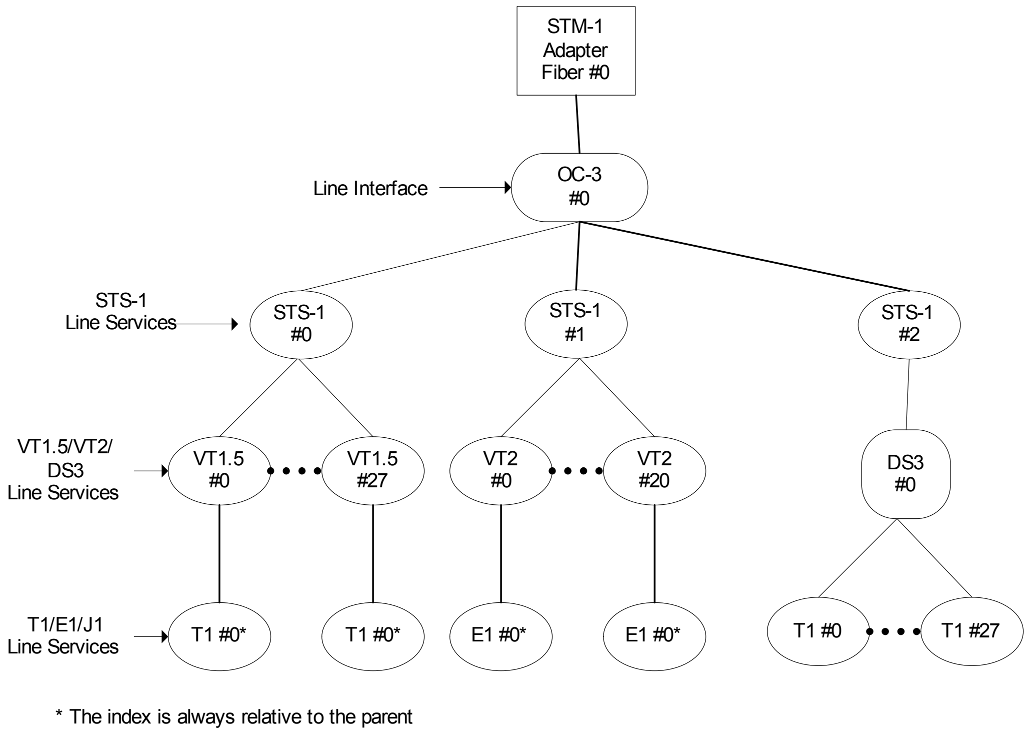


Figure 7 TB640-STM1 Line Interfaces/Services Tree (SONET configuration)

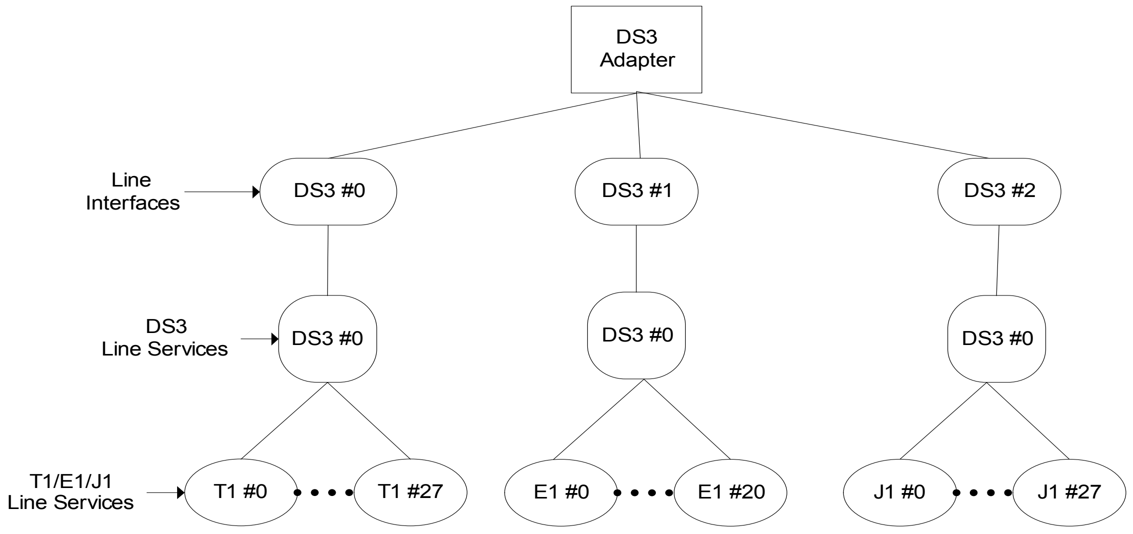


Figure 8 TB640-DS3 Line Interfaces/Services Tree

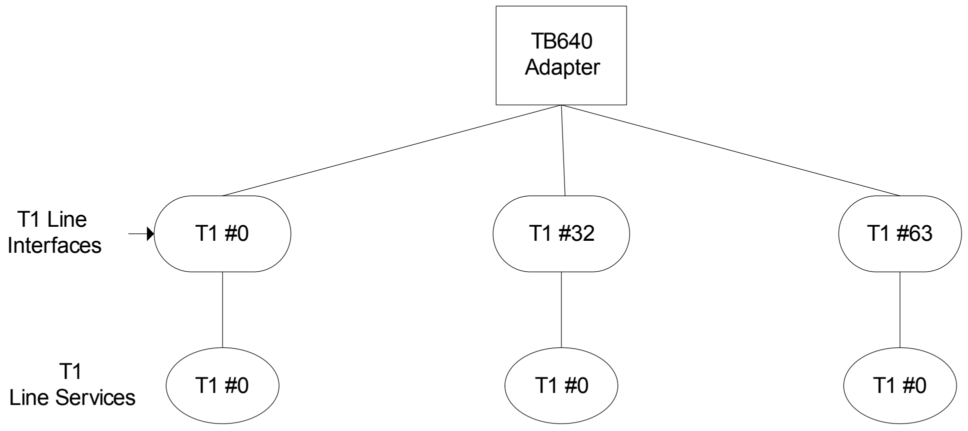


Figure 9 TB640 Line Interfaces/Services Tree

5.1.1 Allocate a Line Interface

Here is a code example showing how to allocate DS3 line interface. An implicit filter is used to retrieve response message synchronously:

…

```
TBX_RESULT_API                     APIResult;
TBX_RESULT                         AllocResult;
TBX_MSG_HANDLE                     hMsg;
TBX_FILTER_HANDLE                  hFilter;
TB640_LSM_HANDLE                   hLineInterface;
TB640_REQ_LINE_INTERFACE_OP_ALLOC*  pRequestAlloc;
TB640_RSP_LINE_INTERFACE_OP_ALLOC*  pResponseAlloc;
PTB640_DS3_LINE_INTERFACE_CFG      pLiDS3Cfg;
/* Initialize local variables */
hFilter = 0;
```

…

```
/* in_hLib is the library handle returned by TBXOpenLib call
in_hAdapter is an adapter handle returned by TBXGetAdaptersList call
in_LineInterfaceIndx is the line interface interface index to be allocated*/

/* Get message buffer */
APIResult = TBXGetMsg(
  in_hLib,
  sizeof( TB640_MSG_LINE_INTERFACE_OP_ALLOC ),
  &hMsg );

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
/* Initialize message header */
TBX_FORMAT_MSG_HEADER (
  hMsg,
  TB640_MSG_ID_LINE_INTERFACE_OP_ALLOC,
  TBX_MSG_TYPE_REQUEST,
  sizeof (PTB640_MSG_LINE_INTERFACE_OP_ALLOC),
  in_hAdapter,
  0,
  0);
```

```
/* Set the message payload */
pRequestAlloc = (PTB640_REQ_LINE_INTERFACE_OP_ALLOC)
TBX_MSG_PAYLOAD_POINTER( hMsg);
/* Fill the request */
pRequestAlloc->un32MsgVersion       = 1;
pRequestAlloc->LineInterfaceCfg.un32LineInterfaceIndex      =
in_LineInterfaceIndx;
pRequestAlloc->LineInterfaceCfg.Type                              =
TB640_LINE_INTERFACE_TYPE_DS3;
pLiDS3Cfg = &pRequestAlloc->LineInterfaceCfg.Cfg.DS3;
pLiDS3Cfg->Encoding           = TB640_DS3_LINE_INTERFACE_ENCODING_B3ZS;
pLiDS3Cfg->LineLength         = TB640_LINE_INTERFACE_LENGTH_SHORT;
pLiDS3Cfg->PayloadType        = TB640_DS3_LINE_INTERFACE_PYLD_TYPE_DS3;

/* Send the allocation message to single adapter. Note that the last argument
is non NULL. This call will return a handle on an implicit filter. This filter
can be used to retrieve the response to this request. */
APIResult = TBXSendMsg (
  in_hLib,
  hMsg,
  &hFilter );
}

/* Insert code here for the operations to be done in parallel */
/* Multi-threading is recommended to obtain best performance and take full
advantage of the asynchronous capabilities */

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
/* Use implicit filter returned by TBXSendMsg call to retrieve response
message. Call blocks for maximum 5 second. */
APIResult = TBXReceiveMsg(
  in_hLib,
  hFilter,
  5000,
  &hMsg );
}

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
/* Retrieve the message payload */
pResponseAlloc = (PTB640_RSP_LINE_SERVICE_OP_ALLOC)
  TBX_MSG_PAYLOAD_POINTER( hMsg );
AllocResult = pResponseAlloc->Result;
hLineInterface = pResponseAlloc-> hLineInterface;

if ( TBX_RESULT_SUCCESS( AllocResult ) )
{
printf( "SUCCESS: Allocation of line interface\n" );
}
else
{
printf( "FAILURE: Allocation of line interface\n" );
}
```

```
/* Release message buffer */
APIResult = TBXReleaseMsg (
  in_hLib,
  hMsg );
}

…

if (hFilter != 0)
{
      /* Destroy the implicit message filter */
      TBXDestroyMsgFilter ( in_hLib, hFilter );
}
```

The steps required to allocate T1/E1/J1 line interface is similar to the example shown above except for the specific line interface configuration parameters. For more information about the line interface allocation, please refer to HTML/CHM help file

5.1.2 Free a Line interface

Here is a code example showing how to free previously the implicit filter to retrieve response message synchronously:

…

```
TBX_RESULT_API                      APIResult;
TBX_RESULT                          FreeResult;
TBX_MSG_HANDLE                      hMsg;
TBX_FILTER_HANDLE                   hFilter;
TB640_REQ_LINE_INTERFACE_OP_FREE*   pRequestFree;
TB640_RSP_LINE_INTERFACE_OP_FREE*   pResponseFree;


/* Initialize local variables */
hFilter = 0;
```

…

```
/* in_hLib is the library handle returned by TBXOpenLib call
in_hAdapter is an adapter handle returned by TBXGetAdaptersList call
in_hLineInterface is the Line interface handle */

/* Get message buffer */
APIResult = TBXGetMsg(
  in_hLib,
  sizeof( TB640_MSG_LINE_INTERFACE_OP_FREE ),
  &hMsg );

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
/* Initialize message header */
TBX_FORMAT_MSG_HEADER (
  hMsg,
  TB640_MSG_ID_LINE_INTERFACE_OP_FREE,
  TBX_MSG_TYPE_REQUEST,
  sizeof (TB640_MSG_LINE_INTERFACE_OP_FREE),
  in_hAdapter,
  0,
  0);

/* Set the message payload */
pRequestFree = (PTB640_REQ_LINE_INTERFACE_OP_FREE)
  TBX_MSG_PAYLOAD_POINTER( hMsg );
pRequestFree->un32MsgVersion = 1;
pRequestFree->hLineInterface = in_hLineInterface;

/* Send the allocation message to single adapter. Note that the last argument
is non NULL. This call will return a handle on an implicit filter. This filter
can be used to retrieve the response to this request. */
APIResult = TBXSendMsg (
  in_hLib,
  hMsg,
  &hFilter );
}
```

```
/* Insert code here for the operations to be done in parallel */
/* Multi-threading is recommended to obtain best performance and take full
advantage of the asynchronous capabilities */

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
/* Use implicit filter returned by TBXSendMsg call to retrieve response
message. Call blocks for maximum 5 second. */
APIResult = TBXReceiveMsg (
  in_hLib,
  hFilter,
  5000,
  &hMsg );
}

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
/* Retrieve the message payload */
pResponseFree = (PTB640_RSP_LINE_INTERFACE_OP_FREE)
  TBX_MSG_PAYLOAD_POINTER( hMsg );
FreeResult = pResponseFree->Result;

if ( TBX_RESULT_SUCCESS( FreeResult ) )
{
printf( "SUCCESS: Deallocation of line interface \n" );
}
else
{
printf( "FAILURE: Deallocation of line interface\n" );
}

/* Release message buffer */
APIResult = TBXReleaseMsg (
  in_hLib,
  hMsg );
}

…
if (hFilter != 0)
{
      /* Destroy the implicit message filter */
      TBXDestroyMsgFilter ( in_hLib, hFilter );
}
…
```

5.2 Line Services

A Line service represents the payload type in a line interface. In addition, this payload can be a container for other low rate line services. For example, a DS3 line service can be channelized into T1/E1/J1 line services (previously named trunks).
The parameters that can be described by a line service are the Framing, payload type(if any), timing, Idle code etc… Please refer to HTML/CHM tb640_lsmgr.h for more information about a complete list of these parameters. The table below shows all the possible combination of line services and their parents. It is important to remember that a line service can have another line service as a parent or a line interface (which represents the physical output of the signal).

Line Service	Valid Index	Payload Type	Parent	Parent type
TB640 blade				
T1/J1	0 *	-	T1/J1	Line Interface
E1	0 *	-	E1	Line Interface
DS3 blade				
DS3	0 *	T1/E1/J1	DS3	Line Interface
T1/J1	0-27	-	DS3	Line Service
E1	0-20	-	DS3	Line Service
STM-1 blade in SONET configuration **				
STS1	0-2	VT2/VT15/DS3	OC3	Line Interface
VT2	0-20	E1/T1	STS1	Line Service
VT15	0-27	T1	STS1	Line Service
DS3	0 *	T1/E1/J1	STS1	Line Service
T1/J1	0-27	-	DS3	Line Service
T1/J1	0	-	VT2/VT15	Line Service
E1	0-20	-	DS3	Line Service
E1	0	-	VT2	Line Service
STM-1 blade in SDH configuration ***				
VC4	0	VC3/VC12/VC11	STM1_OPT	Line Interface
VC3	0-2	DS3	VC4	Line Service
VC3	0-2	VC12/VC11/DS3	STM1_OPT	Line Interface
DS3	0 *	E1/T1/J1	VC3	Line Service
VC12	0-63	E1/T1/J1	VC4	Line Service
VC12	0-20	E1/T1/J1	VC3	Line Service
VC11	0-83	T1/J1	VC4	Line Service
VC11	0-27	T1/J1	VC3	Line Service
T1/J1	0-27	-	DS3	Line Service
T1/J1	0	-	VC12/VC11	Line Service
E1	0-20	-	DS3	Line Service
E1	0	-	VC12	Line Service
* Index are relative to the parent's index ** See Figure 11 for a more explicit diagram of configuration *** See Figure 10 for a more explicit diagram of configuration				

Table 2 Line services and theirs parents

5.2.1 Allocate a line service

Here is a code example showing how to allocate a DS3 line service with a payload type E1. An implicit filter is used to retrieve response message synchronously:

```
TBX_RESULT_API                      APIResult;
TBX_RESULT                          AllocResult;
TBX_MSG_HANDLE                      hMsg;
TBX_FILTER_HANDLE                   hFilter;
TB640_LSM_HANDLE                    hLineService;
TB640_REQ_LINE_SERVICE_OP_ALLOC*    pRequestAlloc;
TB640_RSP_LINE_SERVICE_OP_ALLOC*    pResponseAlloc;
PTB640_DS3_LINE_SERVICE_CFG         pLsDS3Cfg;


/* Initialize local variables */
hFilter = 0;

/* in_hLib is the library handle returned by TBXOpenLib call
in_hAdapter is an adapter handle returned by TBXGetAdaptersList call
in_hLineInterface is the line interface handle handle */

/* Get message buffer */
APIResult = TBXGetMsg(
  in_hLib,
  sizeof( TB640_MSG_LINE_SERVICE_OP_ALLOC ),
  &hMsg );

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
/* Initialize message header */
TBX_FORMAT_MSG_HEADER (
  hMsg,
  TB640_MSG_ID_LINE_SERVICE_OP_ALLOC,
  TBX_MSG_TYPE_REQUEST,
  sizeof (TB640_MSG_LINE_SERVICE_OP_ALLOC),
  in_hAdapter,
  0,
  0);

/* Set the message payload */
pRequestAlloc = (PTB640_REQ_LINE_SERVICE_OP_ALLOC)
TBX_MSG_PAYLOAD_POINTER( hMsg);

pRequestAlloc->un64UserContext1 = 0x0;
pRequestAlloc->un64UserContext2 = 0x0;
pRequestAlloc->LineServiceCfg.un32LineServiceIndex = 0;
pRequestAlloc->LineServiceCfg.hParent     = in_hLineInterface;
pRequestAlloc->LineServiceCfg.Type        = TB640_LINE_SERVICE_TYPE_DS3;
/*Fill line service configuration structure*/
pLsDS3Cfg= &pRequestAlloc->LineServiceCfg.Cfg.DS3;
pLsDS3Cfg->fLoopTime    = TBX_FALSE;
pLsDS3Cfg->PayloadType  = TB640_DS3_LINE_SERVICE_PYLD_TYPE_E1;
pLsDS3Cfg->AISDetAlgo   = TB640_DS3_LINE_SERVICE_AIS_DET_ALGO_DEFAULT;
pLsDS3Cfg->OOFDetAlgo   = TB640_DS3_LINE_SERVICE_OOF_DET_ALGO_DEFAULT;
pLsDS3Cfg->Framing      = TB640_DS3_LINE_SERVICE_FRAMING_CBIT;

/* Send the allocation message to single adapter. Note that the last argument
is non NULL. This call will return a handle on an implicit filter. This filter
can be used to retrieve the response to this request. */
APIResult = TBXSendMsg (
  in_hLib,
  hMsg,
  &hFilter );
}

/* Insert code here for the operations to be done in parallel */
/* Multi-threading is recommended to obtain best performance and take full
advantage of the asynchronous capabilities */

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
/* Use implicit filter returned by TBXSendMsg call to retrieve response
message. Call blocks for maximum 5 second. */
APIResult = TBXReceiveMsg(
  in_hLib,
  hFilter,
  5000,
  &hMsg );
}

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
/* Retrieve the message payload */
pResponseAlloc = (PTB640_RSP_LINE_SERVICE_OP_ALLOC)
  TBX_MSG_PAYLOAD_POINTER( hMsg );
AllocResult = pResponseAlloc->Result;
hLineService = pResponseAlloc->hLineService;

if ( TBX_RESULT_SUCCESS( AllocResult ) )
{
printf( "SUCCESS: Allocation of DS3 line service\n" );
}
else
{
printf( "FAILURE: Allocation of DS3 line service\n" );
}

/* Release message buffer */
APIResult = TBXReleaseMsg (
  in_hLib,
  hMsg );
}
```

```
if (hFilter != 0)
{
      /* Destroy the implicit message filter */
      TBXDestroyMsgFilter ( in_hLib, hFilter );
}
…
```

The steps required to allocate a T1/E1/J1 line services is similar to the example shown above except for the configuration parameter and parent handle. Please refer to Table 2 for more information about the line services and theirs parents' types

5.2.2 Free a line service

Here is a code example showing how to free previously allocated line service. An implicit filter is used to retrieve response message synchronously:

```
…

TBX_RESULT_API                     APIResult;
TBX_RESULT                         FreeResult;
TBX_MSG_HANDLE                     hMsg;
TBX_FILTER_HANDLE                  hFilter;
TB640_REQ_LINE_SERVICE_OP_FREE*    pRequestFree;
TB640_RSP_LINE_SERVICE_OP_FREE*    pResponseFree;


/* Initialize local variables */
hFilter = 0;

/* in_hLib is the library handle returned by TBXOpenLib call
in_hAdapter is an adapter handle returned by TBXGetAdaptersList call
in_hLineService is the Line interface handle */

/* Get message buffer */
APIResult = TBXGetMsg(
  in_hLib,
  sizeof( TB640_MSG_LINE_SERVICE_OP_FREE ),
  &hMsg );

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Initialize message header */
      TBX_FORMAT_MSG_HEADER (
        hMsg,
        TB640_MSG_ID_LINE_SERVICE_OP_FREE,
        TBX_MSG_TYPE_REQUEST,
        sizeof (TB640_MSG_LINE_SERVICE_OP_FREE),
        in_hAdapter,
        0,
        0);
      /* Set the message payload */
      pRequestFree = (PTB640_REQ_LINE_SERVICE_OP_FREE)
        TBX_MSG_PAYLOAD_POINTER( hMsg );
      pRequestFree->un32MsgVersion = 1;
      pRequestFree->hLineService = in_hLineService;

      /* Send the allocation message to single adapter. Note that the last
      argument is non NULL. This call will return a handle on an implicit
      filter. This filter can be used to retrieve the response to this request.
      */
      APIResult = TBXSendMsg (
        in_hLib,
        hMsg,
        &hFilter );
}
```

```
/* Insert code here for the operations to be done in parallel */
/* Multi-threading is recommended to obtain best performance and take full
advantage of the asynchronous capabilities */
if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Use implicit filter returned by TBXSendMsg call to retrieve response
      message. Call blocks for maximum 5 second. */
      APIResult = TBXReceiveMsg (
        in_hLib,
        hFilter,
        5000,
        &hMsg );
}

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Retrieve the message payload */
      pResponseFree = (PTB640_RSP_LINE_SERVICE_OP_FREE)
        TBX_MSG_PAYLOAD_POINTER( hMsg );
      FreeResult = pResponseFree->Result;

      if ( TBX_RESULT_SUCCESS( FreeResult ) )
      {
            printf( "SUCCESS: Deallocation of line service");
      }
      else
      {
            printf( "FAILURE: Deallocation of line service");
      }

      /* Release message buffer */
      APIResult = TBXReleaseMsg (
        in_hLib,
        hMsg );
}

if (hFilter != 0)
{
      /* Destroy the implicit message filter */
      TBXDestroyMsgFilter ( in_hLib, hFilter );
}…
…
```

5.3 SONET and SDH

5.3.1 Mapping of payload within an SDH or SONET framing

The SDH (Synchronous digital hierarchy) standard has been developed in the 80's as a transport mechanism able to incorporate already existing transport facilities (such as E1, E1, DS3, etc.) and allow the transports of multiple data stream (including voice, packets, etc) within the same network architecture. This architecture is able to transport different rate of framings/protocols and to mix them into the same envelope while dealing with clocking transport. As part of the standardization effort, the ANSI standard body (U.S.) and the ITU (European) came to a mutual accommodate where the SONET (Synchronous optical network) standard from ANSI is considered a subset of the SDH architecture. However, both standards are only compatible from the STS-1/STM-0 data rate.

Multiple different configurations can be achieved using the TB640-STM1 blade for both SONET and SDH network. It is not mandatory to actually understand the underlying technologies behind the SDH/SONET networks to be able to use this interface type as the TB640 is taken care of the implementation. But, it is recommended understand the basics of framing modes and options to be able to configure the line interface and line services correctly. Numerous tutorials about SDH and/or SONET are available from the Internet to quickly learn about these networks.

http://www.iec.org/online/tutorials/sonet/
http://www.iec.org/online/tutorials/sdh/

The TB640-STM1 blade is designed to be a terminal point of a SDH/SONET network (not an add/drop/mux). This means that all the payload/data contained into the STM1/OC3 is demultiplexed and dropped to the local TDM bus of the blade. Therefore, it is used in a linear configuration (by opposition to a ring configuration).

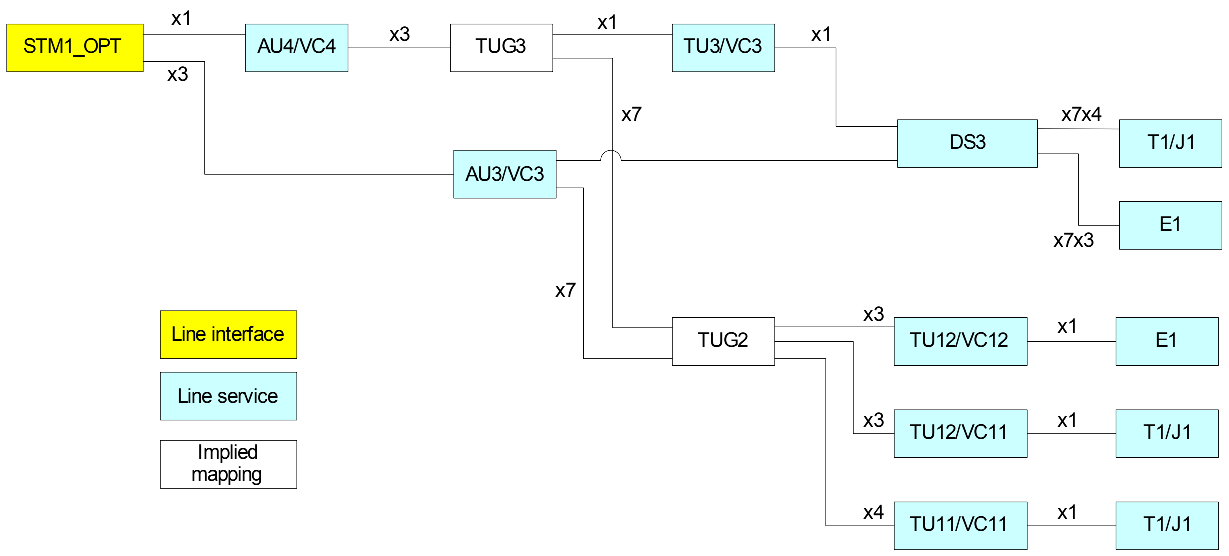


Figure 10 - SDH mapping (ITU G.707)

Figure 10 is a simplification of ITU G.707 mapping diagram of data payload into a STM-1 envelope. The yellow and blue rectangles actually show what an application would need to instantiate in order to achieve a desired SDH framing profile. Although the figure shows all possible configuration paths, it is unlikely that they'll all be used for a particular network interconnection point.

For example, a single STM-1 could contain simultaneously a DS3 frame (containing E1 or T1/J1), 20 E1s and 28 T1s. This would lead to the following configuration:

```
LI STM1_OPT #0
        LS VC3 #0
                LS DS3 #0
                        LS E1 #0 - #20
        LS VC3 #1
                LS VC12 #0 - #20
                        LS E1 #0
        LS VC3 #2
                LS VC11 #0 - #27
                        LS T1 #0
```

Another example, a single STM-1 could contain simultaneously 84 T1s. This would lead to the following configuration:

```
LI STM1_OPT #0
        LS VC4 #0
                LS VC11 #0 - #83
                        LS T1 #0
```

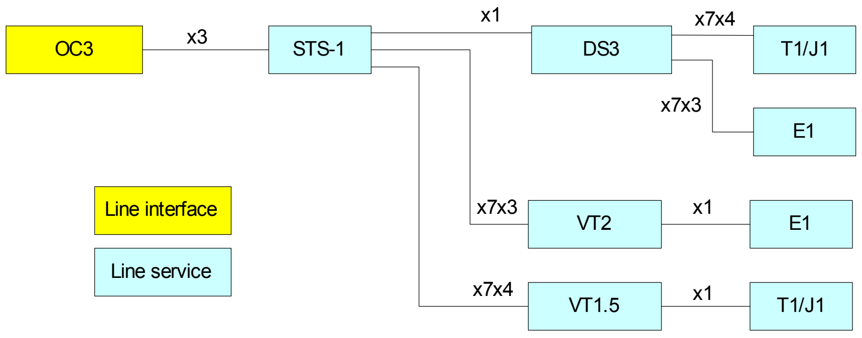


Figure 11 - SONET mapping (GR-253-CORE)

Figure 11 is a simplification of GR-253-CORE mapping diagram of data payload into an OC-3 envelope. The yellow and blue rectangles actually show what an application would need to instantiate in order to achieve a desired SONET framing profile. Although the figure shows all possible configuration paths, it is unlikely that they'll all be used for a particular network interconnection point.

For example, a single OC-3 could contain simultaneously a DS3 frame (containing E1 or T1/J1), 20 E1s and 28 T1s. This would lead to the following configuration:

```
LI OC3 #0
        LS STS-1 #0
                LS DS3 #0
                        LS T1 #0 - #27
        LS STS-1 #1
                LS VT2 #0 - #20
                        LS E1 #0
        LS VC3 #2
                LS VT1.5 #0 - #27
                        LS T1 #0
```

The SONET mapping, by opposition to the SDH model, cannot map directly all the 84 T1s into the same frame. They need to be divided in different STS-1 frames in order to insert them into the OC-3 envelope.

Both SDH and SONET standard have defined different monitoring tools to diagnose the status of the optical network. Those tools are implemented as alarm notifications and performance monitoring counters. The counters for both standards (SONET and SDH) are different although they somehow refer to the same underlying data set. All of these counters and alarms are provided by the performance monitoring API (tb640_pmalarmmgr.h). Be aware that SONET/SDH has added new notifications and counters on top of already existing one. This means that a DS3 imbedded into an SONET envelope still have its own alarms and performance monitoring counters in additions to the SONET specific alarms and counters. Basically, every line interface and line services have their own set of alarms and counters and should be monitored by an application in order to retrieve the status. Refer to the CHM/HTML API documentation for more details about those alarms.

5.3.2 Automatic protection switching (APS)

SONET and SDH standards have defined a mechanism to protect the network from physical fiber problems (unintended disconnection or destruction) and to allow traffic to be redirected to a backup data path. Multiple configurations are defined for the APS and different modes of operations are possible.

5.3.2.1 APS configuration parameters

Linear vs ring protection:

The protection scheme differs depending if the equipment is part of a SONET/SDH ring of if it is used as terminal equipment. For a ring network, the traffic is diverted from the failed data path toward the opposite direction of the ring through a backup path. Thus, the data can reach its destination by moving the other way around the ring (refer to Figure 12). Linear topology is simpler as both equipment are facing each other and decide (mutually or not according to configuration option) to switch the data traffic on a backup fiber upon detection of a failure. As the TB640-STM1 is a terminating equipment (not an add/drop/mux), only the linear configuration is supported.

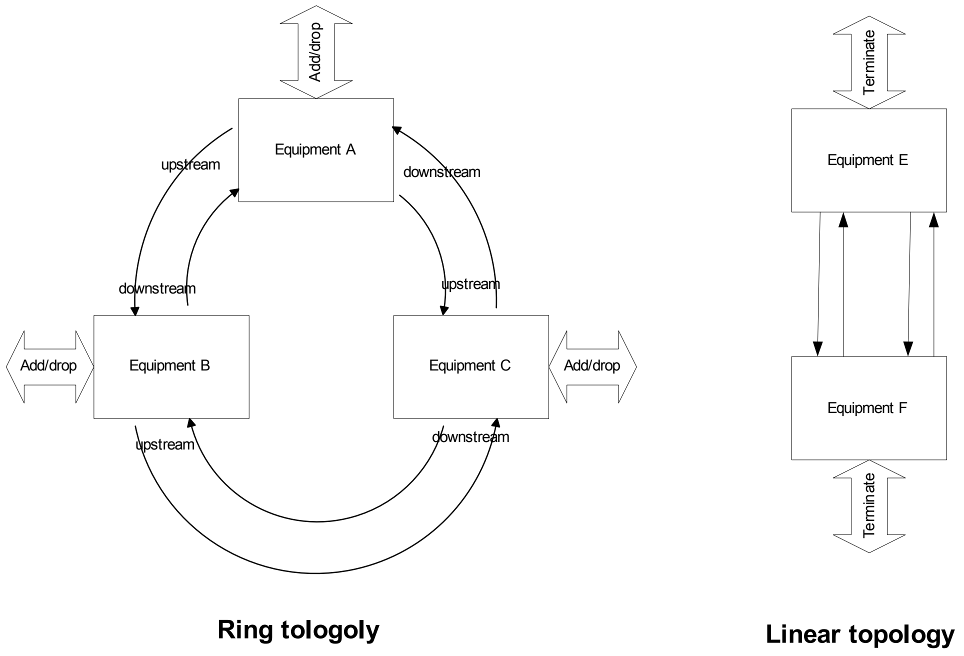


Figure 12 - Linear and ring topology

1+1, 1:n and 1:1

Linear topology leads to different configuration of protection fibers depending on the type of equipments and their capabilities (see Figure 13). The '1+1' configuration means that a supplementary fiber pair (called the 'protection channel') is dedicated to protect the primary fiber pair (referred to as 'working channel 1'). Depending of the operation (unidirectional or bidirectional) and switching configuration (revertive or non-revertive), the data/voice traffic is redirected to the protection fiber pair in case of problems detected on the primary fiber pair. Equipment of this type usually can terminate the totality of the SONET/SDH payload content to the local network. Some equipment has the capability to support more than one time the SONET/SDH payload (10 times an STM-1 rate for example) and to terminate it to a local network. In such product, more than one fiber pair is required in order to transport the payload. For these system, using a 1+1 redundancy scheme would be too much expensive as there would be a protection fiber pair for each working channel. For these applications, the ‘1:N’ scheme is used dedicated the supplementary fiber pair (the protection channel) to protect all of the N working channels (N=10 in the previous mentioned example). When one of the working channels is in problem, the voice/data traffic is redirected to the protection fiber pair. As an option, the protection channel may also transport extra traffic when it is not actively protecting a working channel (with the consequence of loosing this extra channel when a protection switch is required). In modular 1:N system, the case where N=1 exists but still behaves as a simplification of a 1:N system. Thus, extra traffic would be still be allowed providing the hardware platform supports it.

The TB640-STM1 blade supports the STM-1/OC3 level of transports (155Mbps) in a single fiber. Since this data rate can contains the complete voice/data payload from the local side of the blade, the only supported redundancy scheme is 1+1.

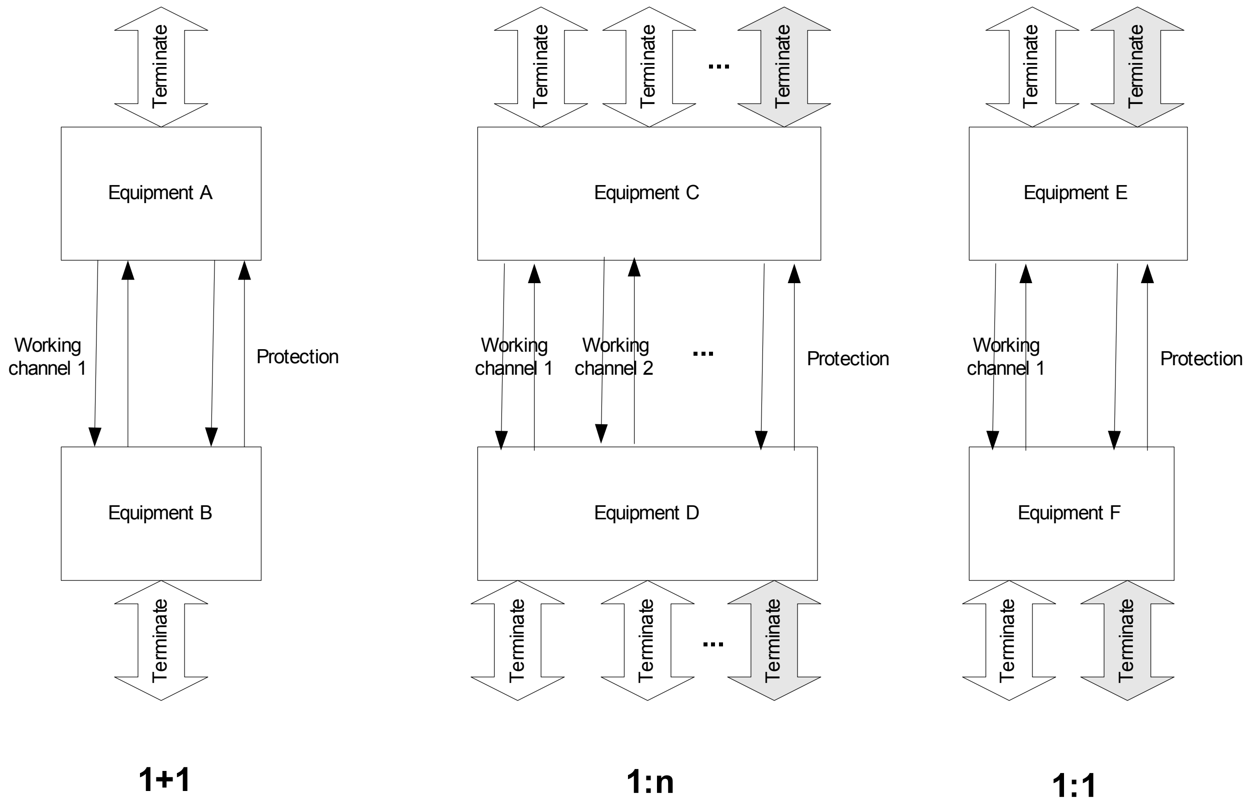


Figure 13 - 1+1, 1:n and 1:1 APS configurations

Unidirectional vs bidirectional:

This configuration parameter refers to the APS switching protocol established between the two facing equipment. In a unidirectional operation mode, the equipment that detects a failure on a working channel no. 1 (let’s assume ‘Equipment B’) requests to the remote equipment (let’s assume ‘Equipment C’ to drive the signal that specific working channel data to the protection line.

Once this request is acknowledged, Equipment B now starts to 'listen' to the protection channel but still drives its signal onto the working channel. This configuration option makes the actual bidirectional voice/data path to be splitted into two different fibers (TX on working channel 1 and RX on protection channel). This methodology has the potential of surviving a dual fiber failure as long as both failures are not in the same directions. That means that Equipment C could later requests for the working channel 3 to be driven to the protection channel. In such event, the protection channel would transport working channel 1 in one direction and working channel 3 is the other direction.

When configured for bidirectional switch-over, the APS protocol will make sure that both direction of the failed working channel is switched to the protection fiber pair (regardless if one of the directions is still working). This makes the management of the fiber a little easier as a fiber can carry (or not) traffic rather than be transporting traffic in a single direction.

The TB640-STM1 supports both of these operation modes per a configuration setting when allocating the OC3/STM1_OPT line interface.

<u>Revertive vs non-revertive:</u>
This configuration options only specifies if the traffic is automatically switched back to its original channel once the failure is cleared. When in 'non-revertive' mode, the traffic will remain on the protection fiber until manually switched back or until a failure is detected on the protection fiber (the failure would need to be of higher priority than any failure already present on the working channel). That later case would then be considered as yet another protection switching (from the protection channel to the working channel). In 'revertive' mode, the switch-back operation is done automatically by the system. To avoid creating oscillation situation (where the voice/data is constantly switched back and forth to the protection channel), a configurable probing period (called the 'wait-to-restore') is applied. If no error occurred during the probing period (which could last from seconds to hours depending on user configuration), the switch-back will occur.

The TB640-STM1 supports both of these switching modes per a configuration setting when allocating the OC3/STM1_OPT line interface. This setting applies to both the unidirectional and bidirectional mode of operations.

5.3.2.2 How to configure (or not) the APS

Both facing systems needs to exchange data for the APS protocol to work correctly. This communication link is provided through the use of dedicated bytes in the SONET/SDH framing called the K1 and K2 bytes. These bytes are only relevant on the protection channel fiber pair. Thus, to activate the APS mode, the host application needs to do two important tasks:

a. Allocate the working channel as if no protection scheme was used. This means allocating the OC3/STM1_OPT line interface and any line services required to match the network configuration.
b. Then, allocate one more line interface as the 'protection channel' (this is a configuration parameter in the line interface configuration structure). No line services is required to be allocated be allocated. The APS protection protocol is actived once a line interface configured as the 'protection channel'.

  ☞ It is **NOT** possible to allocate any underlying line services to a line interface configured as the 'protection channel'.

  ☞ Both the 'working channel' and 'protection' line interfaces need to be configured with the same APS parameters with the exception of the channel identification (working or protection) and the BER detection thresholds.

Once both line interfaces are configured, the APS protocol will make sure that the least errored fiber (or fiber pair when using bidirectional mode) will transport voice/data traffic. A notification (TB640_MSG_ID_LINE_INTERFACE_NOTIF_APS_SWITCH_OVER) is sent toward the host application every time a protection switch occurs.

5.3.2.3 Causes of a protection switching and associated alarms

New alarms have been introduced in the performance monitoring to inform about the status of the APS controller. Some of those new APS alarms combined with typical performance monitoring alarms can trigger a protection switching. Below is a description of the alarms and effect.

Failure name	Description
LOS	A loss-of-signal error failure on a line interface will trigger a SF (signal fail) request and possibly initiate a protection switchover/switchback if not overridden by a higher priority request.
LOF	A loss-of-frame error failure on a line interface will trigger a SF (signal fail) request and possibly initiate a protection switchover/switchback if not overridden by a higher priority request.
AIS	A alarm-indication-signal error failure on a line interface will trigger a SF (signal fail) request and possibly initiate a protection switchover/switchback if not overridden by a higher priority request.
APS_BYTE	This failure is signaled by the APS controller when it detects the reception of incoherent APS K1/K2 bytes or when these bytes are not stable according to the specifications.

APS_CH_MIS	The APS channel mismatch error is signaled when the APS controller detects an incoherence between the requested channel to be driven to the protection line and the actual status reported by the remote end. i.e. the remote end is not driving what the local end is expecting.
APS_MODE_MIS	The APS controller has detected that the remote end is configured in unidirectional mode but the local end is configured as bidirectional. The local end will temporarily switch to unidirectional mode. Although the system is designed to work in this condition, it is highly recommended to fix the setup configuration.
APS_FE	The APS controller sees the remote is reporting a SF (signal fail) error on the protection channel. This means that the protection line TX fiber is non-functional (thus the APS protocol is not able to pass correctly). This will trigger a SF (signal fail) request on the protection channel and possibly initiate a protection switch-back if not overridden by a higher priority request.
APS_BER_SD	The APS controller has detected a bit-error-rate higher than the SD threshold configured on the line interface. This will trigger a SD (signal defect) request and possibly initiate a protection switchover/switchback if not overridden by a higher priority request.
APS_BER_SF	The APS controller has detected a bit-error-rate higher than the SF threshold configured on the line interface. This will trigger a SF (signal fail) request and possibly initiate a protection switchover/switchback if not overridden by a higher priority request.

Table 3- APS affecting failures/alarms

These alarms/failures will generate APS requests to the APS controller. Each request is assigned a priority and a channel identification. Based on the priority and the channel id (the lowest channel ID has the priority), a new request can take precedence over an existing one and trigger a switch-over (or a switch-back). The simplest example is a working channel with a bit error rate above the configured SD threshold. This will generate an SD request and trigger a protection switch to the protection channel. However, if a LOS condition occurs on the protection channel, this will trigger a SF request which will take precedence on the actual request and will trigger a switch-back to the working channel. Thus, the APS controller will choose the least errored channel to carry the voice/data traffic.

APS request	Priority	Description
Lockout of protection	15 (highest)	This manual request is used to prevent any switchover to take place. If a protection switchover was already active, the voice/data will be switched-back to the working channel regardless of its state as this request is the highest priority. This request is available only through a shell command for debugging purposes.
Forced switch	14	This manual request is used force a switchover to the protection line. This request is available only through a shell command for debugging purposes.
SF – Low priority	12	This request indicates a ‘signal fail’ condition on a specific

		channel. It is triggered automatically by the APS controller upon detection of a failure condition and is also available through a shell command for debugging purposes.
SD – Low priority	10	This request indicates a 'signal defect' condition on a specific channel. It is triggered automatically by the APS controller upon detection of a failure condition and is also available through a shell command for debugging purposes.
Manual switch	8	This manual request is do a switchover to the protection line. This request is available only through a shell command for debugging purposes.
Wait to restore	6	This request is an internal state of the APS controller reached when configured in revertive mode and only when all error conditions have been cleared. This request stays active until a configurable probing period is reached.
Reverse request	2	This request is an internal state of the APS controller reached when configured in bidirectional mode only. It is sent on the APS communication channel to instruct the remote end to apply the same switchover/switchback as it has sent.
Do not revert	1	This request is an internal state of the APS controller reached when configured in non-revertive mode and only when all error conditions have been cleared. It instructs the remote end to keep the current switching state.
No request	0 (lowest)	This is the NULL request that is transmitted when no channel is being currently protected.

Table 4 - APS requests and priorities

6 RESOURCES

Two different types of resources exist in the TelcoBridges family of adapters, voice processing resources and channel resources. Resources can be classified into four broad categories defined by two basic attributes. The first one indicates whether the voice streams flow through the resource block. The second one indicates whether the resource is half or full duplex (bidirectional). With these two attributes, we can create four categories that describe the resource.

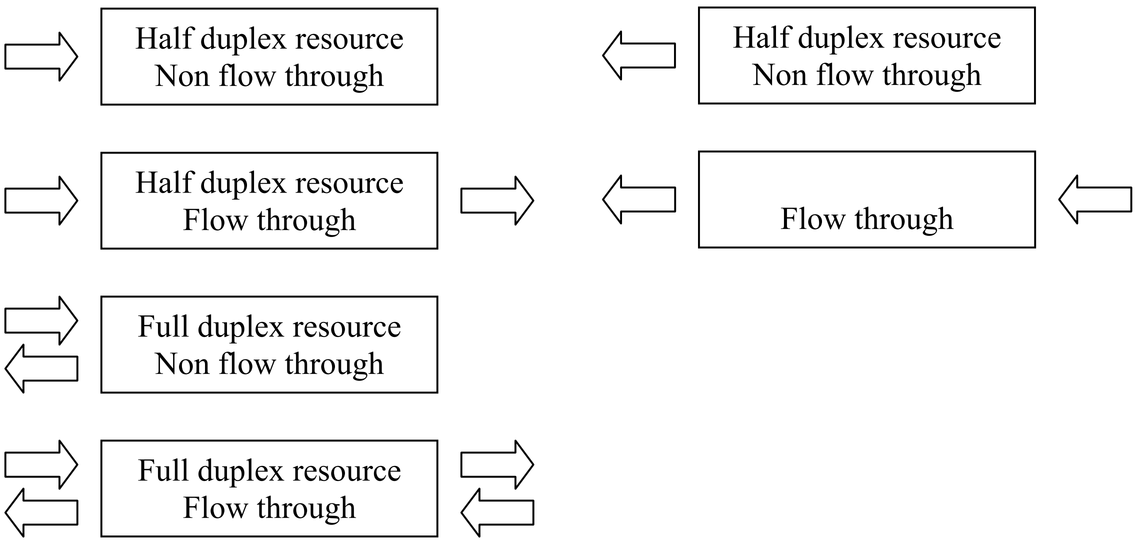


Figure 14: All resource categories

6.1 *Channel Resources*

Channel resources are used to select and configure input/output channels of the adapter. Contrarily to the voice processing resources, channel resources are not automatically allocated by the adapter. They must be allocated by the application on a channel per channel basis. The main exception to this is when the adapter signaling facilities are used. In this case, the signaling module will allocate a channel as a part of its signaling process and will return it to the application. Channels allocated by the signaling component must not be freed by the application; the adapter will free them as part of the teardown procedure. As channel resources are end points, they cannot be of type flow through, but they can be half or full duplex.

6.1.1 Trunk resources/Line Service Resources

Every trunk channel or pair of channels can be seen as a single resource that can be allocated and connected with any type of resources available on the adapter. The trunk channel resources are end points resources and therefore they are non flow through resources (the other side of the resource being the physical connection to the network equipment). It is possible to allocate trunk resources in full duplex mode.

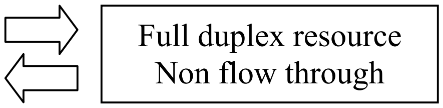


Figure 15: Trunk resource categories

Trunk timeslot allocations are different for E1 and T1/J1. One TB640 adapter can have 16 to 64 trunks, with each of these trunks having 24 to 31 timeslots.

- For E1, timeslots are allocated starting from 1 and ending at 31. If an ISDN stack is allocated, timeslot 16 is reserved for signaling information.
- For T1 and J1, timeslots are allocated starting at 1 and ending at 24. If an ISDN stack is allocated, timeslot 24 is reserved for signaling information.

CAS does not affect timeslot allocation as the signaling information is passed inband.
SS7 channel assignation will block usage of timeslots (timeslots carrying SS7 information are configurable at runtime).
So the application can allocate, for example, trunk 5, timeslot 29 as a resource that can be used in an ISDN or CAS call and connect the same resource to other trunk/timeslots, H.110 bus or other voice processing resources.

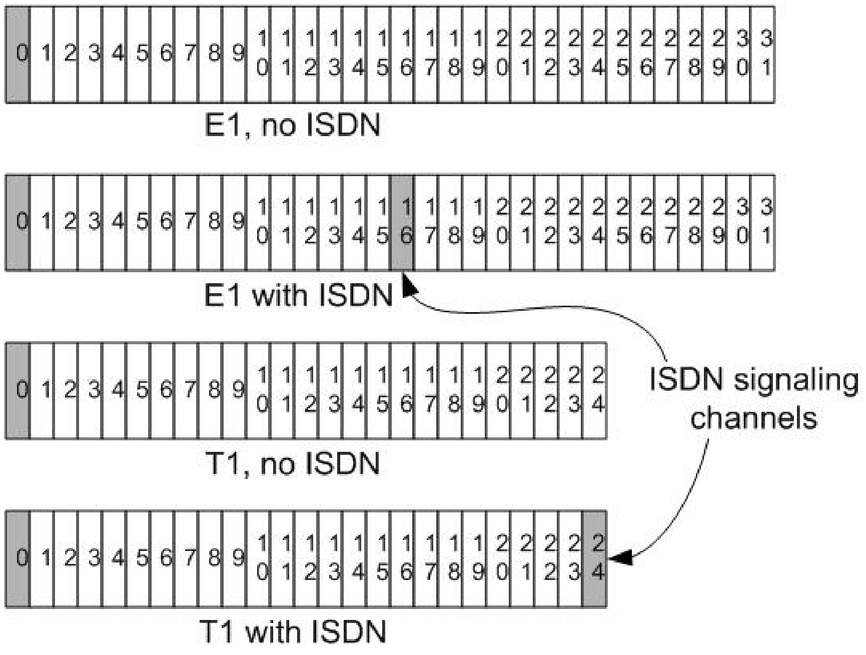


Figure 16: TB640 trunk timeslot allocation

6.1.1.1 Code example #7: Allocate trunk resource using the implicit filter

Here is a code example showing how to allocate trunk resource using the implicit filter to retrieve response message synchronously:

```
…

TBX_RESULT_API                        APIResult;
TB640_RESULT                          AllocResult;
TBX_MSG_HANDLE                        hMsg;
TBX_FILTER_HANDLE                     hFilter;
TB640_RESOURCE_HANDLE                 hTrunkRes;
PTB640_REQ_LINE_SERVICE_RES_ALLOC * pRequestAlloc;
PTB640_REQ_LINE_SERVICE_RES_ALLOC*  pResponseAlloc;

/* Initialize local variables */
hFilter = 0;

…

/* in_hLib is the library handle returned by TBXOpenLib call
in_hAdapter is an adapter handle returned by TBXGetAdaptersList call
in_hLineService is the trunk handle
in_unTimeSlot is the time slot number */

/* Get message buffer */
APIResult = TBXGetMsg(
  in_hLib,
  sizeof( TB640_MSG_LINE_SERVICE_RES_ALLOC ),
  &hMsg );

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Initialize message header */
      TBX_FORMAT_MSG_HEADER (
        hMsg,
        TB640_MSG_ID_LINE_SERVICE_RES_ALLOC,
        TBX_MSG_TYPE_REQUEST,
        Sizeof (TB640_MSG_LINE_SERVICE_RES_OP_ALLOC)
        in_hAdapter,
        0,
        0);

      /* Set the message payload */
      pRequestAlloc = (PTB640_REQ_LINE_SERVICE_RES_ALLOC)
        TBX_MSG_PAYLOAD_POINTER( hMsg);
      pRequestAlloc->un32MsgVersion = 1;
      pRequestAlloc->LineServiceResParams.hLineService = in_hLineService;
      pRequestAlloc->LineServiceResParams.un32TimeSlot = in_unTimeSlot;
      /* Send the allocation message to single adapter. Note that the last
      argument is non NULL. This call will return a handle on an implicit
      filter. This filter can be used to retrieve the response to this request.
      */
      APIResult = TBXSendMsg (
        in_hLib,
        in_hAdapter,
        &hMsg,
        &hFilter );
}
/* Insert code here for the operations to be done in parallel */
/* Multi-threading is recommended to obtain best performance and take full
advantage of the asynchronous capabilities */

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Use implicit filter returned by TBXSendMsg call to retrieve response
      message. Call blocks for maximum 5 second. */
      APIResult = TBXReceiveMsg(
        in_hLib,
        hFilter,
        5000,
        &hMsg );
}

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Retrieve the message payload */
      pResponseAlloc = (PTB640_RSP_LINE_SERVICE_RES_ALLOC)
        TBX_MSG_PAYLOAD_POINTER( hMsg );
      AllocResult = pResponseAlloc->Result;
      hTrunkRes = pResponseAlloc->hLineServiceRes;

      if ( TBX_RESULT_SUCCESS( AllocResult ) )
      {
            printf( "SUCCESS: Allocation of trunk channel resource\n" );
      }
      else
      {
            printf( "FAILURE: Allocation of trunk channel resource\n" );
      }

      /* Release message buffer */
      APIResult = TBXReleaseMsg (
        in_hLib,
        hMsg );
}
…

if (hFilter != 0)
{
      /* Destroy the implicit message filter */
      TBXDestroyMsgFilter ( in_hLib, hFilter );
}

…
```

6.1.1.2 Code example #8: Free trunk resource using the implicit filter

Here is a code example showing how to free trunk resource using the implicit filter to retrieve response message synchronously:

…

```
TBX_RESULT_API                      APIResult;
TB640_RESULT                        AllocResult;
TBX_MSG_HANDLE                      hMsg;
TBX_FILTER_HANDLE                   hFilter;
PTB640_REQ_LINE_SERVICE_RES_FREE *  pRequestFree;
PTB640_RSP_LINE_SERVICE_RES_FREE *  pResponseFree;

/* Initialize local variables */
hFilter = 0;
```

…

```
/* in_hLib is the library handle returned by TBXOpenLib call
in_hAdapter is an adapter handle returned by TBXGetAdaptersList call
in_hLineServiceRes is the trunk channel resource handle */

/* Get message buffer */
APIResult = TBXGetMsg(
  in_hLib,
  sizeof( TB640_MSG_LINE_SERVICE_RES_FREE ),
  &hMsg );

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Initialize message header */
      TBX_FORMAT_MSG_HEADER (
        hMsg,
        TB640_MSG_ID_LINE_SERVICE_RES_FREE,
        TBX_MSG_TYPE_REQUEST,
        sizeof (TB640_MSG_LINE_SERVICE_RES_FREE)
        in_hAdapter,
        0,
        0);

      /* Set the message payload */
      pRequestFree = (PTB640_REQ_LINE_SERVICE_RES_FREE)
        TBX_MSG_PAYLOAD_POINTER( hMsg );
      pRequestFree->un32MsgVersion = 1;
      pRequestFree-> hLineServiceRes= in_hLineServiceRes;




      /* Send the allocation message to single adapter. Note that the last
      argument is non NULL. This call will return a handle on an implicit
      filter. This filter can be used to retrieve the response to this request.
      */
      APIResult = TBXSendMsg (
        in_hLib,
        in_hAdapter,
        &hMsg,
        &hFilter );
}

/* Insert code here for the operations to be done in parallel */
/* Multi-threading is recommended to obtain best performance and take full
advantage of the asynchronous capabilities */

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Use implicit filter returned by TBXSendMsg call to retrieve response
      message. Call blocks for maximum 5 second. */
      APIResult = TBXReceiveMsg (
        in_hLib,
        hFilter,
        5000,
        &hMsg );
}

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Retrieve the message payload */
      pResponseFree = (PTB640_RSP_LINE_SERVICE_RES_FREE)
        TBX_MSG_PAYLOAD_POINTER( hMsg );
      FreeResult = pResponseFree->Result;

      if ( TBX_RESULT_SUCCESS( FreeResult ) )
      {
            printf( "SUCCESS: Deallocation of trunk channel resource\n" );
      }
      else
      {
            printf( "FAILURE: Deallocation of trunk channel resource\n" );
      }

      /* Release message buffer */
      APIResult = TBXReleaseMsg (
        in_hLib,
        hMsg );
}

…
if (hFilter != 0)
{
      /* Destroy the implicit message filter */
      TBXDestroyMsgFilter ( in_hLib, hFilter );
}

…
```

6.1.2 CTBUS resources

The CTBUS channel resources are end points resources and therefore they are non flow through resources. It is possible to allocate CTBUS resources in half or full duplex mode.

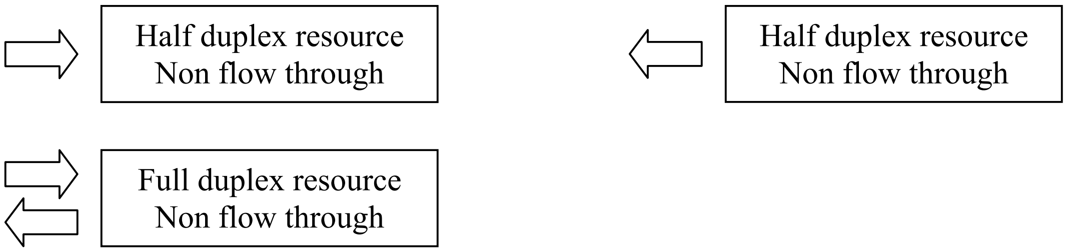


Figure 17: CTBus resource categories

The CTBUS is used to make connections from two different adapters in the system. The CTBUS is allocated in stream and timeslots when using the TB640 API. There are 32 streams of 128 timeslots each to give 4096 unidirectional timeslots (2048 conversations can take place simultaneously).
A bidirectional connection of one voice conversation between two adapters in a system requires 2 timeslots. For example: Adapter A will drive stream 4, timeslot 34 and receive from stream 20, timeslot 36, while Adapter B must drive stream 20, timeslot 36 and receive from stream 4, timeslot 34. Remember that these resources must be allocated on each of the adapters in a system (each adapter being independent) and the resource handles might be different values.

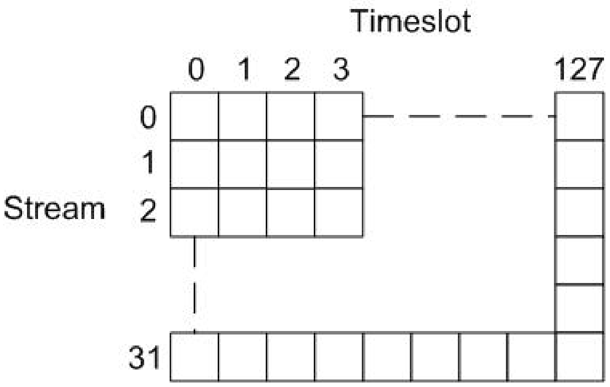


Figure 18: TB640 CTBUS timeslot allocation

Allocation and freeing of CTBUS resources can be made real-time without restrictions.
To speed up real-time processing, the CTBUS resources can be pre-allocated. When a connection is required to go through the CTBUS, one of the previously allocated timeslot pair (one receive, one transmit) can be chosen. Some premises must be validated before this can be implemented. Here is the list

- CTBUS limitation: Total number of non-blocking conversations must not exceed 2048 for one cPCI chassis. This means you can have up to 132 E1 trunks in one cPCI chassis when no ISDN or SS7 is used (4092 channels, 2046 conversations)
- Adapters in a system must always transmit to the same CTBUS timeslots. Only the receive timeslot from CTBUS will be selected.
- The application must have control over the CTBUS timeslot allocation at runtime

Each adapter will have a set of transmit to CTBUS timeslots assigned to it and the receive timeslot from CTBUS will be matched to the transmit timeslot of the other adapter.

For example, you plan having 4 adapters of 32 trunks each in a system, for a total of 128 trunks. Supposing all trunks are configured as E1 with ISDN stack at runtime, this gives 30 channels per trunk or 960 channels per adapter. You can allocate all CTBUS transmit timeslots for every adapter at startup. Adapter#1 will transmit starting at stream0:timeslot0 [0:0], up to stream7:timeslot63 [7:63], Adapter#2 will go from [7:64] to [14:127], Adapter#3 will go from [15:0] to [22:63], Adapter#4 will go from [22:64] to [29:127]. This leaves 2 streams free, [30:0-127] and [31:0-127], or 256 timeslots (8 more trunks could be added in the cPCI chassis). When a connection is made from Adapter#2 to Adapter#4, you can, for example, choose [7:64] as the transmit timeslot of Adapter#2 and [22:64] as the transmit timeslot of Adapter#4. This means the receive timeslot for Adapter#2 must be [22:64] and the receive timeslot for Adapter#4 must be [7:64].

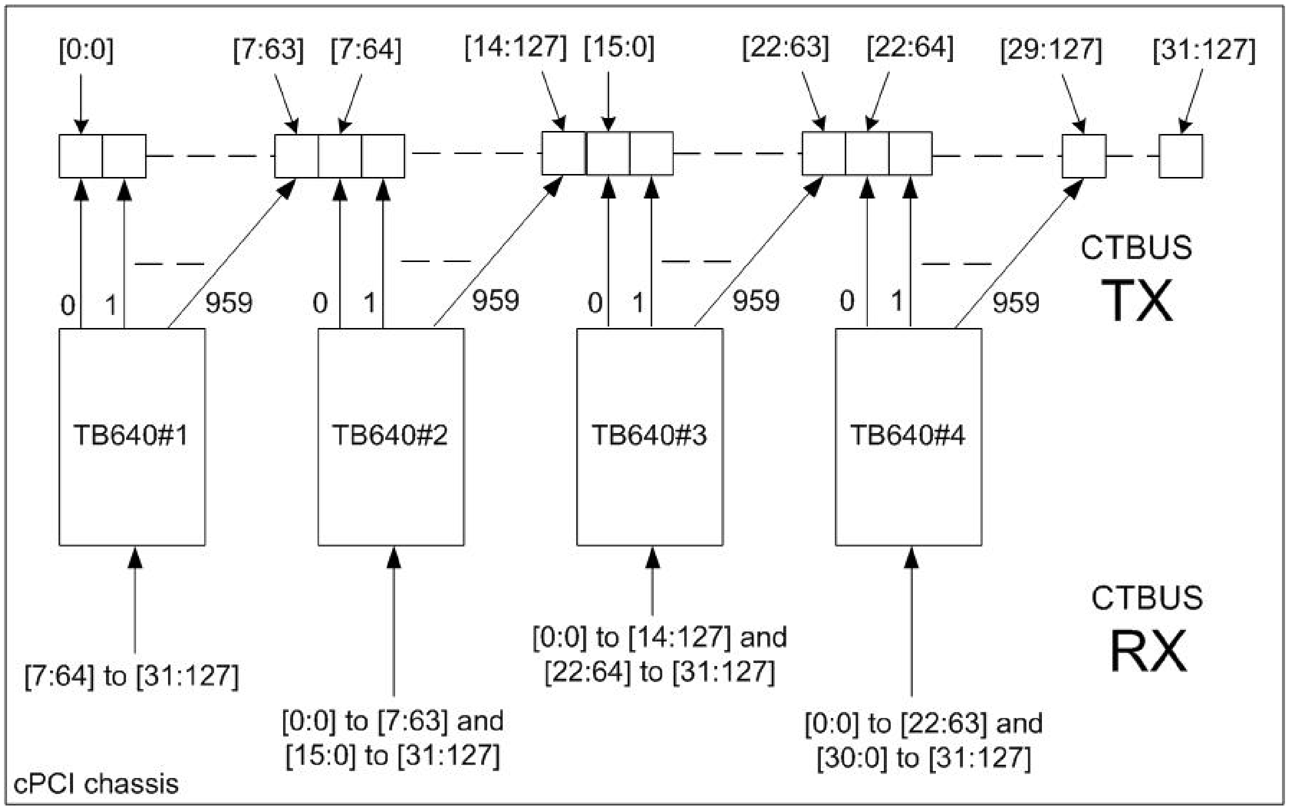


Figure 19: TB640 CTBUS allocation example

6.1.3 Stream resources

Stream resources are used to carry voice, music or other data to a specific destination on an IP network (e.g. stream server, another TB640 or an IP equipment). The stream resource actually represents the IP side (address and port) of a voice/data path. Stream resources can only be connected to Stream Voice Processing resources (any Voice Processing group which includes a stream Voice Processing resource). Output of the Stream resource can be sent to the stream server to be recorded or any other IP equipment supporting the RTP protocol. Input from a stream server or from another IP equipment using RTP can be played out onto a stream resource which can then, in turn, play it onto a stream voice processing resource.

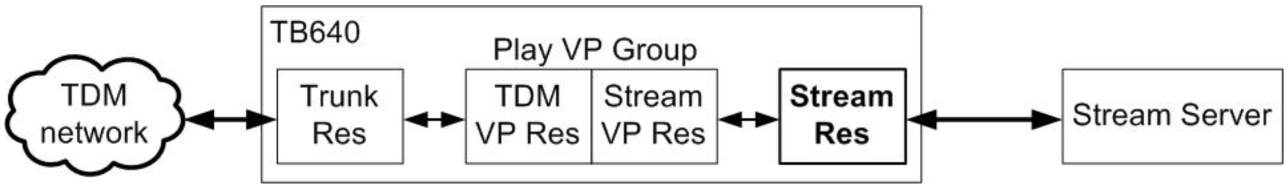


Figure 20: Stream resource high-level view

The stream resources are end points resources and therefore they are non flow through resources (the other side of the resource being the physical connection to the network equipment). Usually, it is possible to allocate stream resources in full duplex or half-duplex mode but some usage may restrict the allocation stream resource to specific configuration parameters. For example, stream resources that are to be used with VoIp voice processing group need to be full-duplex. Refer to section 6.2 for more information about voice processing groups. In case of allocation error, the TB640 will output diagnostic traces through its debugging port allowing a developer to get more information about the problem and fix it quickly.

☞ Do not confuse 'stream resource' with 'voice processing stream resource' (refer to section 6.2). The stream resource represents an endpoint onto the network (with an IP address and port) while the voice processing resource represents an endpoint onto a physical device that provides voice functionalities (i.e. DSPs or VoIp devices). You actually connect those two types of resource together to link the voice processing device to the IP network.

In addition, a few more parameters must be included. Being an IP network endpoint, a stream resource needs to be configured with destination IP addresses, source UDP port and destination UDP ports. Up to two destination IP addresses are available for configuration. Each IP address is associated with a physical Ethernet interface of the TB640. For example, 'szIPAddr0' is always associated with Ethernet interface 'eth0' while 'szIPAddr' is always associated with 'eth1' Ethernet interface when the stream resource is connected to a voice processing resource Group0 (IVR - Interactive Voice Response). Refer to section 6.2 for more information about voice processing groups. Having the 'szIpAddrx' parameters attached to a physical interface (rather than to an IP network through a netmask) is necessary for redundancy purposes.

In reference to RFC791 (IP) and RFC768 (UDP), we can associate these addresses and ports with values that will be used during the UDP or RTP communication with the remote end. During this communication, UDP packets will be sent from the TB640 (i.e. Egress path) and packets will be received by the TB640 (i.e. Ingress path) with a standard packet header format (see Figure 21). Parameters 'szIPAddr0' and 'szIPAddr1' of the stream resource will correspond to the field 'Destination Address' of the IP/UDP packet in the Egress direction. On the other hand, it is not mandatory to have the 'Source Address' from the Ingress direction to have these values. The TB640 only uses the 'Destination UDP port' and 'Source Address' in the Ingress direction to validate the received packets final destination. The parameter 'un16ToNetworkIPPort' from the stream resource corresponds to the 'Destination UDP port' of packets in the Egress direction. The parameter 'un16FromNetworkIPPort' from the stream resource corresponds to the 'Destination UDP port' of packets in the Ingress direction (i.e. the TB640 local port to which the remote end needs to send packet to). There is no way for the user to specify the 'source UDP port' for packets in the Egress direction but the TB640 will always use the same port as the 'Destination UDP port' from the Ingress direction.

```
                    1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live | Protocol(UDP) |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source UDP port      |      Destination UDP port     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Length            |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 21 - IP/UDP packet header description

- For IP/UDP Egress packets
  - Source Address = TB640 ethernet interface IP address
  - Destination addess = 'szIPAddr0' or 'szIPAddr1' parameter from the stream resource
  - Source UDP port = 'un16FromNetworkIPPort' parameter from the stream resource
  - Destination UDP port = 'un16ToNetworkIPPort' parameter from the stream resource
- For IP/UDP Ingress packets
  - Source Address = Any
  - Destination addess = TB640 ethernet interface IP address
  - Source UDP port = Any
  - Destination UDP port = 'un16FromNetworkIPPort' parameter from the stream resource

Multiple redundancy modes are supported by a stream resources (i.e. switched, redundant and none) but not all of them are relevant depending on the application or the network in which the application is deployed. Therefore, the developer must refer to the Voice processing API document to get the limitation/usage of every redundancy mode (refer to section 6.1.3.1).

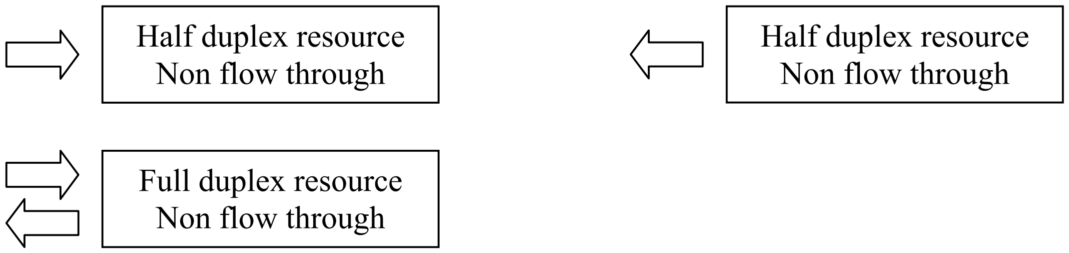


Figure 22: Stream resource categories

As another configuration parameter, different packet types are supported depending on the voice processing group to which the stream resource will be connected to. For example, IVR services are usually done using G.711 Alaw or uLaw encoding (PCM, 8 bits, 64kbps). On the other hand, VoIp services support much more compression codecs and speed rates in order to interoperate with other VoIp equipment on the network. Along with the codec type, the application needs to specify the packet rate (or duration) to be used for the desired codec. These durations vary according to the codec and the type of voice processing group that will be used in the connection. They range from 5msec duration (VoIp) up to 160msec duration (IVR). Lower packet duration values will lower latency of the stream but will increase the bandwidth usage due to the IP/UDP/RTP protocol overhead. It will also increase the processing power required to use the stream resource (i.e. more packets per seconds to process). On the other hand, greater values of packet duration will lower bandwidth usage but increase the latency of the voice conversation. Similarly, the processing power required to use such stream will decrease as well. The use of one packet duration versus the other depends on the type of application. As an example, voice mail or ring-back tone systems don't suffer from greater latency from the recorded data. In these cases, the stream resource will be configured with the longest packet duration and connected to a voice processing resources for IVR (Group0). Refer to section 6.2.2.2 for more information about voice processing groups. Having long packet duration will also allows a stream server (located on a host machine) to process more streams in parallel, thus getting a higher density of streams per host machine. In situations where bi-directional voice or data conversations are taking place, latency will affect the conversation's overall quality (and possibly create echo). To avoid this, the use of much lower packet durations (i.e. 5-20 msec) is desirable as in typical VoIp applications (using voice processing resources from Group1).

To get a complete description of configuration parameters for stream resource, the developer should refer to the Stream API document included in an official software release package.

6.1.3.1 Stream redundancy

The stream resource has the capability to support network redundancy for voice RTP stream when communicating with the stream server. Different types of redundancy are available but not all of them are supported by every voice processing groups. The stream resource redundancy allows an RTP stream to be protected against a network failure (i.e. an Ethernet switch failure) or a stream server application failure by monitoring the path between the blade and the remote entity. If the blade detects that the remote peer cannot be reached anymore, it applies the redundancy procedure described below:

6.1.3.1.1 Redundancy mode "none"

This mode actually tells the blade to turn off any redundancy scheme. Therefore, it the host application has configured 'szIPAddr0', the RTP traffic will be sent out through the first physical interface to the specified address. On the other hand, configuring an IP address in 'szIPAddr1' will force the traffic out from the second Ethernet interface. If both addresses are configured, 'szIPAddr0' only will be used.

- ☞ The RTP traffic is sent out through the first or second physical Ethernet interface (depending on the configurations of szIPAddr0 and szIPAddr1) regardless of the destination IP address. No routing is applied to the packets.
- ☞ This redundancy mode is **NOT** supported by VP group type 0 (IVR).

6.1.3.1.2 Redundancy duplicate mode

The redundancy duplicate mode is actually the simplest mode to implement. The RTP traffic is sent to the two configured IP addresses (i.e. szIPAddr0 and szIPAddr1). Since each IP address is assigned to a different physical Ethernet ports (see section 6.1.3), it is easy to build a system with different subnet going to different Ethernet switches. This method consumes twice the normal required network bandwidth (since the traffic is sent twice) but has the advantage that there will be no packet loss at the receiving end. Indeed, the receiving end will take RTP packets from one or the other RTP stream by respecting the sequence number (and obviously drop the duplicate packet). In case one of the streams fails, the other RTP stream will continue feeding the receiving end. To benefit from this redundancy method, the system needs to be protected with multiple Ethernet switch as shown in Figure 23.

- ☞ This redundancy mode is **NOT** supported by VP group type 1 (VoIP).

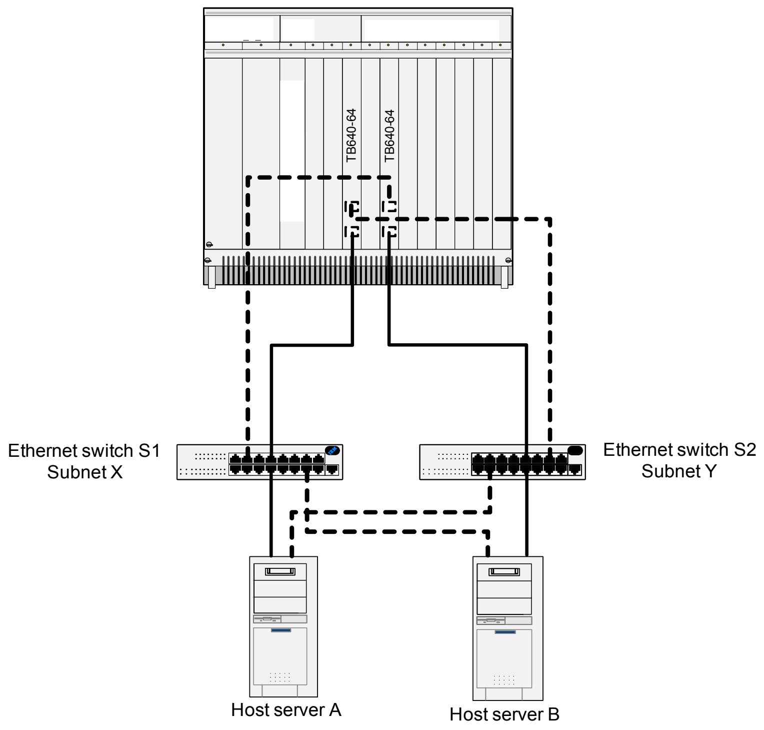


Figure 23: Redundancy network configuration

6.1.3.1.3 Redundancy switched mode

This redundancy mode uses a TelcoBridge's proprietary protocol to monitor the RTP stream to make sure that both ends can still reach each other. When the protocol detects that the remote end is no longer reachable, it switches the RTP traffic to the redundant stream. Again, this redundancy mode is useful only if the system is designed with different subnets and physical Ethernet switches connected to the physical Ethernet interfaces of the blade as shown in Figure 23 (otherwise, there will still be single point of failures in the data path). This redundancy mode has the advantage of consuming only relevant network bandwidth but has a longer turnaround time when detecting a failure. In the worst condition, the receiving end will hear a gap of up to 300msec in the voice while the failure is detected and worked-around through the redundant stream.

☞ This redundancy mode is **NOT** supported by VP group type 1 (VoIP).

6.1.4 Transcoding resources

The transcoding resources are processing resources that allow translation of the G.711 encoding from µlaw to ALaw and ALaw to µLaw. A transcoding resource is defined by the application and can be used by multiple connections without freeing it. When a connection is made with a transcoding resource definition handle, the physical resources are allocated automatically.
They are flow through type of resource. It is possible to define transcoding resource in half or full duplex mode.

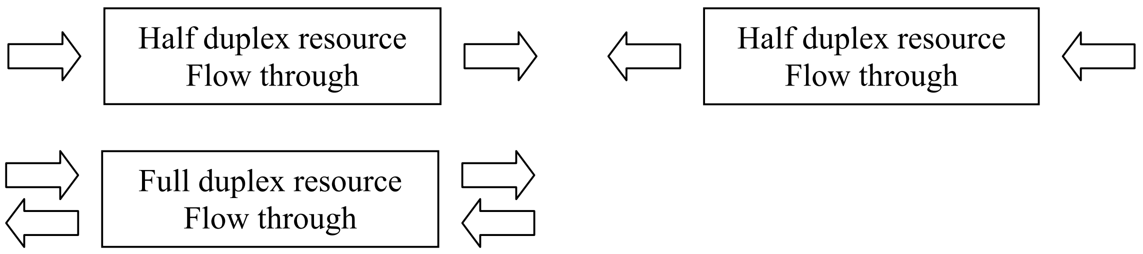


Figure 24: Transcoding resource categories

6.1.5 Multi-Blade Link resources

The Multi-Blade Link (MBL) channel resources are end points resources and therefore they are non flow through resources. It is possible to allocate Multi-Blade Link resources in half or full duplex mode.

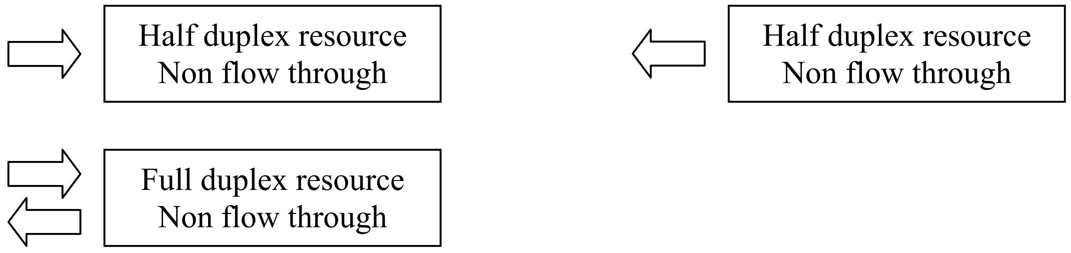


Figure 25: Multi-Blade Link resource categories

The Multi-Blade Link resources are used to interconnect channels from different TB640 adapters via a TB-MB adapter.

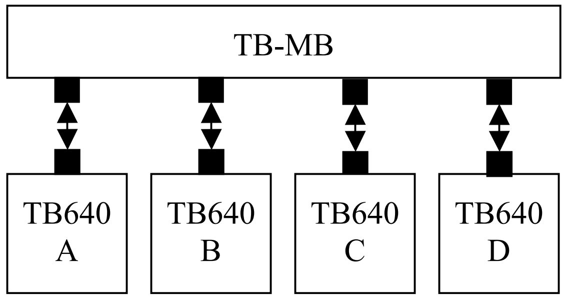


Figure 26: Multi-Blade system

The TB-MB is able to interconnect channels from up to sixteen TB640 adapters via its 16 MBL ports. Each MBL port consist of 16 input streams of 128 timeslots and 16 output streams of 128 timeslots (total of 2048 full-duplex channels). The TB640 is able to interconnect channels on redundant TB-MB adapters (an active and a standby TB-MB adapters) via its single MBL port with dual link capability (Link A and Link B).

In a Multi-Blade system like in Figure 26 to interconnect channels from adapter A to C, the following operations have to be done:

- Allocate MBL port on A (port 0);
- Allocate MBL port on C (port 0);
- Allocate MBL resource on A (port 0, stream x0, timeslot y0);
- Allocate MBL resource on C (port 0, stream x1, timeslot y1);
- Allocate MBL port on TB-MB (port where A is physically connected);

- Allocate MBL port on TB-MB (port where C is physically connected);
- Allocate MBL resource on TB-MB (port where A is physically connected, stream x0, timeslot y0);
- Allocate MBL resource on TB-MB (port where C is physically connected, stream x1, timeslot y1);
- Connect the MBL resources allocated on TB-MB;
- Connect the MBL resource on A (port 0, stream x0, timeslot y0) to another available resource on A (i.e. trunk resource, CT Bus resource, VP resource…);
- Connect the MBL resource on C (port0, stream x1, timeslot y1) to another available resource on C (i.e. trunk resource, CT Bus resource, VP resource…).

Allocation and freeing of Multi-Blade Link resources can be made real-time without restrictions.
To speed up real-time processing, the Multi-Blade Link resources can be pre-allocated. When a connection is required to go through the Multi-Blade Link, one of the previously allocated timeslot pair (one receive, one transmit) can be chosen.

It is possible to allocate MBL resource in half-duplex mode if required.

6.2 Voice processing

The Voice Processing unit is used to activate functions on streams to create applications. For example, IVR (Interactive Voice Response) for calling card application requires DTMF detection and prompt playing. By allocating Voice Processing (VP) resources in Voice Processing (VP) groups and connecting them to channel resources, specific functions can be created. These functions can run on DSPs or other devices on the TB640 such as VoIp specialized devices.

To obtain a specific voice processing function, the application needs to allocate a voice processing group which will contain one or more voice processing resources. Each resource within a same group can have different configuration parameters. Within a group, a resource represents an endpoint that can be used during a connection.

☞ It is **NOT** possible to create connections to a voice processing group. It is possible to create connections to the different resources within a voice processing group.

Regrouping different voice processing resources together within a group activates an interaction between the different resources. For example, having a voice processing TDM resource and a voice processing STREAM resource allocated together within a group creates a ‘TDM to packet’ converter with specific parameters (i.e. codecs, jitter buffers, VAD, etc). Having many voice processing TDM resources within the same group creates a conferencing function with specific parameters (i.e. AGC, volume control, number of dominant talker, etc).

☞ Do not confuse ‘stream resource’ (refer to section 6.1.3) with ‘voice processing stream resource’. The stream resource represents an endpoint onto the network (with an IP address and port) while the voice processing resource represents an endpoint onto a physical device that provides voice functionalities (i.e. DSPs or VoIp devices). You actually connect those two types of resource together to link the voice processing device to the IP network.

6.2.1 Voice processing resources

The Voice Processing unit includes many types of Voice Processing (VP) resources that falls into two different categories: resources than can are TDM-based and resources that are stream-based (packet). Regardless of the category, each voice processing resource has specific function and configuration parameters. Some resource types (e.g. pure TDM, TDM-FSK, TDM-T.38) are designed to be used in conjunction with other resource types (e.g. pure stream, stream-SFK, stream-T.38), while some resource types (e.g. TDM-flowthru, TDM-non-flowthru) can be used alone within a voice processing group. The use of all different types of resource is explained in the sections below.

Each of the VP resource can have an input and an output available for a connection. Voice Processing can be performed on input or on output. Voice processing TDM encoding is a global configuration and is done on G.711 Alaw input by default (this can be changed to µlaw upon next reboot using TB640_MSG_VP_GROUP_SET_NVPARAMS). Any resource connected to a Voice Processing resource must have the same encoding type otherwise the voice quality will be degraded. In case where the encoding would be different (e.g. connecting to a voice channel on a trunk with different encoding law), a transcoding resource must be used to ensure proper voice quality (refer to section 6.1.4).

☞ The voice processing unit global encoding affects the TDM side of a group, **NOT** the packet side.

6.2.1.1 TDM VP resource

Voice processing TDM resources (also referred as ‘pure-TDM resources’) represent a TDM connection endpoint on the internal TDM bus. Such resource can be connected in a half-duplex or full-duplex fashion to any other TDM endpoint of the blade (i.e. trunk resource, Ct-bus resource, Multi-blade resource or another voice processing TDM resource). If the Voice Processing unit is configured as Alaw (by default), TDM G.711 µlaw input can be converted to Alaw by using a transcoding TDM resource (see section 6.1.4). Usually, a voice processing TDM resource is either used in conjunction with a voice processing stream resource to create a ‘TDM to packet’ converter or is used as a TDM member part of a conference. Depending on its usage within a group, some configuration parameters may or may not be used. Refer to Voice processing API documentation for further details.

6.2.1.2 TDM Flowthru VP resource

Voice processing TDM flowthru resources are almost identical to 'pure-TDM' resources (refer to section 6.2.1.1) with the exception that they are designed to be used alone within a group. Such resource actually represents a TDM loopback endpoint that can provide voice functionalities (i.e. tone detection/generation, volume control, etc). It can be connected in a half-duplex or full-duplex fashion to any other TDM endpoint of the blade (i.e. trunk resource, Ct-bus resource, Multi-blade resource or another voice processing TDM resource). If the Voice Processing unit is configured as Alaw (by default), TDM G.711 µlaw input can be converted to Alaw by using a transcoding TDM resource (see section 6.1.4). This resource type is not available for all voice processing group types (refer to section 6.2.2).

6.2.1.3 TDM FSK VP resource

Voice processing TDM FSK resources represent the TDM connection endpoint of an FSK (Frequency shift keying) modulator/demodulator. Using a specialized voice processing API, an application can send/receive data that is modulated/demodulated to/from a 64kbps timeslot. Such resource can be connected in a half-duplex or full-duplex fashion to any other TDM endpoint of the blade (i.e. trunk resource, Ct-bus resource, Multi-blade resource or another voice processing TDM resource). If the Voice Processing unit is configured as Alaw (by default), TDM G.711 µlaw input can be converted to Alaw by using a transcoding TDM resource (see section 6.1.4). This type of resource can be used to establish a data path onto the TDM network with a remote equipment supporting the same FSK protocol. This resource type is not available for all voice processing group types (refer to section 6.2.2).

6.2.1.4 TDM Echo Near/Far VP resource

Voice processing TDM Echo near/far resources represent the TDM connection endpoint of a full-duplex TDM-only echo canceller. It is designed to be used always in pair within a group (i.e. one TDM echo near resource and one TDM echo far resource). Such resource can be connected in a half-duplex or full-duplex fashion to any other TDM endpoint of the blade (i.e. trunk resource, Ct-bus resource, Multi-blade resource or another voice processing TDM resource). If the Voice Processing unit is configured as Alaw (by default), TDM G.711 µlaw input can be converted to Alaw by using a transcoding TDM resource (see section 6.1.4). Note that such resources are not always required since other 'TDM to packet' resource combination have inline echo cancellation functions as well. These resources should be used in case where echo cancellation is required and that no in-line echo function is available. This resource type is not available for all voice processing group types (refer to section 6.2.2).

6.2.1.5 TDM T.38 VP resource

Voice processing TDM T.38 resources represent the TDM connection endpoint of a 'TDM to packet' converter specialized in Fax relay (RFC2833). Such resource can be connected in a half-duplex or full-duplex fashion to any other TDM endpoint of the blade (i.e. trunk resource, Ct-bus resource, Multi-blade resource or another voice processing TDM resource). If the Voice Processing unit is configured as Alaw (by default), TDM G.711 µlaw input can be converted to Alaw by using a transcoding TDM resource (see section 6.1.4). This resource MUST be used in conjunction with a voice processing stream T.38 resource to create a 'TDM to packet fax relay'. This resource type is not available for all voice processing group types (refer to section 6.2.2).

☞ A TB640 VpGrp1 Fax resource can only be used to process one fax transmission. Once the transmission is done, the resource needs to be de-allocated. This is due to the Fax state machine handling in the physical voip devices. Thus, an application cannot pre-allocate T.38 VpGrp1 resource as it does for voice-related resources.

6.2.1.6 Stream VP resource

Voice processing stream resources (also referred as 'pure-stream resources') represent a packet connection endpoint internal to the blade. Such resource can be connected in a half-duplex or full-duplex fashion to any other stream endpoint of the blade (i.e. stream resource). Usually, a voice processing stream resource is either used in conjunction with a voice processing TDM resource to create a 'TDM to packet' converter or is used as a stream member part of a conference.

☞ Even if it would seem logical to do so, connecting two voice processing stream resources together is not allowed. The main reason is that networking information (IP address, ports, etc.) is located into a stream resource (not a voice processing stream resource). This information is required to establish a complete connection. Thus, loopbacks on stream-side of a blade is achieved by using the IP address 127.0.0.1.

For validation purposes, there are some redundant parameters within a voice processing stream resource and a stream resource (e.g. codec type, packet duration, etc). These parameters need to be identical for a connection between the two resources to be accepted by the blade.

6.2.1.7 Stream T.38 VP resource

Voice processing stream T.38 resources represent a packet connection endpoint of a 'TDM to packet' converter specialized in Fax relay (RFC2833). Such resource can be connected in a half-duplex or full-duplex fashion to any other stream endpoint of the blade (i.e. stream resource). This resource MUST be used in conjunction with a voice processing TDM T.38 resource to create a 'TDM to packet fax relay'. This resource type is not available for all voice processing group types (refer to section 6.2.2).

☞ A TB640 VpGrp1 Fax resource can only be used to process one fax transmission. Once the transmission is done, the resource needs to be de-allocated. This is due to the Fax state machine handling in the physical voip devices. Thus, an application cannot pre-allocate T.38 VpGrp1 resource as it does for voice-related resources.

6.2.2 Voice processing groups

As mentioned before, voice processing resources can be combined within a single entity called 'group'. This group represents the voice functionalities resulting of the resources combination (i.e. conference, 'TDM-to-packet' converter, etc.). To offer the best level of flexibility and performance, a voice processing group capacity is intimately related to the hardware capabilities of the TB640 voice processing unit. Such unit has multiple formats and implementations resulting in different sets of voice capabilities. For example, a DSP mezzanine is the most appropriate platform to achieve IVR and conferencing services but is not powerful enough to do complex codec transcoding. Thus, another mezzanine using dedicated VoIp devices is available for this type of application. The voice processing API reflects these physical characteristics by having two different groups of functions: group0 (IVR) and group1 (VoIp). In the future, the APIs will also adapt to any other new voice processing devices and capabilities. The same base principles will still apply: one voice processing group can contain one or multiple voice processing resources.

When more than two VP resources are specified in a VP group, a conferencing function will be used on the inputs and the conference output will be sent to each VP resource output. This conference group can be managed using the VP group handle rather than affecting each individual resource handle. The VP group handle must be used when using messages associated with the group (for example: TB640_MSG_ID_VP_GROUP_RES_GET_PARAMS). When creating a group, you will also get a handle for each VP resources within the group allowing the application to connect them to other resource in the system (such as trunk or CTBUS resources).

Since specialized hardware have different configuration options from one group type to the other, different configuration structure are used. Some resource types are only allowed for certain group types, again, because of hardware capabilities. To ease the configuration, the voice processing API has been separated in different sets of structures regrouping the different functionalities together for every group type. Although the actual functionalities and capabilities of each group are described in details within the voice processing API guide, a brief description is given in the following sections.

☞ If proper hardware support is present on the blade, the application can make use of any voice processing group capabilities (from any groups) and combine them to build extremely flexible, non-blocking overall product.

6.2.2.1 Voice processing group0 (IVR) capabilities

Voice processing group type 0 is implemented using a DSP mezzanine and is used when the application requires one or more voice services listed below:

. Tone detection and generation (DTMF, MFR1, MFR2 Fwd/Bckwd) [inline or not]
. TDM to packet conversion (G.711 20, 80, 160 msec) and associated functions (VAD, jitter buffers, etc.)
. Interaction with TelcoBridges' Stream server (with or without RTP redundancy)
. Conferencing and associated functions (AGC, dominant talker algorithms, etc.) up to 12 members per group
. FSK/ADSI/CallerId reception and transmission capabilities

Since each voice processing function may have an influence on the voice according to its location within the processing block, overview diagrams are provided below of all supported types of voice processing resources for group type 0 (IVR). More details about their configuration parameters are provided within the voice processing API document.

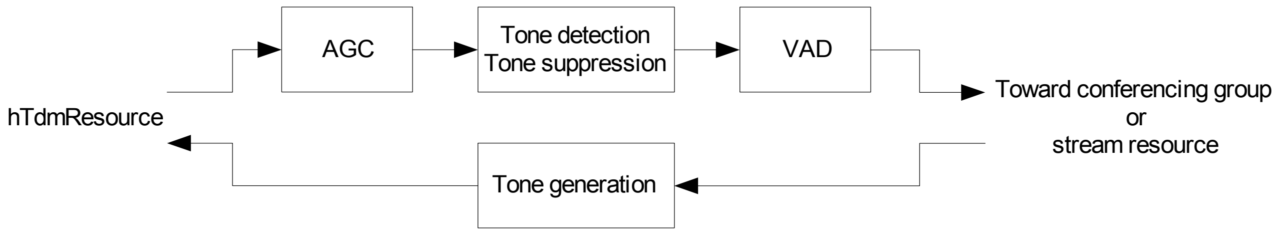


Figure 27: VP group0 TDM resource schematic

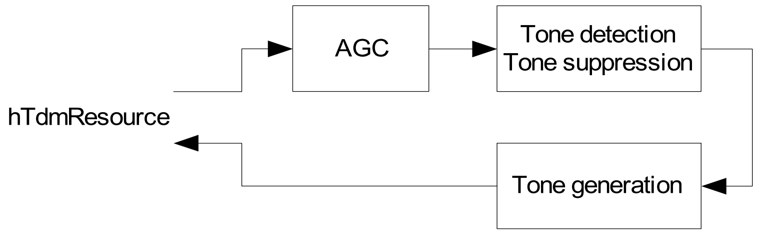


Figure 28: VP group0 TDM flowthru resource schematic

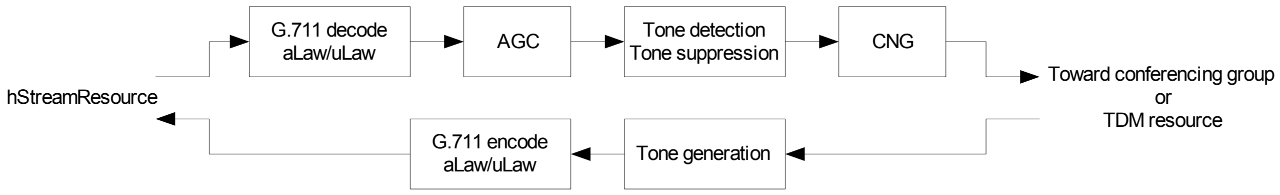


Figure 29: VP group0 stream resource schematic

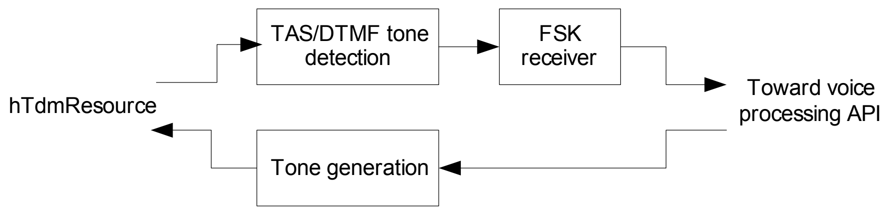


Figure 30: VP group0 TDM FSK (Rx) resource schematic

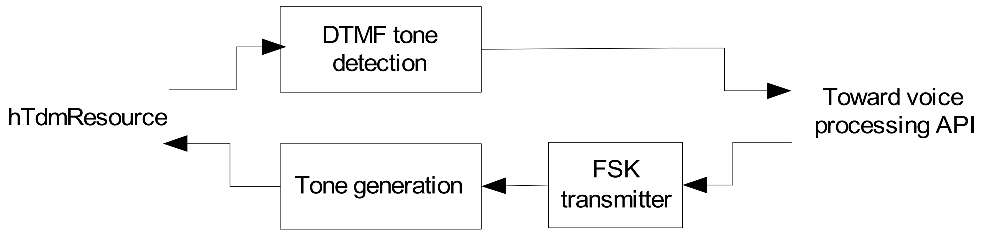


Figure 31: VP group0 TDM FSK (Tx) resource schematic

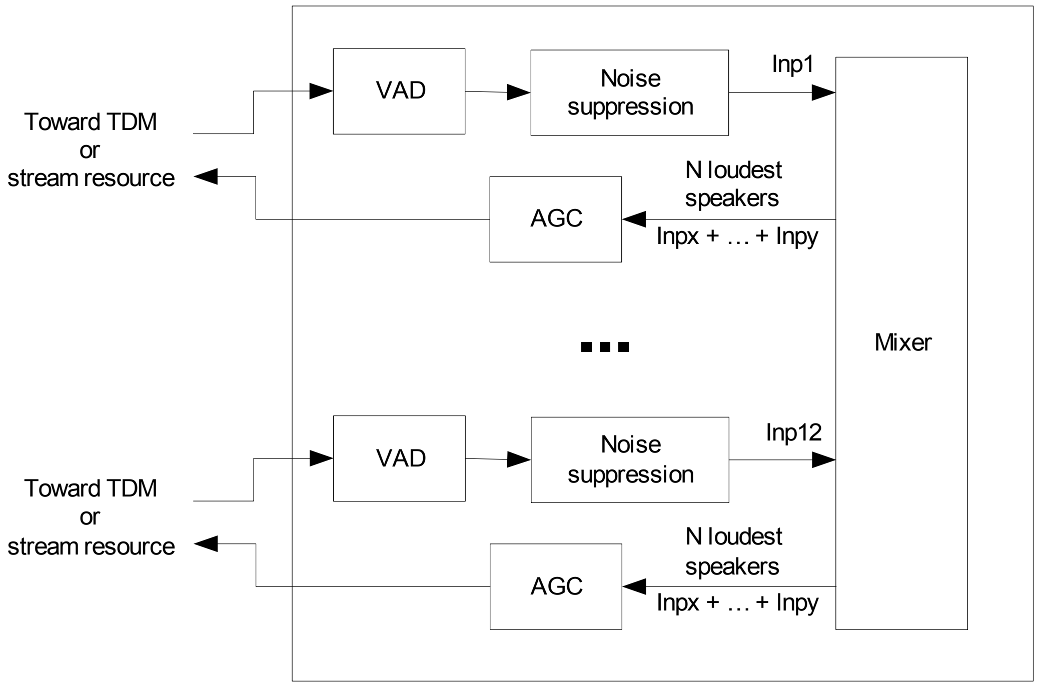


Figure 32: VP group0 conferencing schematic

6.2.2.1.1 VP group0 resource usage examples

6.2.2.1.1.1 One TDM VP resource in a VP group

A TDM stream comes-in, is processed, then comes out on the same VP resource. This is usually a simplex connection that needs to have the DTMF tones monitored or to be able to generate a tone at any time. The difference between a TDM and a TDM flowthru resource resides in the loopback capability. The pure-TDM resource will not loop the voice traffic back onto the TDM bus while the TDM flowthru will.

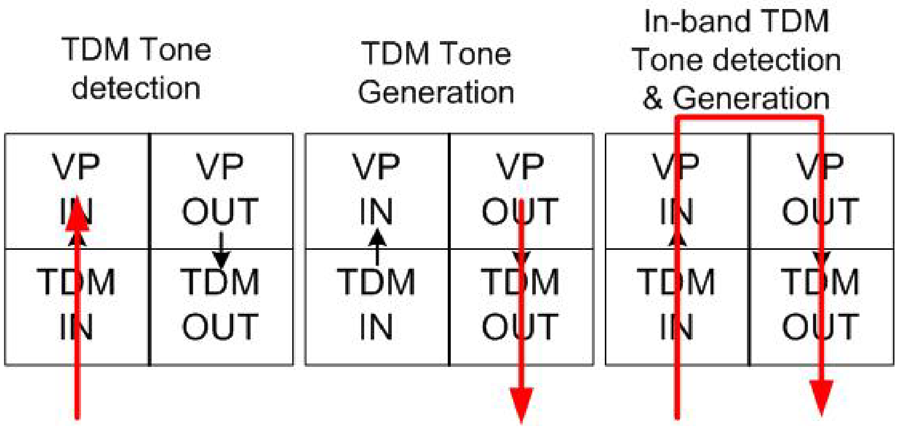


Figure 33: TDM VP resource

6.2.2.1.1.2 One Stream VP resource in a VP group

A Stream comes-in, is processed, then comes out on the same VP resource. This is usually a simplex connection that needs to have the DTMF tones monitored or to be able to generate a tone at any time. Stream tone detection and generation can be either in-band or out-of-band packets.

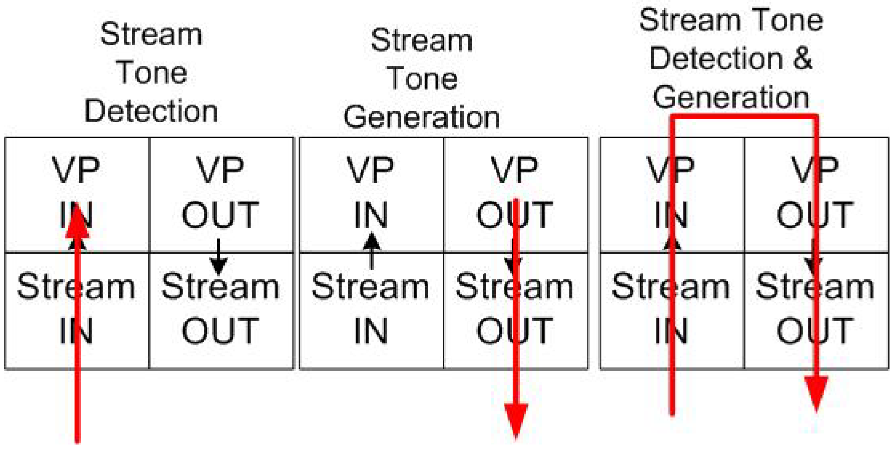


Figure 34: Stream VP resource

6.2.2.1.1.3 Two TDM VP resources in a VP group

This mode is not recommended. The system will create a conference of two members and will reach the limit of conferences available. Other ways of doing this are available.

6.2.2.1.1.4 Two Stream VP resources in a VP group

Similar to two TDM resources except that the inputs are from Stream resources instead of TDM resources. A Stream enters first VP resource, is processed, and then comes out on second VP resource, while the Stream that enters the second VP resource is processed and then comes out on the first resource. This is to monitor and generate tones both ways in a two-party conversation.

6.2.2.1.1.5 One TDM and one Stream VP resource in a VP group

A Stream input comes out as a TDM stream while the TDM stream input comes out as a Stream. This is to create a TDM to Stream transfer for recording or playing streams, and other applications. This group exceptionally requires only one VP resource.

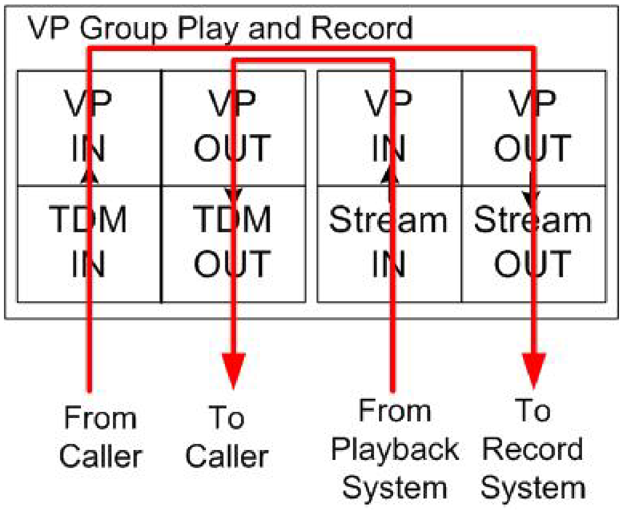


Figure 35: Play and Record

6.2.2.1.1.6 More than two TDM and/or Stream VP resources in a VP group

This is a conference between multiple TDM and/or Streams. Each incoming party will be included in the conference and the output will be sent to each of the parties. A VP group formed of 5 TDM VP resources will generate a 5 party conference. For example, Figure 36 shows one VP group composed of four VP resources. Two VP resources are configured with TDM and two others are configured as Streams. All of them are configured with tone detection and suppression.

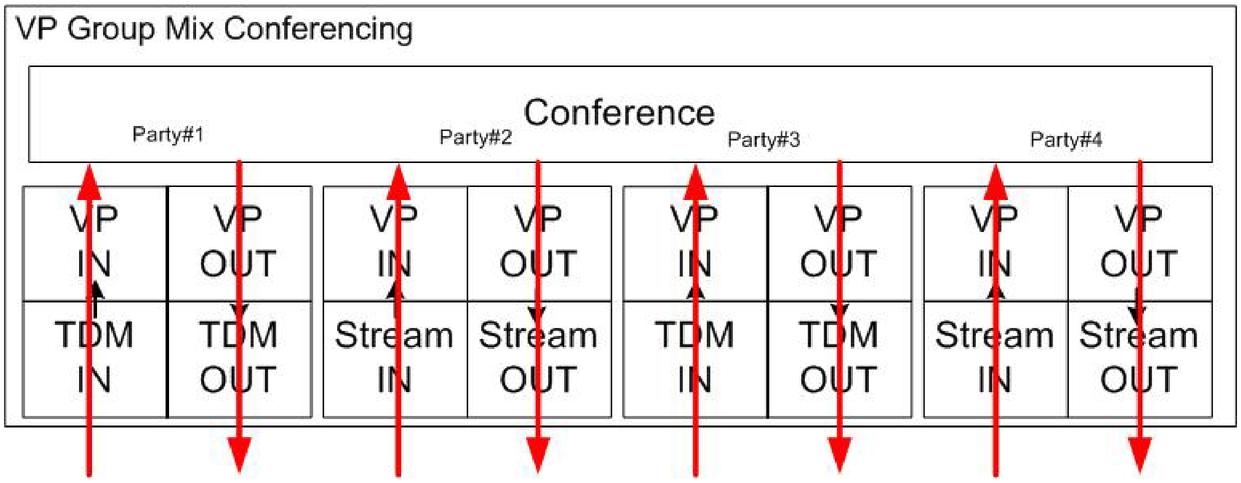


Figure 36: Four party conference with TDM and Stream inputs and tone detection and suppression

6.2.2.2 VP group0 functions

6.2.2.2.1 Tone Detection, Suppression and Generation

Tones work on both the TDM and the Stream VP resources.

6.2.2.2.1.1 Tone detection and suppression

Each tone detector must be configured to detect either DTMF, MFR1, MFR2 Forward or MFR2 backward tones. When a tone is detected, the TB640_MSG_ID_VP_TONE_NOTIF_DETECTION event will be generated to the host application. When the tone stops, the TB640_MSG_ID_VP_TONE_NOTIF_STOP event will be generated. If suppression is enabled, the tone will be removed from the voice path and silence will be inserted. You also get a timestamp in millisecond for each of those events. This can be compared to any previous event received with this timestamp.

6.2.2.2.1.2 CAS tone detection

If a CAS stack is allocated on a trunk, the CAS stack will free Voice Processing resources after the connection is made. Application must free the Voice Processing resource once the call is complete. You must not pre-allocate the resources or the CAS stack will not be able to use the resources.
When a CAS stack is allocated and a call is in progress, the tone detected will be trapped by the CAS stack and will not be sent to the host application.

6.2.2.2.1.3 Tone generation

To generate a continuous tone, the frequencies (in hertz), amplitude (in dbm) and duration of the tone must be specified. If the duration is infinite (value of -1), the TB640_ MSG_ID_ VP_TONE_STOP must be used. To generate a cadenced tone (busy, ringback, reorder, etc.), in addition to the frequency and amplitude, the on and off time must be specified and the TB640_ MSG_ID_ VP_TONE_STOP message must be used to stop the tone.

6.2.2.2.2 Automatic Gain Control (AGC)

This function is available on both TDM and Stream channels. If enabled, input amplitude will be adjusted. This is useful for recording and conferencing functions.

6.2.2.2.3 Voice Activity Detection (VAD)

This function is useful when recording (TDM to Stream transfers). When voice stops being detected, less bandwidth will be used on the network and the recording system can detect the start and end of the voice. When connecting to ASR (Automatic Speech Recognition) servers using RTP transfers, this function must be disabled.
VAD can also be used on a TDM VP resource to detect when a user starts or stops speaking. It will generate an event to a host application (if this event is registered). You will get the *TB640_EVT_VP_VAD_NOTIF_ACTIVITY* every time silence has stopped and *TB640_EVT_VP_VAD_NOTIF_SILENCE* when nothing is being heard. You also get a timestamp in millisecond for each of those events. Be careful as this might generate many events to the host application.
Three parameters are used in VAD: The noise floor, the Speech Hangover time and the Analysis Window size.
The noise floor is the threshold that defines the maximum signal level (in dBm) that is considered noise. The default value is *TB640_VP_VAD_NOISE_FLOOR_LEVEL_CHAN_DEFAULT*.
The hang time is the amount of time in milliseconds that a silence decision is held before sending the event. The default value is *TB640_VP_GROUP0_CONF_VAD_HANG_TIME_DEFAULT_VALUE.*
The window is the time to detect activity/silence (must be lower than the hang time).
The default value is *TB640_VP_GROUP0_CONF_VAD_WINDOW_SIZE_DEFAULT_VALUE*.

6.2.2.2.4 TDM to Stream switching (Record)

When recording or processing is required on a TDM stream, it can be transferred to a Stream on a given IP address and port. This Stream can then be recorded or processed by a Stream server (see the Stream Server user's guide). The Stream Server can be any system (a 1U chassis, an industrial computer or the same system running the application).

Because the interface used for this recording is RTP, it can be coupled directly with multiple ASR (Automatic Speech Recognition) or TTS (Text To Speech) servers supporting RTP.

6.2.2.2.5 Stream to TDM switching (Play)

When required to play voice, music or tone files to a TDM stream, the most efficient way is to have a Stream Server store the files and stream them out to the TB640 adapter, which then converts it to TDM. This gives full flexibility on the number of files stored and their size. Moreover, multiple Stream servers can be used to drive multiple TB640 adapters. Again, it can be coupled with available ASR or TTS engines using the RTP interface.

6.2.2.2.6 Conferencing (12 channels or less)

The conferencing function is used to mix inputs together. It can be used for phone conference, music while talking, voice recording with music, etc. The input ports can be TDM or Stream. Each connection to a conference takes one VP resource.
There is no special configuration for a conference: If more than two VP resources are specified in a VP group this means a conferencing function will be used on the inputs. A VP group has to be allocated with a specific number of VP resources, so this means the maximum conference size has to be determined when allocating the group.
The maximum size of a VP group is 12 members.
It is suggested to enable AGC when adding members to a conference.
A TDM member of a conference can be used for broadcasting (sending to an unlimited number of listeners). A connection can be put to this member and copied to any resource (Trunk, CTBUS, Multi-blade, VoIP, etc.).
A Stream member of a conference can be used for play and record in that conference. Total delay of a conference is 10ms.

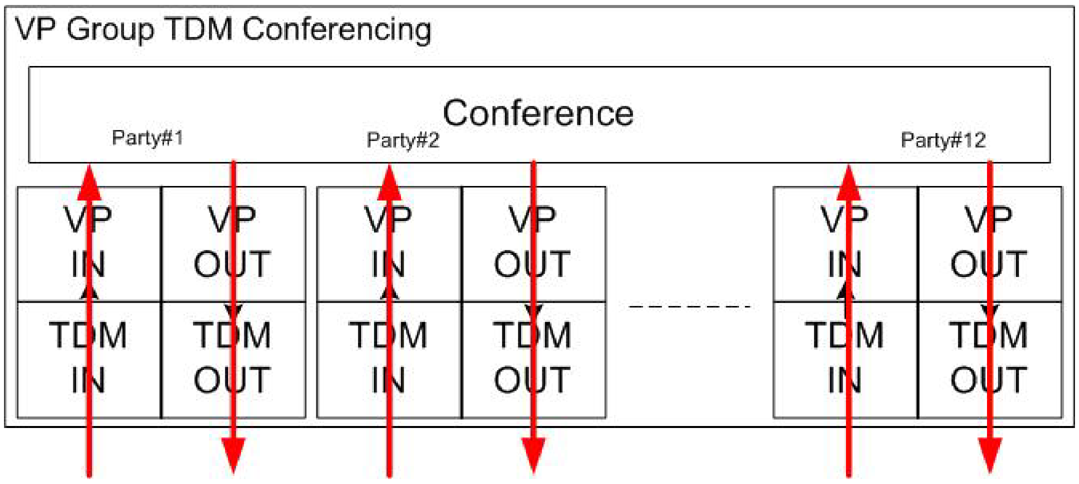


Figure 37: TDM conference

6.2.2.2.7 Bridged Conferencing (12 channels or more)

To create a conference of more than 12 channels, a conferencing tree must be created. A root conference must be created to bridge all other conferences, each of which will allocate a TDM channel for bridging. The root can bridge 12 conferences, each of which can have 11 channels (one is reserved for bridging to the root), for a total of 132 participants. Total delay for the conference is 30ms.

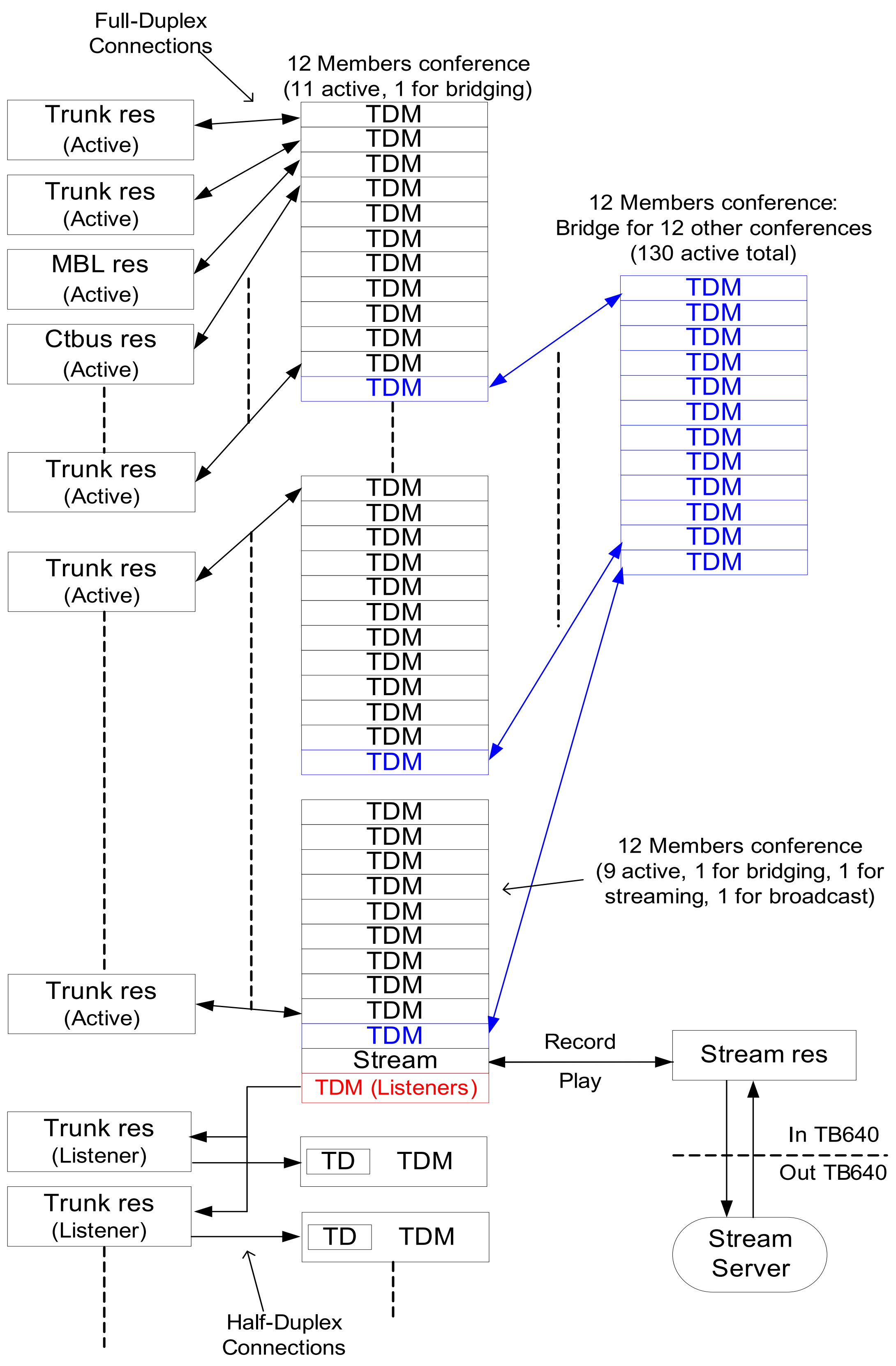


Figure 38: Bridged Conference

6.2.2.2.8 Fsk

Fsk is mainly used to transfer CallerId to Fsk compatible customer premise equipments. It is also used to exchange date, time, notifications messages presence in a message box and others application specific information to the CPE. Different protocols are built on top of Fsk. Among them are CallerId type I, CallerId type II, and ADSI.

6.2.2.2.8.1 TelcoBridges Fsk engine

TelcoBridges Fsk resources can be used to receive Fsk data from a server and to transmit Fsk data to a CPE device. TelcoBridges API allows one to build an application that will support almost any protocol layer as long as the frame format conforms to GR-30-CORE. The Fsk engine supports layer II Single Message Burst (SMB) framing format and layer II Multiple Message Burst (MMB) framing format. The TB640 has no knowledge of the layer III message formatting, thus all layer III processing must be handled by the host. For Example, GR-30-CORE SMF and MMF are layer III protocol, and must be handled by the host. Two different sets of frequency modulation are supported, ITU V.23 and Bell 202. The Fsk receiver requires a minimum of 40 mark bits, or 40 ms, and does not require any seizure bytes. The minimum Fsk receive level is -36 dBm.

A single Fsk resource supports reception and transmission but not at the same time. It must be told to either receive or transmit according to protocol in use. For example, if CallerId type I is used and the TB640 is acting as the Fsk CPE, then the resource must be toggled into Fsk reception. For the same protocol, if it's acting as the Server, then sending a request to transmit an Fsk message will automatically toggle the Fsk resource into transmission. Of course a resource can be initially opened in either reception or transmission mode.

CallerId type II and ADSI are off-hook enabled protocols and uses TAS detection and generation. For off-hook protocols, mostly when acting as a CPE, there is no way to predict in advance when an Fsk will be transmitted. A "just in time" TAS signal is used to inform the CPE that Fsk data is to be expected. Upon TAS detection, the CPE must, if it supports type II or ADSI, acknowledge it has received the TAS within a short time period. TelcoBridges DSP solutions provide a mean to detect an incoming TAS signal on almost any Voice Processing resources and to automatically fallback into Fsk reception mode upon TAS detection. When the resource does fallback, it automatically sends a pre-configured tone as acknowledge to the Fsk Server. This insures the timing requirement between the TAS and the CPE acknowledge is respected.

6.2.2.2.8.2 Acting as FSK CPE device

To emulate a CPE devices that supports on-hook only Fsk reception, the VP resource must be initially configured in Fsk reception mode. If an off-hook protocol is to be used, then the VP resource may be allocated as normal "voice mode" IVR resource with Fsk auto-reception fallback. Upon TAS detection the resource falls back into Fsk reception and can fallback again into "voice mode" after reception is completed. The TAS acknowledge signal is customizable and therefore can be DTMF-A, DTMF-B or any other application specific tone. An Fsk reception VP resource allows tone generation, DTMF and TAS tone detection.

TelcoBridges Fsk solution supports for reception of CallerId type I, CallerId type II and multi- messages burst required by ADSI GR-1273-CORE.

6.2.2.2.8.3 Acting as FSK server

To act as an Fsk server, the VP resource must be opened as Fsk, and configure in transmit mode.
An Fsk transmission VP resource allows tone generation and DTMF tone detection.
TelcoBridges Fsk solution supports for transmission of CallerId type I, CallerId type II and multi- messages burst required by ADSI GR-1273-CORE.

6.2.2.2.8.4 Supported Fsk Frame format

TelcoBridges Fsk solution fully supports GR-30-CORE and GR-1273-CORE Fsk framing formats.

```
FSK Frame Format:
   Layer II Burst Frame format: (Handled by the TB640)
   . SMB: Single Message Burst. GR-30-CORE
   . MMB: Multiple Messages Burst. Up to 5 according to GR-1273-CORE

                    <--------------- Layer II SMB Burst Format -------------------->
                    <---------------------- One Message --------------------------->
                                                <-        SMF or MMF Payload      ->
     <- Variable -> <- 1 Octet -> <- 1 Octet -> <-         Up to 255 Octets       ->
     +-------------+-------------+-------------+-------------------------------------+
SMB  |  Preamble   |  MsgType    |   MsgLen    |         Layer III Payload           |
     +-------------+-------------+-------------+-------------------------------------+

                    <------- Layer II MMB Burst Format --------->
                    <-- Up to Five Messages in a single burst -->
     +-------------+---------+--------+--------+--------+--------+
MMB  |  Preamble   | Msg # 1 | Msg #2 | Msg #3 | Msg #4 | Msg #5 |
     +-------------+---------+--------+--------+--------+--------+
                   |         |
                /--/         \---------------------------------------------------\
                |                                                                 |
                |<-------------------- Msg #1 --------------------------------->|
                                                           <- SMF/MMF Payload  ->
                 <- 1 Octet -> <- 1 Octet -> <- 1 Octet -> <- Up to 254 Octets ->
                +-------------+-------------+-------------+----------------------+...
MMB Fragment #1 |  MsgType    |   MsgLen    |   MsgNum    |  Layer III Payload   |...
                +-------------+-------------+-------------+----------------------+...

Notes on layer II:
  1) The preamble is the addition of mark bytes and seizure bytes at the beginning of a burst.
  2) Layer II MMB MsgType is ranged between 1 and 5 according to GR-1273-CORE.

Layer III Payload Frame format: (Handled by the host)
. SMF: Single Message Format. GR-30-CORE
. MMF: Multiple Messages Format. GR-30-CORE

      <------------------------ Layer III MMF Payload Format --------------------------------->
      <----------- Message #1 ------------> <----------- Message #2 ------------> ... N Mesgs->
     +-----------+----------+-------------+-----------+----------+--------------+...+-------+
MMF  | ParamType | ParamLen | Params Words | ParamType | ParamLen | Params Words |...|   N   |
     +-----------+----------+-------------+-----------+----------+--------------+...+-------+

      <------------------------ Layer III SMF Payload Format --------------------------------->
      <------------------------------ Single Message ---------------------------------------->
     +-----------+----------+-----------------------------------------------------------------+
SMF  | Application specific message                                                           |
     +-----------+----------+-----------------------------------------------------------------+
```

MMB framing format, is based upon GR-30-CORE, except that it supports for up to 5 messages aggregated in the same Fsk burst. That speeds up transmission by not having to repeat the preamble before each message. Each aggregated message must be separated by 0 or N mark bytes immediately followed by the next message.

6.2.2.2.8.5 Reception flow

The next diagram will show the API usage doing Fsk reception. The VP resource is initially opened as an IVR resource with Fsk auto-reception enabled. The server in this example is GR-1273-CORE ADSI compliant and therefore sends multiple messages burst (MMB). To act as an ADSI compliant CPE, the acknowledge digit has been configured as DTMF-A. After an undetermined time, the server needs to send Fsk data to the CPE and therefore sends a TAS signal.

6.2.2.2.8.5.1 Five Adsi messages reception in a single Burst (MMB)

	Host	TB640 acting as ADSI CPE	Voice Channel	ADSI Server
1	Host knows Fsk data is to be received	🡐 Sends TAS detection event **TB640_MSG_ID_VP_TONE_NOTIF_DETECTION - TB640_VP_DIGIT_TYPE_FSK_TAS**	🡐 TAS	

2		Fallback in Fsk reception		
3		Sends DMTF-A as TAS ack ➔	DTMF-A ➔	Server identifies the CPE to be ADSI compliant and sends FSK ADSI data
4	Reception of message 1/5 of the burst	← Sends decoded Fsk message 1 **TB640_MSG_ID_VP_FSK_NOTIF_RECEIVE**	← Fsk MMB ADSI burst	Sends MMB of 5 consecutives messages
5	Reception of message 2/5 of the burst	← Sends decoded Fsk message 2 **TB640_MSG_ID_VP_FSK_NOTIF_RECEIVE**		
6	Reception of message 3/5 of the burst	← Sends decoded Fsk message 3 **TB640_MSG_ID_VP_FSK_NOTIF_RECEIVE**		
7	Reception of message 4/5 of the burst	← Sends decoded Fsk message 4 **TB640_MSG_ID_VP_FSK_NOTIF_RECEIVE**		
8	Reception of message 5/5 of the burst	← Sends decoded Fsk message 5 **TB640_MSG_ID_VP_FSK_NOTIF_RECEIVE**		
9	Sends ack telling all 5 messages were received correctly. Send DTMF "D5" with voice fallback flag set ➔ **TB640_MSG_ID_VP_FSK_SEND_DIGIT**	Sends DTMF "D5" ➔ And VP resource falls back into "voice mode" for normal operation	DTMF "D5" ➔	Reception of acknowledge

6.2.2.2.8.6 Transmission flow

The next diagram will show the API usage doing Fsk transmission. The VP resource is initially opened as an Fsk resource in transmission mode. The CPE in this example is an ADSI GR-1273-CORE compliant device and therefore is capable of receiving multiple messages in a single burst (MMB). The TB640 offers ping-pong buffer interface for transmitting multiple messages in a single burst. Therefore, it is able to queue a maximum of 1 message while transmitting another one. Therefore, when a multiple messages burst is to be transmitted, it is necessary to wait for the first message completion event before sending the third message. Then, it is necessary to wait for second message completion event before sending the fourth one, and so on. In the following example, the host wants to sends data to CPE and the total payload fits in 3 messages.

6.2.2.2.8.6.1 Three Adsi messages transmission in a single Burst (MMB)

	Host	TB640 acting as ADSI Server	Voice Channel	ADSI CPE
1	Sends TAS **TB640_MSG_ID_VP_FSK_SEND_DIGIT**	Sends TAS ➔	TAS ➔	CPE detects TAS signal
2	Host identify CPE to be ADSI compliant, and therefore, will use a MMB to speed up transfer.	← Sends DTMF A detection event **TB640_MSG_ID_VP_TONE_NOTIF_DETECTION - TB640_VP_DIGIT_TYPE_DTMF_DIGITA**	← DTMF-A	← Sends TAS acknowledge

3	Sends message 1/3 of the burst → **TB640_MSG_ID_VP_FSK_TRANSMIT**	Encode message 1/3 and transmits it →	Fsk MMB ADSI burst →	Receives MMB of 3 consecutives messages
4	Sends message 2/3 of the burst → **TB640_MSG_ID_VP_FSK_TRANSMIT** And waits for message 1/3 end event	Queue message 2/3 until first message is completely sent		
5	Receives message 1/3 end notification,	← Sends message 1/3 end event **TB640_MSG_ID_VP_FSK_NOTIF_END** Encode message 2/3 and transmits it in the same burst →		
6	Sends message 3/3 of the burst → **TB640_MSG_ID_VP_FSK_TRANSMIT** Waits for message 2/3 end event	Queue 3/3 message until second message is completely sent		
7	Waits for message 3/3 end event	← Sends message 2/3 end event **TB640_MSG_ID_VP_FSK_NOTIF_END** Encode 3/3 and last message and transmits it in the same burst →		
8	Host knows transmission is finished and waits for CPE acknowledge	← Sends message 3/3 end event **TB640_MSG_ID_VP_FSK_NOTIF_END**		
9	Reception of acknowledge	← Sends DTMF “D3” detection event **TB640_MSG_ID_VP_TONE_NOTIF_DETECTION - TB640_VP_DIGIT_TYPE_DTMF_DIGITD TB640_MSG_ID_VP_TONE_NOTIF_DETECTION - TB640_VP_DIGIT_TYPE_DTMF_DIGIT3**	← DTMF “D3”	Acknowledges all three messages ← Sends DTMF “D3”

6.2.2.2.8.7 Transmission and reception flow

The next diagram will show the API usage doing Fsk transmission and reception on the same VP resource. The VP resource is initially opened as an Fsk resource in transmission mode. The CPE in this example uses a custom protocol enabling two way Fsk data exchange. The custom protocol is a simple one in which the server initiate the CPE reception thru a TAS signal, then sends a simple message burst, exchange acknowledges with the CPE, and then receives Fsk data from the CPE.

6.2.2.2.8.7.1 Two-Way Fsk transfer

	Host	TB640 acting as ADSI server	Voice Channel	Custom CPE
2	Sends TAS **TB640_MSG_ID_VP_FSK_SEND_DIGIT**	Sends TAS →	TAS →	CPE Receives TAS Signale
5	Host receives TAS acknowledge and	← Sends DTMF 1 detection event **TB640_MSG_ID_VP_TONE_NOTIF_DETECTION**	← DTMF-	← Sends TAS acknowledge

	acknowledge and knows CPE is able to receive one message	**- TB640_VP_DIGIT_TYPE_DTMF_DIGIT1**	DTMF-1	acknowledge and at the same time specifies it is ready to receive only one message
6	Sends 1 message ➔ with flag set to fallback in reception mode after tranmission **TB640_MSG_ID_VP_FSK_TRANSMIT**	Encode message and transmits it ➔	Fsk ADSI burst ➔	Fsk message received
11	Host knows transmission has ended and now waits for CPE acknowledge	🡐 Sends message end event **TB640_MSG_ID_VP_FSK_NOTIF_END** Falls back in Fsk reception mode		
12	Host receives acknowledge and waits a for a message coming from the CPE	🡐 Sends DTMF 1 detection event **TB640_MSG_ID_VP_TONE_NOTIF_DETECTION - TB640_VP_DIGIT_TYPE_DTMF_DIGIT1**	🡐 DTMF-“1”	🡐 Acknowledges the message
14	Successful reception of 1 Fsk message	🡐 Sends decoded Fsk message **TB640_MSG_ID_VP_FSK_NOTIF_RECEIVE**	🡐 Fsk ADSI burst	🡐 Sends 1 Fsk Message and waits for server acknowledge
14	Sends DTMF-1 **TB640_MSG_ID_VP_FSK_SEND_DIGIT** To acknowledge received message	Sends DTMF “1” ➔	➔ DTMF-“1”	Reception of acknowledge

6.2.2.2.8.8 Fsk API

TelcoBridges API usage follows. The section covers some of the main API call regarding Fsk.

6.2.2.2.8.8.1 Opening Fsk resource in V.23 reception mode

```
TBX_RESULT_API                          Result;
PTB640_MSG_VP_GROUP_ALLOC               pMsgVpResAlloc;
TBX_MSG_HANDLE                          hMsg;
TBX_FILTER_HANDLE                       hFilter;
TB640_VPGROUP_HANDLE                    hVpGroup;
TB640_RESOURCE_HANDLE                   hFskRes;
PTB640_VP_GROUP0_RES_PARAMS             pParams;

Result = TBXGetMsg (in_hLib, sizeof(*pMsgVpResAlloc), &hMsg);
if (TBX_RESULT_FAILURE (Result))
{
       TBX_EXIT_ERROR (Result, 0, "Get Msg Failed");
}
/* Clear the buffer... */
memset (TBX_MSG_PAYLOAD_POINTER (hMsg), 0, TBX_MSG_PAYLOAD_MAX_LENGTH_GET (hMsg));
/* Set the message header */
TBX_FORMAT_MSG_HEADER (
       hMsg,
       TB640_MSG_ID_VP_GROUP_ALLOC,
       TBX_MSG_TYPE_REQUEST,
       sizeof(*pMsgVpResAlloc),
       hAdapter,
       0,
       0);
/* Fill the request */
pMsgVpResAlloc = TBX_MSG_PAYLOAD_POINTER (hMsg);
pMsgVpResAlloc->Request.un32MsgVersion        = 1;
pMsgVpResAlloc->Request.GroupParams.GroupType  = TB640_VP_GROUP_TYPE_0;
pMsgVpResAlloc->Request.un32NbResources        = 1;

pParams = &pMsgVpResAlloc->Request.aResourcesParam[0].Group0;
pParams->un64UserContext1                      = un64UserCtx;
pParams->ResType                               = TB640_VP_RES_TYPE_TDM_FSK;
pParams->TdmFsk.Level                          = TB640_VP_FSK_LEVEL_DEFAULT;
pParams->TdmFsk.Type                           = TB640_VP_FSK_TYPE_V23;
pParams->TdmFsk.Mode                           = TB640_VP_FSK_MODE_RX;
pParams->TdmFsk.un16NumMarkBytes = TB640_VP_GROUP0_FSK_NUM_MARK_BYTES_MAX_VALUE;
pParams->TdmFsk.un16NumSeizureBytes =
                                 TB640_VP_GROUP0_FSK_NUM_SEIZURE_BYTES_MAX_VALUE;
Result = TBXSendMsg (in_hLib, hMsg, &hFilter);
...
Result = TBXReceiveMsg (in_hLib, hFilter, 1000 /*ms*/, &hMsg);
...
pMsgVpResAlloc = TBX_MSG_PAYLOAD_POINTER (hMsg);
...
Result = pMsgVpResAlloc->Response.Result;
hFskRes = pMsgVpResAlloc->Response.ahResources[0];
hVpGroup = pMsgVpResAlloc->Response.hVPGroup;
```

6.2.2.2.8.8.2 Opening IVR resource with B202 Fsk Auto-Reception

```
TBX_RESULT_API                          Result;
PTB640_MSG_VP_GROUP_ALLOC               pMsgVpResAlloc;
TBX_MSG_HANDLE                          hMsg;
TBX_FILTER_HANDLE                       hFilter;
TB640_VPGROUP_HANDLE                    hVpGroup;
TB640_RESOURCE_HANDLE                   hIvrRes;
PTB640_VP_GROUP0_RES_PARAMS             pParams;
PTB640_VP_GROUP0_FSK_RX_AUTO_PARAMS     pFskRxAuto;

Result = TBXGetMsg (in_hLib, sizeof(*pMsgVpResAlloc), &hMsg);
if (TBX_RESULT_FAILURE (Result))
{
       TBX_EXIT_ERROR (Result, 0, "Get Msg Failed");
}
/* Clear the buffer... */
memset (TBX_MSG_PAYLOAD_POINTER (hMsg), 0, TBX_MSG_PAYLOAD_MAX_LENGTH_GET (hMsg));
/* Set the message header */
TBX_FORMAT_MSG_HEADER (
       hMsg,
       TB640_MSG_ID_VP_GROUP_ALLOC,
       TBX_MSG_TYPE_REQUEST,
       sizeof(*pMsgVpResAlloc),
       hAdapter,
       0,
       0);
/* Fill the request */
pMsgVpResAlloc = TBX_MSG_PAYLOAD_POINTER (hMsg);
pMsgVpResAlloc->Request.un32MsgVersion         = 1;
pMsgVpResAlloc->Request.GroupParams.GroupType  = TB640_VP_GROUP_TYPE_0;
pMsgVpResAlloc->Request.un32NbResources        = 1;

pFskRxAuto = &pMsgVpResAlloc->Request.GroupParams.Group0.FskRxAuto;
pFskRxAuto->fEnabled                    = TBX_TRUE;
pFskRxAuto->Type                        = TB640_VP_FSK_TYPE_VB202;
pFskRxAuto->Level                       = TB640_VP_FSK_LEVEL_DEFAULT;
pFskRxAuto->TasAcknowledge.Digit        = TB640_VP_DIGIT_TYPE_DTMF_DIGITA;
pFskRxAuto->TasAcknowledge.Level        = TB640_VP_TONE_LEVEL_MINUS_10_DBM;
pFskRxAuto->TasAcknowledge.un32OnTimeMs = 70;

pParams = &pMsgVpResAlloc->Request.aResourcesParam[0].Group0;
pParams->ResType                        = TB640_VP_RES_TYPE_TDM;
pParams->Tdm.Agc.fEnabled               = TBX_FALSE;
pParams->Tdm.Tone.fDetectionEnabled     = TBX_TRUE;
pParams->Tdm.Tone.fGenerationEnabled    = TBX_TRUE;
pParams->Tdm.Tone.fSuppressionEnabled   = TBX_FALSE;
pParams->Tdm.Tone.DetectionTypeMask     = TB640_VP_TONE_DETECT_TYPE_DTMF;
pParams->Tdm.Vad.fEnabled               = TBX_FALSE;
Result = TBXSendMsg (in_hLib, hMsg, &hFilter);
...
Result = TBXReceiveMsg (in_hLib, hFilter, 1000 /*ms*/, &hMsg);
...
pMsgVpResAlloc = TBX_MSG_PAYLOAD_POINTER (hMsg);
...
Result = pMsgVpResAlloc->Response.Result;
hIvrRes = pMsgVpResAlloc->Response.ahResources[0];
hVpGroup = pMsgVpResAlloc->Response.hVPGroup;
```

6.2.2.2.8.8.3 Sending Fsk message in a single message burst (SMB)

```
User_Result
User_FillFskMessageFunction(
  IN PTB640_VP_FSK_MESSAGE pFskMsg,
  IN TB640_RESOURCE_HANDLE hRes );
...

TBX_RESULT_API                          Result;
TBX_MSG_HANDLE                          hMsg;
PTB640_MSG_VP_FSK_TRANSMIT              pMsgVpFskTx;
TBX_FILTER_HANDLE                       hFilter;

Result = TBXGetMsg( pThreadCtx->hLib, sizeof(*pMsg), &hMsg2 );
if (TBX_RESULT_FAILURE (Result))
{
      TBX_EXIT_ERROR (Result, 0, "Get Msg Failed");
}
/* Clear the buffer... */
memset (TBX_MSG_PAYLOAD_POINTER (hMsg), 0, TBX_MSG_PAYLOAD_MAX_LENGTH_GET (hMsg));

/* Set the message header */
TBX_FORMAT_MSG_HEADER (
      hMsg,
      TB640_MSG_ID_VP_FSK_TRANSMIT,
      TBX_MSG_TYPE_REQUEST,
      sizeof(*pMsgVpFskTx),
      hAdapter,
      0,
      0);
/* Fill the request */
pMsgVpFskTx = TBX_MSG_PAYLOAD_POINTER (hMsg);
pMsgVpFskTx->Request.un32MsgVersion     = 1;
pMsgVpFskTx->Request.fVoiceFallBack     = TBX_FALSE;
pMsgVpFskTx->Request.fFskRxFallBack     = TBX_FALSE;
pMsgVpFskTx->Request.fMsgEndEvent       = TBX_TRUE; /* We want the end event */
pMsgVpFskTx->Request.un8BurstSize       = 1; /* Single Message Burst */
pMsgVpFskTx->Request.un8MsgNumber       = 0;
pMsgVpFskTx->Request.hVPResource        = hFskRes;

User_FillFskMessageFunction( &pMsgVpFskTx->Request.FskMsg, hFskRes );
...
TBXSendMsg( pThreadCtx->hLib, hMsg2, &hFilter );
...
Result = TBXReceiveMsg (in_hLib, hFilter, 1000 /*ms*/, &hMsg);
...
pMsgVpFskTx = TBX_MSG_PAYLOAD_POINTER (hMsg);
...
Result = pMsgVpResAlloc->Response.Result;
```

6.2.2.3 VP group0 Applications

Some applications are a combination of the previous functions. Here are a few examples.

6.2.2.3.1 IVR

IVR requires DTMF detection on input and playing a prompt to guide the user in his decisions. This configuration uses one VP resource.

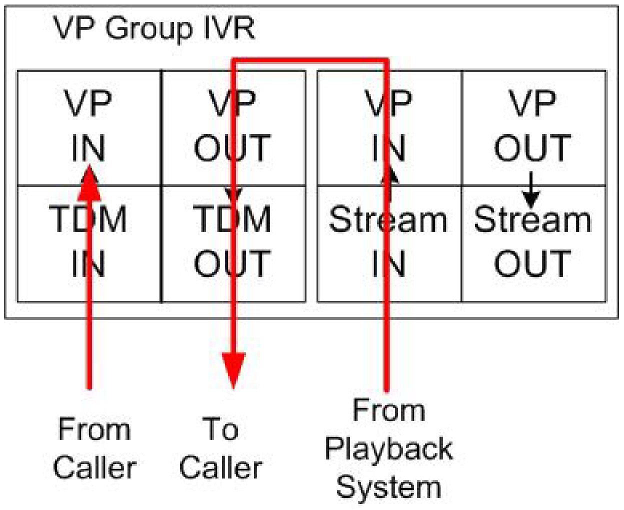


Figure 39: IVR application

6.2.2.3.2 Voice recording with music

In this application, we want the user to record his voice with some music in the background. User selects music by IVR and then the music starts, played on one resource using the Stream. The caller can then start talking or singing while the mix of voice and music is being transferred to a recording system. When the caller presses a DTMF to stop recording, he can then hear the recording back.
This configuration uses three voice processing resources for recording and one for playing.

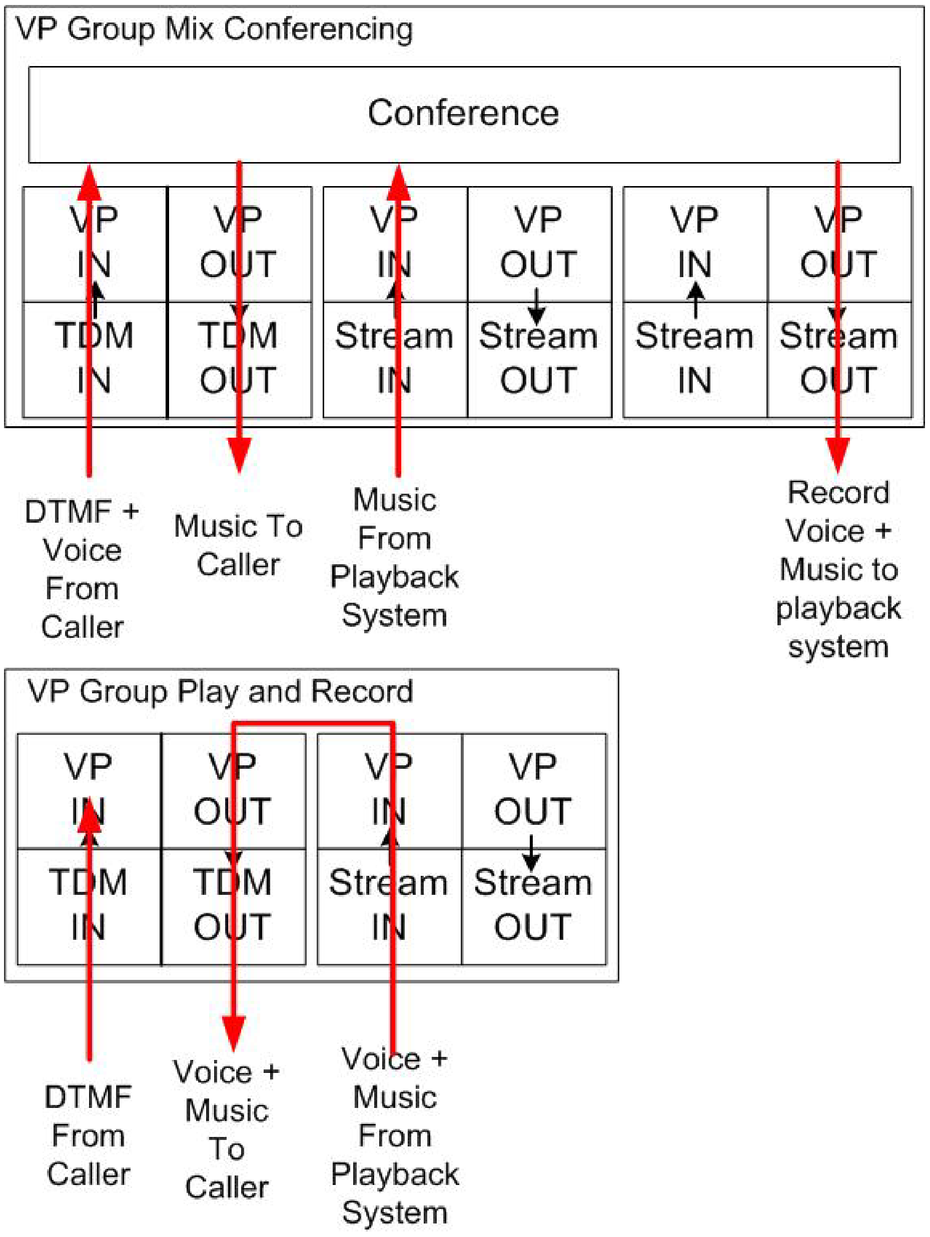


Figure 40: Voice recording with music application

6.2.2.3.3 Recording a conference

In this application, we want to record a conference. To do this, you must assign one more stream member in the group and it will be used for recording.

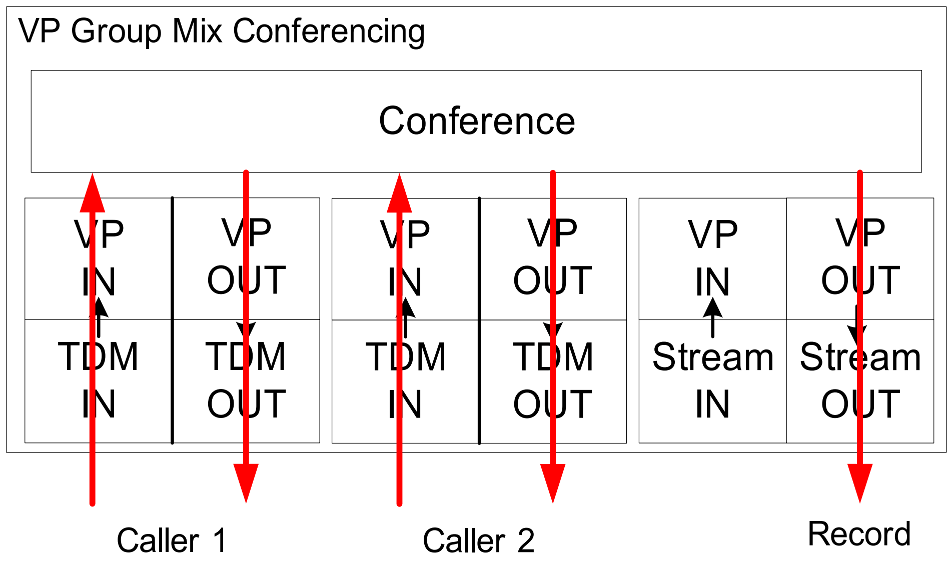


Figure 41: Recording a conference

6.2.2.3.4 Background Music application

In this application, we want background music to be played while callers are talking together. To do this, you assign on TDM member per caller and you must assign one more stream member in the group for playing the music.

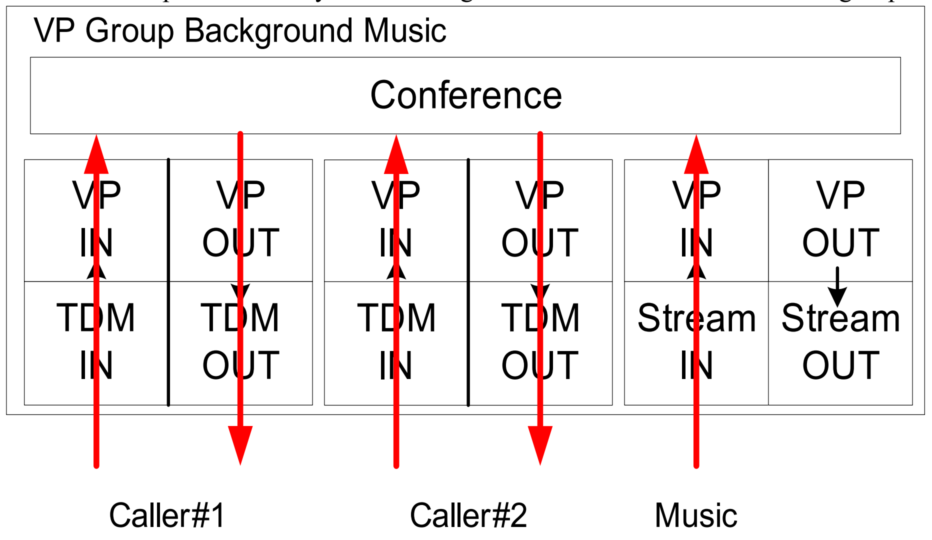


Figure 42: Background Music Application

6.2.2.4 Voice processing group1 (VoIp) capabilities

Voice processing group type 1 is implemented using a VoIp mezzanine and is used when the application requires one or more voice services listed below:

. TDM to packet conversion using different types of codecs (G.711, G.723.1, G.726, G.728,G.729AB, G.729EG, AMR, EVRC) and associated functions (VAD, jitter buffers, comfort noise generation, etc).
. Echo cancellation up to 128 msec of echo tail [inline within a 'TDM to packet' or pure TDM]
. Fax relay over VoIp network (RFC2833)

Since each voice processing function may have an influence on the voice according to its location within the processing block, overview diagrams are provided below of all supported types of voice processing resources for group type 1 (VoIp). More details about their configuration parameters are provided within the voice processing API document.

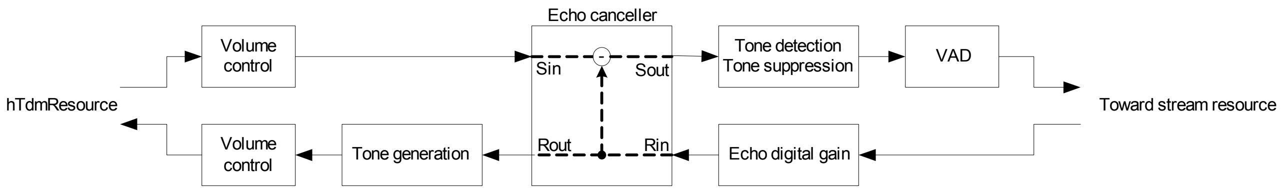


Figure 43: VP group1 TDM resource schematic (when used in conjunction with a stream resource)

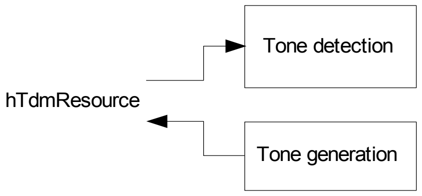


Figure 44: VP group1 TDM resource schematic (when used standalone)

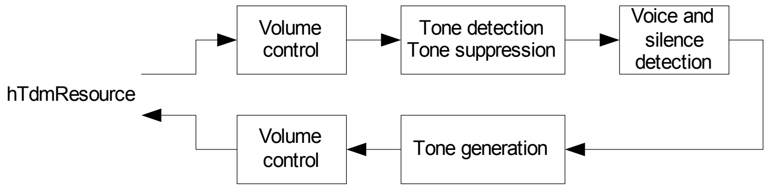


Figure 45: VP group1 TDM flowthru resource schematic

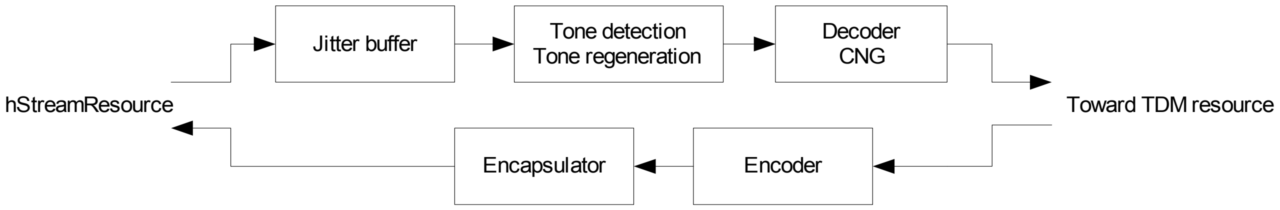


Figure 46: VP group1 stream resource schematic

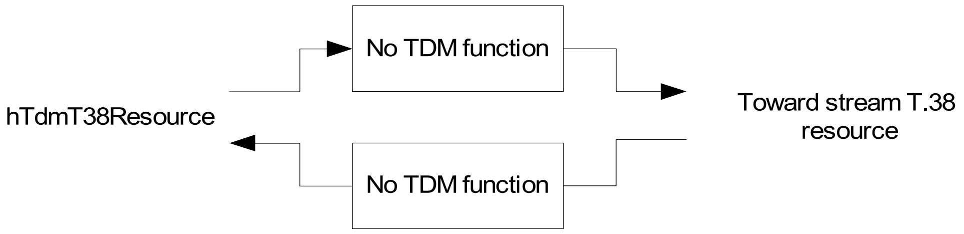


Figure 47: VP group1 TDM T.38 resource schematic

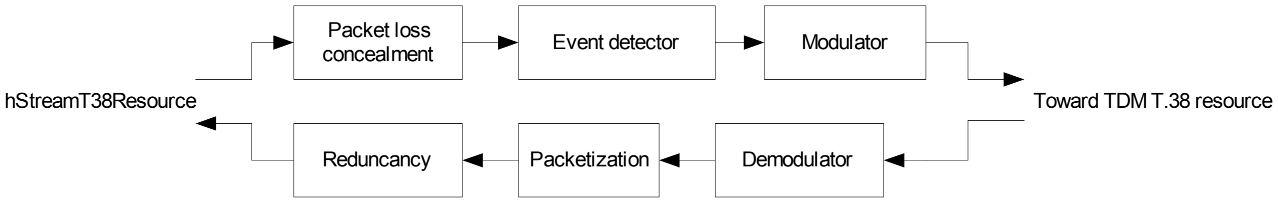


Figure 48: VP group1 stream T.38 schematic

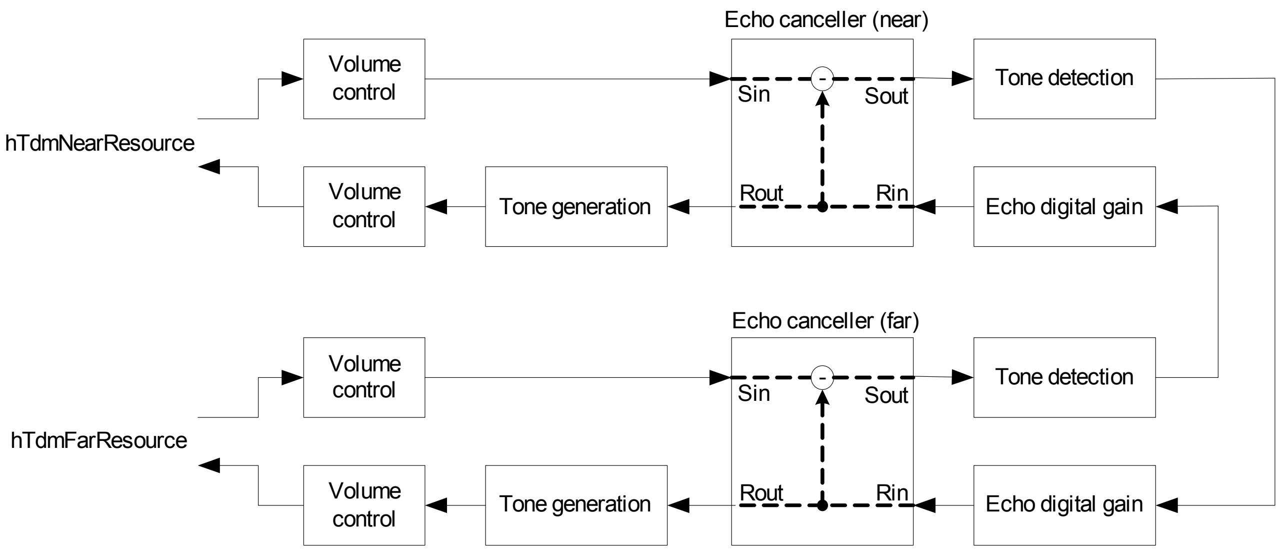


Figure 49: VP group1 TDM echo schematic

6.2.2.4.1 VP group1 resource usage examples

6.2.2.4.1.1 One VP TDM resource (flowthru) in a VP group

A TDM stream comes-in, is processed, then comes out on the same VP resource. This is usually a simplex connection that needs to have the DTMF tones monitored or to be able to generate a tone at any time.

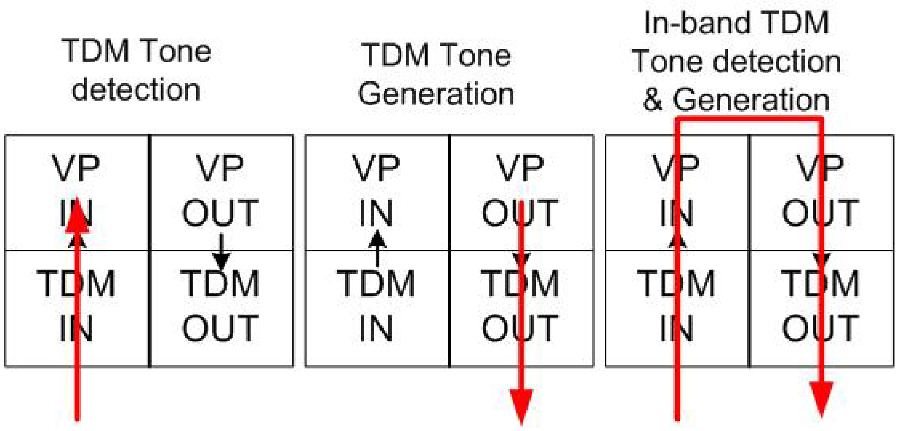


Figure 50: VP group using VP TDM resource (flowthru)

6.2.2.4.1.2 One VP TDM resource (pure-TDM) in a VP group

A TDM stream comes-in and is processed (i.e. tone detection). The resource can also be used to generate stream out to the TDM (i.e. tone generation). But the traffic from the TDM is <u>not</u> looped back onto the TDM output. A typical example of such resource is to detect call progress tones which only required the voice stream from the TDM (i.e. no need to put the resource inline on the TDM path. The only need to is 'fork' the TDM path to such resource).

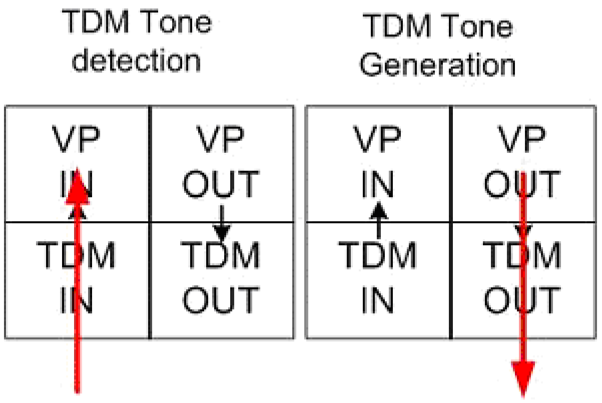


Figure 51: VP group using VP TDM resource (pure-TDM)

6.2.2.4.1.3 One VP TDM and one Stream resource in a VP group

A Stream input comes out as a TDM stream while the TDM stream input comes out as a Stream. This is to create a TDM to Stream transfer for a VoIp voice conversation, recording and/or playing streams, and other applications.

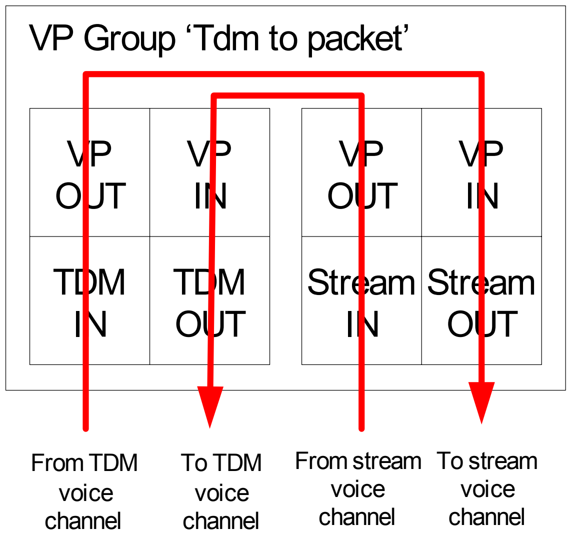


Figure 52: VP group using VP TDM and stream resources

6.2.2.4.1.4 One VP TDM T.38 and one Stream T.38 resource in a VP group

A Stream input (data) comes out as a TDM T.38 stream (modulated) while the TDM stream input (modulated) comes out as a Stream (data). This is to create a TDM to Stream transfer for a fax transmission.

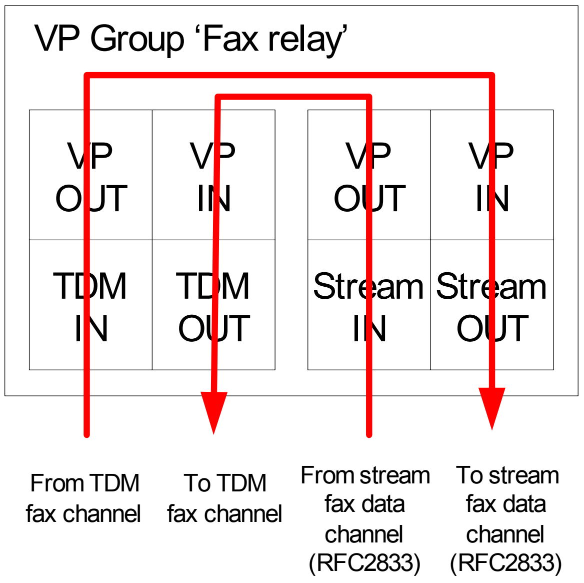


Figure 53: VP group using VP TDM T.38 and stream T.38 resources

6.2.2.4.1.5 Two VP TDM echo resources in a VP group

A first TDM input comes out as a second TDM stream while the second TDM stream input comes out as the first TDM stream. This is to create a full-duplex TDM echo canceller that can connect to any other TDM resource of the blade.

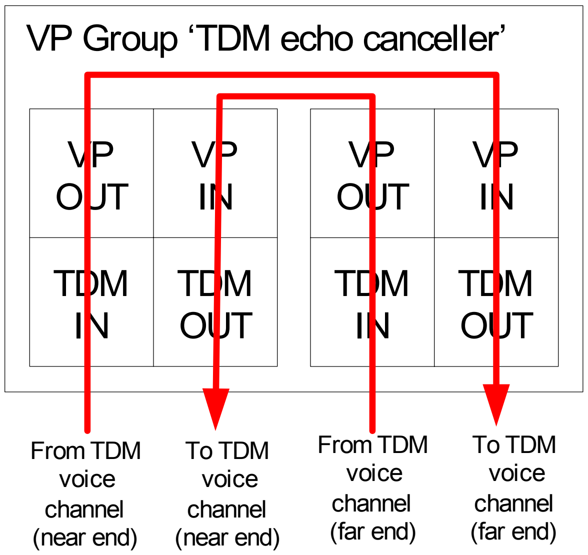

VP Group 'TDM echo canceller'
VP OUT
VP IN
VP OUT
VP IN
TDM IN
TDM OUT
TDM IN
TDM OUT
From TDM voice channel (near end)
To TDM voice channel (near end)
From TDM voice channel (far end)
To TDM voice channel (far end)

Figure 54: VP group using two VP TDM echo resources

6.2.2.5 VP group1 functions

6.2.2.5.1 Tone detection, suppression, generation and relay

Depending on the schematic of VP resource used (refer to section 6.2.2.4), tone detection/suppression/generation/relay can be configured to process DTMF tones. When a tone is detected, the TB640_MSG_ID_VP_TONE_NOTIF_DETECTION event will be generated to the host application. If the 'tone validation' option is configured, the TB640_MSG_ID_VP_TONE_NOTIF_CADENCE_DETECTION event will be sent once the tone has respected proper timing constraints (which are configurable when the VP resource is allocated). When the tone stops, the TB640_MSG_ID_VP_TONE_NOTIF_STOP event will be generated. If suppression is enabled, the tone will be removed from the voice path and silence will be inserted. You also get a timestamp in millisecond for each of those events. This can be compared to any previous event received with this timestamp.

When the 'tone relay' option is configured in the 'TDM toward stream' direction, the tone will be automatically converted to a data packet and sent to the network. Note that the user still has to use the 'tone suppression' option he does not want the in-band tone to be encoded in the voice stream as well. When the 'tone regeneration' option is configured in the 'stream to TDM' direction, a tone data packet will be automatically converted to a TDM tone.

The message TB640_MSG_ID_VP_TONE_PLAY is used to generate in-band tones onto VP resources. Both TDM and stream resource can generate such tones. The user needs to specify the frequencies (in hertz), amplitude (in dbm) and duration of the tone. If the duration is infinite (value of -1), the TB640_MSG_ID_VP_TONE_STOP must be used to stop the tone from being played. To generate a cadenced tone (busy, ringback, reorder, etc.), in addition to the frequency and amplitude, the on and off time must be specified and the TB640_ MSG_ID_VP_TONE_STOP message must be used to stop the tone. Segment tones can also be generated by a specific sequence of frequencies with their on and off times. Refer to the voice processing API document for more details.

6.2.2.5.2 Call progress tone detection

6.2.2.5.2.1 Call progress basics

Call progress tones are special tones sent by the network to inform the listener about an ongoing call status. VP Group1 resource are able to detect some of those tones namely ringback, busy, dial and number unobtainable (also referred to as 'NUT'). The dial tone is used to indicate to the listener that the line is ready for signaling (i.e. to enter his 'called' number). The ringback tone is used to indicate that the remote side has been reached and that his phone is ringing. This tone is usually sent by the local switch (not the remote-end switch). Instead of the ringback tone, the network can send a busy or NUT tone to indicate respectively that the remote-end is already on another call or that the remote end is unreachable.

Call progress tone characteristics varies from one country to the other in term of frequency used, cadence and signal power. This is why the user needs to configure the VP group1 channel about the call progress tone it should be expecting. For example, listed below are the characteristics of North America, China and Korea call progress tones:

Table 5 – North America/USA call progress tone definitions

Tone ID	Frequency #1	Frequency #2	Power	Cadence
Dial tone	350Hz	440Hz	-13 dBmo	Continuous
Busy tone	480Hz	620Hz	-24 dBmo	0.5s ON, 0.5s OFF
Ringback tone	440Hz	480Hz	-19 dBmo	2s ON, 4s OFF
NUT	480Hz	620Hz	-24 dBmo	0.25s ON, 0.25s OFF

Table 6 – China call progress tone definitions

Tone ID	Frequency #1	Frequency #2	Power	Cadence
Dial tone	450Hz	None	-10 dBmo	Continuous
Busy tone	450Hz	None	-10 dBmo	0.35s ON, 0.35s OFF
Ringback tone	450Hz	None	-10 dBmo	1s ON, 4s OFF
NUT	450Hz	None	-9 dBmo	75ms ON, 100ms OFF 75ms ON, 100ms OFF 75ms ON, 100ms OFF 75ms ON, 400ms OFF

Table 7 – Korea call progress tone definitions

Tone ID	Frequency #1	Frequency #2	Power	Cadence
Dial tone	350Hz	440Hz	-10 dBmo	Continuous
Busy tone	480Hz	620Hz	-20 dBmo	0.5s ON, 0.5s OFF
Ringback tone	440Hz	480Hz	-15 dBmo	1s ON, 2s OFF
NUT	None	None	None	Not used, IVR prompt are usually sent. Fast-busy signal is also used
Fast-busy*	480Hz	620Hz	-20 dBmo	0.3s ON, 0.2s OFF

* Fast-busy is not one of the 4 tones that the VoIp Group1 can detect using the pre-defined countries but it can be configured to replace one of those four tones using the custom country parameters (explained below).

VP Group1 resource has predefined set of parameters already configured such as the user only needs to specify the country code upon allocation. It is also possible to specify different country codes for the different tones within the same resource. This is to allow international calls where the ringback tone is generated by the local switch and the other tones are generated by the foreign switch. For example, a USA to China call would require the ringback tone to be configured as 'USA' while the other tones would be configured as 'China'. Upon detection, the TB640 will send a notification (TB640_MSG_ID_VP_TONE_NOTIF_CADENCE_DETECTION) toward the application with the identifier of the VP resource generating the event and the tone that was detected (e.g. TB640_VP_DIGIT_TYPE_CALL_PROGRESS_DIAL_TONE).

6.2.2.5.2.2 Custom call progress tones

Under some circumstance, an application may require detecting more tones than those 4 pre-defined tones or need to replace the characteristics of basic call progress tones. The user can create up to 8 custom country call progress tone characteristics in the non-volatile memory using message TB640_MSG_ID_VP_GROUP_SET_NVPARAMS (see API reference guide for a more detailed description). Within the Group1 NvParam structure, the user needs to configure the bi-quad filter parameters and cadence characteristics of all 4 tones. Adjusting the filter parameters is not a trivial task and may require the help from the Telcobridges' support team. Unspecified behavior can occur if wrong or contradictory parameters are entered.

☞ Although the four types of tone are 'named', nothing prevents the user to replace the NUT tone characteristics with a 'fast-busy' for a specific application. The only thing to remember is that the notification received upon detection will be a tone of type TB640_VP_DIGIT_TYPE_CALL_PROGRESS_NUMBER_UNAVAIL_TONE.

6.2.2.5.3 Echo cancellation

Two types of echo exist in communication systems: electrical and acoustical echo. The former is created by an impedance mismatch of 4-wire phone circuits to 2-wire phone circuits. The reflection created by this mismatch is an attenuated version of the received signal and is sent back onto the wire. The later is created when the microphone of a phone or handset is capturing a portion of the output from the speaker (i.e. cellular phone, videoconference systems, etc). The actual feedback (echo) becomes audible when the round trip delay of the voice path begins to exceed 15msec. Echo cancellation is required in most VoIp applications due to the packetization delay of converting from the TDM domain to the packet domain; not that this delay creates the echo but it multiplies the effect of echo currently present on the TDM-side making it more audible (thus more disturbing) for full-duplex voice conversations. Echo can also create situations of false tone detection since a user could 'hear' the echo of a tone he generated and, thus, detect a tone

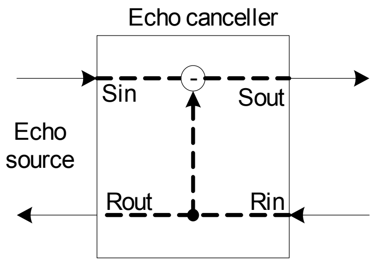


Figure 55: Unidirectional echo canceller

In order to cancel the echo, the echo canceller must first try to locate the echo within the signal. Being inserted serially in the full-duplex connection, it does so by comparing the history of the signal sent toward the echo source and the resulting signal received from the echo source. Once the adaptation is done and that the canceller has locked onto the echo, it can subtract the injected echo from the signal sent by the echo source. The faster the echo canceller can lock onto the echo pattern, the faster the echo part will be removed from the voice stream and the conversation quality will improve (i.e. low convergence delay). On the other hand, the echo canceller still needs to be careful not to remove any valid component. Some situations such as double-talk (i.e. when both persons speak at the same time) require special processing to get a good quality conversation. The echo canceller may also have to adapt to the echo as it is possible for the echo to vary in terms of delay and length. This will require the canceller to re-locate the echo position within the stream until it has a new lock. More complex voice setup such as three-way conferencing may create multiple echo paths as the delay between each participant may differ. This has an impact on the canceller location function as it will receive different feedback and delays from the different participant. Furthermore, when the echo canceller detects that the echo source does not feed any voice (i.e. silence period), it can decide to inject comfort noise in the other direction instead of try to cancel the echo. This last function is done by an NLP (non-linear processor).

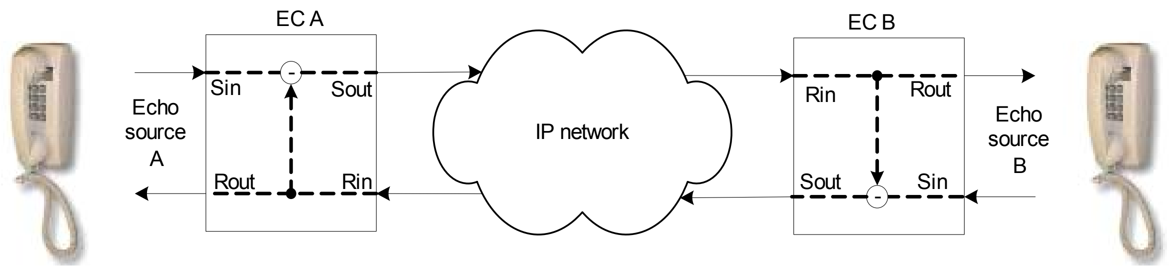


Figure 56: Bidirectional echo cancellation

In a network, an endpoint is responsible to cancel the echo generated on its TDM-side. Figure 56 shows two echo cancellers located each side of an IP network. Echo canceller A is responsible to cancel the echo from the echo source A. It won't cancel the echo generated by the source B. Therefore, both echo canceller are unidirectional and are inserted in-line of the conversation. In a voice processing group where a TDM resource and a stream resource are present, an in-line echo canceller is also present (if enabled) to cancel the echo injected on the TDM side of the connection (refer to schematic in Figure 43). It therefore assumes the remote VoIp end also cancels the echo on its side.

In the case where no VoIp stream is involved (no packet network), it is still possible to use the echo canceller of the VoIp mezzanine by creating a voice processing TDM echo group containing two echo resources (i.e. near and far) as shown by Figure 49. This will create a voice processing entity capable of being either unidirectional or bidirectional echo canceller entity. The application is required to connect both TDM full-duplex resources of the entity to each side of the voice conversation.

☞ Do not confuse "unidirectional" or "bidirectional" attributes with the attributes "half-duplex" and "full-duplex". An echo canceller is **ALWAYS** full-duplex as it required both input and output stream data to do its job. But it can be unidirectional or bidirectional depending if it is processing only local and/or remote echo on a stream.

For more information about configuration parameters of echo canceller resource within a voice processing group, please refer to the Voice processing API document.

☞ Echo tail length is usually a characteristic limiting the total number of echo canceller resources in a VoIp equipment. With the current implementation, each voice processing group TDM<->stream has an inline echo canceller with 128 msec echo tail length without any penalty on the total number of resource available.

☞ Echo canceller may affect the content of the voice stream. Thus, it is highly recommended to turn it off when the connection is transporting data such as BERT, fax or modem signal. A 2100Hz tone with phase reversal is usually generated in-band to signal every echo canceller within a voice path to disable. Some tones may also indicate early stages of a modem/fax connection which would require disabling the echo canceller.

6.2.2.5.4 Codecs

As in the TDM network architecture, the service providers are always trying to optimize the bandwidth usage of their transport infrastructures. Numerous standards exist for compression algorithms in order to achieve an acceptable voice quality while trying to reduce bandwidth usage. Depending in which environment a VoIp product is deployed, it may need to interoperate with different codecs (coder/decoder) in order to transport voice streams. Other parameters such as the packet duration have an influence on the voice quality and bandwidth usage. Indeed, using a large packet duration (i.e. 160msec) means that the VoIp device will gather more TDM samples into a single packet before sending it out of the IP network thus reducing the overhead of the Ethernet/IP/UDP/RTP header. Unfortunately, this will also create a latency of the same duration amount since the receiver will hear the voice delayed by the same amount as the packetization time.

Therefore, a VoIp equipment needs to support a wide variety of codecs and packet duration in order to be compatible with almost every type of network or applications. All supported codecs by the voice processing group 1 are listed below. This list does not contain the amount of codecs available for the VoIp product as it depends on the hardware configuration and on runtime parameters (such as packet duration and other initialization parameters). This information is detailed in the "TBVoipWizard.html" document included in the release package's documentation directory.

Table 8: VoIp supported codec list

Codec type	Packet duration supported (msec)	Complex	Wireline	Wireless	CDMA
Clear channel	5, 10, 20, 30	No	√		
G.711	5, 10, 20, 30	No	√		
Pass-thru	5, 10, 20, 30	No	√		
G.726	10, 20, 30	No	√		
G.723.1	30, 60	Yes	√		
G.728	5, 10, 20, 30	Yes	√		
G.729AB	10, 20, 30	Yes	√		
G.729EG	10, 20, 30	Yes	√		
AMR	20, 30	Yes		√	
EVRC	20, 30	Yes		√	√
QCELP-8K	20, 30	Yes		√	√
QCELP-13K	20, 30	Yes		√	√
iLBC-13K	30,60	Yes	√		
iLBC-15K	20,40	Yes	√		
T.38 Fax	---	Yes	√		

The document "TBVoipWizard.html" will actually give the number of channels an application can allocate depending on which codecs are used at the same time. These density values are predictable and reproducible meaning that neither the history of channel opening/closing nor the order in which the codecs are used has any effect on the overall amount of available channel. But in order to use the full potential of the VoIp devices, the on-board software needs to know the type of application (i.e. codec usage) in order to distribute the load properly on the different devices.

It is always easier to tell which codec types will be used in a specific application (usually dictated by service providers) but it is not that easy to predict the actual number of channels that will use each codec types. Therefore, many allocation algorithms are available that will make the blade to distribute the load differently on the VoIp devices and lead to more optimized performances (i.e. channel density).

☞ The 'clear-channel' codec will pass IP data directly to the PCM stream thus bypassing all other subsystems. For example, echo canceller will not work when using this codec.

☞ The 'pass-thru' codec is equivalent to G.711. It is used for systems requiring different payload type values to differentiate G.711 audio from G.711 fax/modem pass-through.

6.2.2.5.4.1 Allocation algorithm 1 (mixed codecs)

This algorithm assigns the different VP Group1 resources to each device in a first-come, first-served fashion. This algorithm may mix any codec types on the same device. This algorithm is better suited for customer's application (or environment) where it is impossible to predict how many instances of a specific codec will be used. It will ensure a deterministic allocation of channels. The application designer can then use the values from this wizard to know how many VoIp channels will be available.

> ☞ This algorithm can be understand as an **OR** type algorithm in the sense that the TBVoipWizard.html gives the maximum of each codec if all other aren't being used. For Example, you get 343 G711_20 per Voip device, **OR** 128 T38, not both at the same time.

This algorithm is better suited for:

1. Application (or environment) where it is impossible to predict how many instances of a specific codec will be used.
2. Application (or environment) that needs all channels in a specific codec, and at a later point in time, needs all channels in another codec.
3. Application (or environment) making **MOSTLY** uses of the following codecs:
   - G711_5
   - G723_1
   - AMR
   - EVRC
   - QCELP8
   - QCELP13
   - SMV
   - EFR
   - T38

   The above list of codecs can be used without any lost of density efficiency when mixed with any other codecs **NOT** in this list.
4. Application (or environment) making **ONLY** uses of the following codecs:
   - G711_20
   - G711_10
   - G711_5
   - G726_20
   - G726_10

   The above list of codecs can be used without any lost of density efficiency when mixed with other codecs **ONLY** from this list.

6.2.2.5.4.1.1 Coding your application using algorithm 1 (mixed mode):

The algorithm 1 allows for more flexibility in terms of codec availability. This flexibility imposes the software designer to use a certain algorithm in order know channel availability. The problem is that a voip device supports up to 343 channels of G711_20 **or** up to 128 T38 channels **or** up to 66 channels of G728. Allocating one T38 channel leaves 343 – (343/128) = 340 channels of G711_20 free for allocation. So one could think the cost of a T38 channel is more or less 3 G711_20. But the cost is also 66/128 = 0.5 G728 channels. The cost approach is not appropriate since we are not only dealing with 3 codecs, but over 15.

The best approach on this problem is to give each codec a weight. To do so, we choose the codec with the highest density, which is G711_20 at 343 channels per Voip device. All others codecs are weighted against G711_20. In the following example, all weights are also multiplied by 1000 to prevent usage of floating points.

MAX_CHANS_PER_VOIP	343	(max number of G711_20 channels per devices)	
MAX_WEIGHT_PER_VOIP	343000	(MAX_CHANS_PER_VOIP*1000)	
Codec	Max per voip device*	Calcul	Weight
G711_20	343	MAX_WEIGHT_PER_VOIP/343	1000
G711_10	217	MAX_WEIGHT_PER_VOIP/217	1581
G711_5	128	MAX_WEIGHT_PER_VOIP/128	2680
G726_20	217	MAX_WEIGHT_PER_VOIP/217	1581
G726_10	170	MAX_WEIGHT_PER_VOIP/170	2018
G729AB	154	MAX_WEIGHT_PER_VOIP/154	2227
G723_1	128	MAX_WEIGHT_PER_VOIP/128	2680
G728	66	MAX_WEIGHT_PER_VOIP/66	5197
G729EG	71	MAX_WEIGHT_PER_VOIP/71	4831
EFR	120	MAX_WEIGHT_PER_VOIP/120	2858
AMR	96	MAX_WEIGHT_PER_VOIP/96	3573
EVRC	96	MAX_WEIGHT_PER_VOIP/96	3573
QCELP8	96	MAX_WEIGHT_PER_VOIP/96	3573
QCELP13	72	MAX_WEIGHT_PER_VOIP/72	4764
SMV	84	MAX_WEIGHT_PER_VOIP/84	4083
T38	128	MAX_WEIGHT_PER_VOIP/128	2680
* (Values taken directly from TBVoipWizard.html, and are subject to changes)			

Example, let’s say we have only one Voip device:

We want to allocate 30 G729E, and 10 G728, how much space is left?
MAX_WEIGHT_PER_VOIP – (30 * G729EG_WEIGHT) – (10 * G728_WEIGHT) =
343000 – (30 * 4831) – (10 * 5197) = **146 100**
The left weight in the Voip device is 146100. We have enough space to allocate either 146 G711_20, or 54 T38, etc…

The software designer must use a total weight representing the MAX_WEIGHT_PER_VOIP multiplied the number of Voip devices present on the TB640. Upon each channel allocation and channel free, the total weight must be updated according to the respective weight of each codec. That way, the application will be able to know how much of each codec is available for allocation at a specific time by knowing what codecs are currently being allocated.

6.2.2.5.4.2 Allocation algorithm 2 (fixed codecs)

This algorithm assigns the different VP Group1 resources in order to maximize the mixing of simple (G.711 and G.726) and complex codecs (all other codecs). This will end-up in having better channels density for application For example, application switching between G.711, G.726 and G.729AB will benefit from this algorithm.

☞ This algorithm can be understand as an **AND** type algorithm in the sense that the TBVoipWizard.html gives the maximum of each codec available at all time. For Example, you get 202 G711_20 per Voip device, **AND** 52 T38, both at the same time.

This algorithm is better suited for:

1. Application (or environment) where it is important to know exactly how much of each codec there is. This algorithm RESERVES the requested amount of channels of each codecs. This algorithm is more deterministic in a sense that codecs are statically reserved in advance.
2. Application (or environment) that needs Great Density of the following codecs list, mixed with any other codecs:
   - G711_20
   - G711_10
   - G726_20
   - G726_10
   - G729AB
   - G728
   - G729EG

6.2.2.5.4.2.1 Coding your application using algorithm 2 (fixed mode):

The algorithm 2 has a more deterministic approach in the sense that all codec are being reserved in advance. Once properly configure, the numbers from the TBVoipWizard.html are exactly what the application can get at all time, no matter what has been previously allocated. Thus, all the application needs to do is to remember how many of each codecs has been allocated and keep these counts under or equal to the number of codec reserved.

6.2.2.5.5 Payload type values

The RTP protocol used to transport voice across an IP network is described by RFC3550 (previous version was RFC1889). Within the RTP header, one field named 'PT' (payload type) is used to identify the type of codec information carried in the packet. A list of payload type values for each codec is defined within RFC3551. Unfortunately, since the payload type field is only 7 bits-wide, all codecs cannot have a permanent payload type value understood universally by all VoIp systems. Therefore, some codecs have dynamic values that need to be negotiated through a call control or session control protocol such as SIP or H.323 before the actual RTP session can take place.

In order for the voice processing group1 resource to understand the codecs for a particular network, the application must configure the payload type values that will be recognize by this resource and assign it to a specific codec (refer to the Voice processing API document in the stream resource section for more details on configuration fields). Assigning the same payload value to two different codecs is not recommended and behavior may be erratic as the VoIp device won't be able to know which codec to use for decoding a particular packet. For every channel, the application only needs to configure the payload type values for what needs to be processes (see below for typical types to configure). The voice processing resource will always send a notification toward the host application when a payload type different from the currently configured payload types will be received (i.e. TB640_MSG_ID_VP_VOIP_NOTIF_PAYLOAD_TYPE_CHANGE).

Table 9 - Codec payload type values per RFC3551

Codec	Payload type value
G.711 uLaw	0
G.723.1	4
G.711 aLaw	8
G.728	15
G.729AB	18
G.726-40	dynamic
G.726-32	2 or dynamic (depends on the network)
G.726-24	dynamic
G.726-16	dynamic
G.729EG	dynamic
AMR	dynamic
EVRC	dynamic
QCELP	dynamic

☞ In a real VoIp public network, if the local end and remote end are not configured with the same payload type values, the VoIp device will probably drop all incoming packets due to the invalid/unknown payload type in the received packets. Theses errors are logged into the channel statistics. Also, when too many of those errors are seen by the device, a notification (TB640_MSG_ID_VP_NOTIF_STATUS_INDICATION with status TB640_VP_GROUP_STATUS_INDICATION_EXCESSIVE_PAYLOAD_TYPE_CHANGE) is sent to the host application informing it about the possible misconfiguration between the VoIp endpoint. At that point, it is up to the application to either fix the situation (e.g. change the payload type values of the VP Group1 resource) or close the resource (and possibly cancel the associated call).

Typically, only a few payload type values need to be set in order to have a fully functional Voip RTP session. The only mandatory payload type to set is the value for the codec to use for the Voip resource. For example, if the resource is set to use the G.728 codec, the application needs to tell the TB640 the payload type value to use when transmitting and receiving RTP packets of G.728 audio. Optionally, other payload type values can be set to get supplementary services out of the voice stream. Typically, the CN (comfort noise) payload type value needs to be programmed as well if VAD (voice activity detection) is to be used. Also, the RFC2833 payload type value needs to be programmed in order to transmit out-of-band DTMF and telephone events on the packet network.

Table 10 shows the packet types that have fixed payload type values. These values are defined in RFC3551.

Table 10 – Payload type values assigned statically

Packet type enum	Description
TBX_MEDIA_TYPE_AUDIO_PCMU	This packet type represents the G.711 ulaw codec. The payload type value for this codec is fixed per RFC3551 [Table4] to the value 0x00.
TBX_MEDIA_TYPE_AUDIO _PCMA	This packet type represents the G.711 alaw codec. The payload type value for this codec is fixed per RFC3551 [Table4] to the value 0x08.
TBX_MEDIA_TYPE_AUDIO _G723_1	This packet type represents the G.723 codec. The payload type value for this codec is fixed per RFC3551 [Table4] to the value 0x04.
TBX_MEDIA_TYPE_AUDIO _G728	This packet type represents the G.728 codec. The payload type value for this codec is fixed per RFC3551 [Table4] to the value 0x0F.
TBX_MEDIA_TYPE_AUDIO _CN	This packet type represents a silence frame (or comfort noise frame) within the RTP flow. The payload type value for this codec is fixed per RFC3551 [Table4] to the value 0x0D.
TBX_MEDIA_TYPE_AUDIO _G729AB	This packet type represents the G.729 codec. The payload type value for this codec is fixed per RFC3551 [Table4] to the value 0x12.

☞ The enum TBX_STREAM_PACKET_TYPE is equivalent to the enum TBX_MEDIA_TYPE. For historical reason, the VP API uses the former enum. Any new application should be written using the later enum.

Table 11 shows the packet types that have dynamic payload type values. The payload type value associated with it is defined in the dynamic range of RFC3551. It means that the actual value will be negotiated using an out-of-band protocol such as SIP or H.323.

Table 11 – Payload type values dynamically negotiated

Packet type enum	Description
TBX_MEDIA_TYPE_AUDIO_G726	This packet type represents the G.726 16/24/32/40kbps codec.
TBX_MEDIA_TYPE_AUDIO_RED	This packet type represents the RTP frame used for redundancy payload according to **RFC2198**. This type of frame can carry any other standard RTP packet including telephony event packets (defined below), voice packets or others. This payload type is used when tones are configured for NOT using AAL2 redundancy. Refer to structure TB640_VP_GROUP1_TONE_PARAMS.
TBX_MEDIA_TYPE_AUDIO_TELEPHONY_EVENT	This packet type represents the telephony event frame within the RTP flow. It is primarily used when sending out-of-band DTMF packets. These frames use the **RFC2833** packet format.
TBX_MEDIA_TYPE_IMAGE_T38	This packet type represents the fax relay UDPTL packet format (T.38)
TBX_MEDIA_TYPE_AUDIO_G729E	This packet type represents the G.729 EG codec.
TBX_MEDIA_TYPE_AUDIO_AMR	This packet type represents the AMR codec.
TBX_MEDIA_TYPE_AUDIO_EFR	This packet type represents the EFR codec.
TBX_MEDIA_TYPE_AUDIO_CLEAR_CHANNEL	This packet type represents an RTP encapsulation where no specific codec is used.
TBX_MEDIA_TYPE_AUDIO_SMV	This packet type represents the SMV codec.
TBX_MEDIA_TYPE_AUDIO_EVRC	This packet type represents the EVRC codec.
TBX_MEDIA_TYPE_AUDIO_QCELP8	This packet type represents the QCELP-8k codec.
TBX_MEDIA_TYPE_AUDIO_QCELP13	This packet type represents the QCELP-13k codec.
TBX_MEDIA_TYPE_AUDIO_ILBC	This packet type represents the iLBCP-13k/15k codec.

6.2.2.5.6 Mapping SIP SDP to VP Group1 resource parameters and payload types

On a SIP Voip network, the payload type values for codecs and other events are negotiated per call within the SIP protocol. The tricky part is then to decipher the SIP SDP information in order to format the proper parameters to set in the voice processing resource. The codec and payload type values are negotiated with the SDP information but the packet duration is not a parameter that is always part of the SDP. Typically, a well known default packet duration is defined for almost all codec. This default can be optionally overridden by an attribute (called *ptime*) in the SDP information. However, TelcoBridges has found many implementations of SIP devices that do not make use of this parameter. Even if a VP Group1 resource can accept packets with different durations (e.g. a PCMA 60msec resource will properly process 5msec received packets), the host application should always use the packet duration closest to the the typical packet duration for a specific Voip network. This will avoid overloading the Voip devices with unexpected rate of packets. This means that if the network is known to use PCMA 5msec, then all resources should be configured to process 5msec packets.

RFC2327:

```
Note that RTP audio formats typically do not include information
about the number of samples per packet.  If a non-default (as
defined in the RTP Audio/Video Profile) packetization is required,
the "ptime" attribute is used as given below…
```

RFC1890:

```
For packetized audio, the default packetization interval should
have a duration of 20 ms, unless otherwise noted when describing
the encoding.
```

encoding	sample/frame	bits/sample	ms/frame
1016	frame	N/A	30
DVI4	sample	4	
G721	sample	4	
G722	sample	8	
G728	frame	N/A	2.5
GSM	frame	N/A	20
L8	sample	8	
L16	sample	16	
LPC	frame	N/A	20
MPA	frame	N/A	
PCMA	sample	8	
PCMU	sample	8	

Listed below are the mostly used SDP format an application would encounter and their corresponding packet type configuration:

1. SIP INVITE containing PCMA, telephony events and VAD

```
m=audio 10332 RTP/AVP 8 101 13          ← Using UDP port 10332
a=rtpmap: 8 PCMA/8000                   ← G.711 alaw using payload type 8
a=rtpmap: 101 telephone-event/8000      ← RFC2833 for telephony using payload type 101
a=rtpmap: 13 CN/8000                    ← VAD comfort noise packets using payload type 13
```

```
PTB640_VP_GROUP1_STREAM_PARAMS          pStreamParams;
PTB640_VP_GROUP1_TDM_PARAMS             pTdmParams;

pTdmParams =  &(pVpGroup->aRes[0].Params.Group1.Tdm);
pStreamParams  = &(pVpGroup->aRes[1].Params.Group1.Stream);

/* Set the default values */
TB640_VP_GROUP1_TDM_PARAMS_DEFAULT (pTdmParams);
TB640_VP_GROUP1_STREAM_PARAMS_DEFAULT (pStreamParams);

/* Choose the codec */
pStreamParams->Codec.MediaType = TBX_MEDIA_TYPE_AUDIO_PCMA;
pStreamParams->PacketDurationMs = TBX_STREAM_PACKET_DURATION_20MS;
```

```
/* Clear the array of payload type values */
for (un32Count=0; un32Count<TB640_VP_GROUP1_MAX_PAYLOAD_TYPES; un32Count++)
{
        pStreamParams->aPacketProtocol[un32Count].PacketType = TBX_MEDIA_TYPE_INVALID;
        pStreamParams->aPacketProtocol[un32Count].un8TxProtocolType = 0xFF;
        pStreamParams->aPacketProtocol[un32Count].un8RxProtocolType =  0xFF;
}

/* Used for RFC2833 */
pStreamParams->aPacketProtocol[0].PacketType = TBX_MEDIA_TYPE _AUDIO_TELEPHONY_EVENT;
pStreamParams->aPacketProtocol[0].un8TxProtocolType = 101;
pStreamParams->aPacketProtocol[0].un8RxProtocolType = 101;

/* Used for RFC3389 */
pStreamParams->aPacketProtocol[1].PacketType = TBX_MEDIA_TYPE _AUDIO_CN;
pStreamParams->aPacketProtocol[1].un8TxProtocolType = 13;
pStreamParams->aPacketProtocol[1].un8RxProtocolType = 13;

/* Used for main codec */
pStreamParams->aPacketProtocol[2].PacketType = TBX_MEDIA_TYPE _AUDIO_PCMA;
pStreamParams->aPacketProtocol[2].un8TxProtocolType = 8;
pStreamParams->aPacketProtocol[2].un8RxProtocolType = 8;

/* Configure VAD */
pTdmParams ->Vad.fEnabled = TBX_TRUE;
pTdmParams ->Vad.SidGenerationScheme= TB640_VP_SID_GENERATION_SCHEME_DEFAULT;
pTdmParams ->Vad.CngGenerationScheme= TB640_VP_CNG_GENERATION_SCHEME_PT13;
pTdmParams ->Vad.NoiseFloor = TB640_VP_VAD_NOISE_FLOOR_LEVEL_MINUS_20_DBM;

/* Activate tone relay */
pTdmParams ->Tone.fDtmfSuppressionEnabled = TBX_TRUE;
pTdmParams ->Tone.fDtmfCompleteSuppression= TBX_TRUE;
pTdmParams ->Tone.fDtmfToneRegeneration= TBX_TRUE;
pStreamParams ->fEnableRfc2833ToneRelay = TBX_TRUE;

/* Use DTMF AAL2 redundancy (RFC2833 only)
pTdmParams ->Tone.fDtmfToneAal2Redundancy = TBX_TRUE;
```

2. SIP INVITE containing G.723, telephony events

```
m=audio 9332 RTP/AVP 4 101              ← Using UDP port 9332
a=rtpmap: 4 G723/8000                   ← G.723 using payload type 4
a=rtpmap: 101 telephone-event/8000      ← RFC2833 for telephony using payload type 101
```

```
PTB640_VP_GROUP1_STREAM_PARAMS          pStreamParams;
PTB640_VP_GROUP1_TDM_PARAMS             pTdmParams;

pTdmParams =  &(pVpGroup->aRes[0].Params.Group1.Tdm);
pStreamParams  = &(pVpGroup->aRes[1].Params.Group1.Stream);

/* Set the default values */
TB640_VP_GROUP1_TDM_PARAMS_DEFAULT (pTdmParams);
TB640_VP_GROUP1_STREAM_PARAMS_DEFAULT (pStreamParams);

/* Choose the codec */
pStreamParams->Codec.MediaType = TBX_MEDIA_TYPE_AUDIO_G723_1;
pStreamParams->Codec.G723.fUseHighRateCodec = TBX_TRUE;
pStreamParams->Codec.G723.fEnableDcRemoval= TBX_TRUE;
pStreamParams->PacketDurationMs = TBX_STREAM_PACKET_DURATION_30MS;

/* Clear the array of payload type values */
for (un32Count=0; un32Count<TB640_VP_GROUP1_MAX_PAYLOAD_TYPES; un32Count++)
{
        pStreamParams->aPacketProtocol[un32Count].PacketType = TBX_MEDIA_TYPE_INVALID;
        pStreamParams->aPacketProtocol[un32Count].un8TxProtocolType =  0xFF;
        pStreamParams->aPacketProtocol[un32Count].un8RxProtocolType =  0xFF;
}

/* Used for RFC2833 */
pStreamParams->aPacketProtocol[0].PacketType = TBX_MEDIA_TYPE _AUDIO_TELEPHONY_EVENT;
pStreamParams->aPacketProtocol[0].un8TxProtocolType = 101;
pStreamParams->aPacketProtocol[0].un8RxProtocolType = 101;

/* Used for main codec */
TB640_VP_GROUP1_STREAM_SET_PAYLOAD_TYPE (&(pStreamParams->aPacketProtocol[1]), TBX_MEDIA_TYPE_AUDIO_G723_1, 4, 4);

/* Activate tone relay */
pTdmParams ->Tone.fDtmfSuppressionEnabled = TBX_TRUE;
pTdmParams ->Tone.fDtmfCompleteSuppression= TBX_TRUE;
pTdmParams ->Tone.fDtmfToneRegeneration= TBX_TRUE;
pStreamParams ->fEnableRfc2833ToneRelay = TBX_TRUE;

/* Use DTMF AAL2 redundancy (RFC2833 only)
pTdmParams ->Tone.fDtmfToneAal2Redundancy = TBX_TRUE;
```

3. SIP INVITE containing iLBC-13k, telephony events

```
m=audio 49162 RTP/AVP 97 101           ← Using UDP port 49162
a=rtpmap: 97 iLBC/8000                 ← iLBC (default is 13k)  using payload type 97
a=rtpmap: 101 telephone-event/8000     ← RFC2833 for telephony using payload type 101
```

```
PTB640_VP_GROUP1_STREAM_PARAMS          pStreamParams;
PTB640_VP_GROUP1_TDM_PARAMS             pTdmParams;

pTdmParams =  &(pVpGroup->aRes[0].Params.Group1.Tdm);
pStreamParams  = &(pVpGroup->aRes[1].Params.Group1.Stream);

/* Set the default values */
TB640_VP_GROUP1_TDM_PARAMS_DEFAULT (pTdmParams);
TB640_VP_GROUP1_STREAM_PARAMS_DEFAULT (pStreamParams);

/* Choose the codec */
pStreamParams->Codec.MediaType = TBX_MEDIA_TYPE_AUDIO_ILBC;
pStreamParams->Codec.iLBC.SpeedRate = TBX_STREAM_ILBC_RATE_13_KBPS;
pStreamParams->PacketDurationMs = TBX_STREAM_PACKET_DURATION_30MS;

/* Clear the array of payload type values */
for (un32Count=0; un32Count<TB640_VP_GROUP1_MAX_PAYLOAD_TYPES; un32Count++)
{
        pStreamParams->aPacketProtocol[un32Count].PacketType = TBX_MEDIA_TYPE_INVALID;
        pStreamParams->aPacketProtocol[un32Count].un8TxProtocolType =  0xFF;
        pStreamParams->aPacketProtocol[un32Count].un8RxProtocolType =  0xFF;
}

/* Used for RFC2833 */
pStreamParams->aPacketProtocol[0].PacketType = TBX_MEDIA_TYPE _AUDIO_TELEPHONY_EVENT;
pStreamParams->aPacketProtocol[0].un8TxProtocolType = 101;
pStreamParams->aPacketProtocol[0].un8RxProtocolType = 101;

/* Used for main codec */
pStreamParams->aPacketProtocol[1].PacketType = TBX_MEDIA_TYPE _AUDIO_ILBC;
pStreamParams->aPacketProtocol[1].un8TxProtocolType = 97;
pStreamParams->aPacketProtocol[1].un8RxProtocolType = 97;

/* Activate tone relay */
pTdmParams ->Tone.fDtmfSuppressionEnabled = TBX_TRUE;
pTdmParams ->Tone.fDtmfCompleteSuppression= TBX_TRUE;
pTdmParams ->Tone.fDtmfToneRegeneration= TBX_TRUE;
pStreamParams ->fEnableRfc2833ToneRelay = TBX_TRUE;

/* Use DTMF AAL2 redundancy (RFC2833 only)
pTdmParams ->Tone.fDtmfToneAal2Redundancy = TBX_TRUE;
```

4. SIP INVITE containing iLBC-15k

```
m=audio 3400 RTP/AVP 98          ← Using UDP port 3400
a=rtpmap: 98 iLBC/8000           ← iLBC  using payload type 98
a=fmtp:98 mode=20                ← This indicates that it is iLBC-15k rather than 13k
```

```
PTB640_VP_GROUP1_STREAM_PARAMS          pStreamParams;
PTB640_VP_GROUP1_TDM_PARAMS             pTdmParams;

pTdmParams =  &(pVpGroup->aRes[0].Params.Group1.Tdm);
pStreamParams  = &(pVpGroup->aRes[1].Params.Group1.Stream);

/* Set the default values */
TB640_VP_GROUP1_TDM_PARAMS_DEFAULT (pTdmParams);
TB640_VP_GROUP1_STREAM_PARAMS_DEFAULT (pStreamParams);

/* Choose the codec */
pStreamParams->Codec.MediaType = TBX_MEDIA_TYPE_AUDIO_ILBC;
pStreamParams->Codec.iLBC.SpeedRate = TBX_STREAM_ILBC_RATE_15_KBPS;
pStreamParams->PacketDurationMs = TBX_STREAM_PACKET_DURATION_20MS;

/* Clear the array of payload type values */
for (un32Count=0; un32Count<TB640_VP_GROUP1_MAX_PAYLOAD_TYPES; un32Count++)
{
        pStreamParams->aPacketProtocol[un32Count].PacketType = TBX_MEDIA_TYPE_INVALID;
        pStreamParams->aPacketProtocol[un32Count].un8TxProtocolType =  0xFF;
        pStreamParams->aPacketProtocol[un32Count].un8RxProtocolType =  0xFF;
}

/* Used for main codec */
pStreamParams->aPacketProtocol[0].PacketType = TBX_MEDIA_TYPE _AUDIO_ILBC;
pStreamParams->aPacketProtocol[0].un8TxProtocolType = 98;
pStreamParams->aPacketProtocol[0].un8RxProtocolType = 98;
```

5. SIP INVITE containing PCMA, telephony events, VAD and RFC2198 for redundancy

m=audio 20334 RTP/AVP 8 101 13 121	← Using UDP port 20334
a=rtpmap: 8 PCMA/8000	← G.711 alaw using payload type 8
a=rtpmap:121 red/8000/1	← RFC2198 for redundancy using payload type 121
a=rtpmap: 101 telephone-event/8000	← RFC2833 for telephony using payload type 101
a=rtpmap: 13 CN/8000	← VAD comfort noise packets using payload type 13

```
PTB640_VP_GROUP1_STREAM_PARAMS          pStreamParams;
PTB640_VP_GROUP1_TDM_PARAMS             pTdmParams;

pTdmParams =  &(pVpGroup->aRes[0].Params.Group1.Tdm);
pStreamParams  = &(pVpGroup->aRes[1].Params.Group1.Stream);

/* Set the default values */
TB640_VP_GROUP1_TDM_PARAMS_DEFAULT (pTdmParams);
TB640_VP_GROUP1_STREAM_PARAMS_DEFAULT (pStreamParams);

/* Choose the codec */
pStreamParams->Codec.MediaType = TBX_MEDIA_TYPE_AUDIO_PCMA;
pStreamParams->PacketDurationMs = TBX_STREAM_PACKET_DURATION_20MS;

/* Clear the array of payload type values */
for (un32Count=0; un32Count<TB640_VP_GROUP1_MAX_PAYLOAD_TYPES; un32Count++)
{
        pStreamParams->aPacketProtocol[un32Count].PacketType = TBX_MEDIA_TYPE_INVALID;
        pStreamParams->aPacketProtocol[un32Count].un8TxProtocolType =  0xFF;
        pStreamParams->aPacketProtocol[un32Count].un8RxProtocolType =  0xFF;
}

/* Used for RFC2833 */
pStreamParams->aPacketProtocol[0].PacketType = TBX_MEDIA_TYPE_AUDIO_TELEPHONY_EVENT;
pStreamParams->aPacketProtocol[0].un8TxProtocolType = 101;
pStreamParams->aPacketProtocol[0].un8RxProtocolType = 101;

/* Used for RFC3389 */
pStreamParams->aPacketProtocol[1].PacketType = TBX TBX_MEDIA_TYPE_AUDIO_CN;
pStreamParams->aPacketProtocol[1].un8TxProtocolType = 13;
pStreamParams->aPacketProtocol[1].un8RxProtocolType = 13;

/* Used for main codec */
pStreamParams->aPacketProtocol[2].PacketType = TBX_MEDIA_TYPE_AUDIO_PCMA;
pStreamParams->aPacketProtocol[2].un8TxProtocolType = 8;
pStreamParams->aPacketProtocol[2].un8RxProtocolType = 8;

/* Used for RFC2198 */
pStreamParams->aPacketProtocol[3].PacketType = TBX_MEDIA_TYPE_AUDIO_RED;
pStreamParams->aPacketProtocol[3].un8TxProtocolType = 121;
pStreamParams->aPacketProtocol[3].un8RxProtocolType = 121;

/* Configure VAD */
pTdmParams ->Vad.fEnabled = TBX_TRUE;
pTdmParams ->Vad.SidGenerationScheme= TB640_VP_SID_GENERATION_SCHEME_DEFAULT;
pTdmParams ->Vad.CngGenerationScheme= TB640_VP_CNG_GENERATION_SCHEME_PT13;
pTdmParams ->Vad.NoiseFloor = TB640_VP_VAD_NOISE_FLOOR_LEVEL_MINUS_20_DBM;

/* Activate tone relay */
pTdmParams ->Tone.fDtmfSuppressionEnabled = TBX_TRUE;
pTdmParams ->Tone.fDtmfCompleteSuppression= TBX_TRUE;
pTdmParams ->Tone.fDtmfToneRegeneration= TBX_TRUE;
pStreamParams ->fEnableRfc2833ToneRelay = TBX_TRUE;

/* Use RFC2198 encapsulating RFC2833 for DTMF tone relay */
pTdmParams ->Tone.fDtmfToneAal2Redundancy = TBX_FALSE;
```

6.2.2.5.7 Jitter buffers

In a VoIp network, packets may be delayed or re-routed due to numerous conditions usually out-of-the-control of any VoIp endpoint or equipment. A failure in an Ethernet switch or route somewhere may also cause packet loss, out-of-order packet receiving or receiving bursts. Therefore, the VoIp device cannot assume that all RTP packets will be delivered just-in-time to be played onto the TDM bus. It needs to store the received packets into a queue, re-order them and schedule them accordingly to be played at regular interval onto the TDM bus. When too many packets to fit into the jitter-buffer are received, the surplus packets will be dropped by the VoIp device. If not enough packets are received, the jitter-buffer will play the packets stored into its queue until it is empty. At that point, silence will be played on the TDM interface.

Although a jitter-buffer is necessary to protect against network impairments, it also injects delay into the receiving path (i.e. the jitter-buffer need to hold the packets a little while before playing them). Thus, larger jitter-buffers will accommodate more unreliable networks (in terms of packet latency) but will also introduce a larger delay in the voice stream and then reduce the conversation quality. Therefore, an application will want the jitter-buffer to be large enough for the network but the smaller possible to minimize delay.

To achieve this goal, the jitter-buffers included in voice processing group1 allow two different configuration mode: fixed and adaptative. A fixed-mode jitter buffer will always have a constant delay according to its depth (e.g.. 20 msec). This will ensure that the latency in the voice stream will never change. On the other hand, if the network jitter never goes over 10msec, the jitter buffer will still hold every packet for the time duration configured by the user (e.g. 20 msec)

The adaptative jitter-buffer is designed to adjust to the smallest buffer length to accommodate the network while allowing the buffer to grow to a certain limit. To do so, the application needs to configure an initial delay period (msec), minimum delay period (msec), a maximum delay period (msec), an adaptation period (sec), a deletion threshold (msec) and a deletion method.

The initial delay period is a delay that the VoIp device will wait when the very first RTP packet is received. It allows the jitter buffer to fill (usually up to the middle) before starting the playback. This is to avoid hearing gaps when opening a new connection and that the second RTP packet is delayed by the network. The minimum and maximum delay period are the range (or depth) of the jitter buffer. The adaptation period is the time that the VoIp device will take to adapt toward lower jitter buffer length if it detects that the network jitter is smaller. The deletion method is to select if the exceeding packets are to be dropped immediately when the maximum buffer length is reached (hard deletion method) or if the devices tries to do deletion on packets that may have less impact on the voice stream (smooth method). The ‘hard’ method will ensure the delay never exceeds the maximum jitter-buffer length but may create gaps in voice when a packet is dropped. The ‘smooth’ method has less chance of creating voice gaps but the jitter-length (thus the latency) may exceed the maximum configured value. The deletion threshold is the absolute maximum jitter depth after which packets are deleted regardless of the deletion method.

☞ When transmitting modem or fax data through a clear-channel or a G.711 codec, it is important to configure the channel with fixed jitter-buffer. Modem and fax signals are greatly disturbed by variation in the voice stream. Clear-channel modem/fax transmission should be used <u>only</u> when T.38 protocol is not available since the later is designed to counter the effect of latency and variable jitters of VoIp network.

6.2.2.5.8 VAD

Voice activity detection (VAD) is primarily used by the packetization engine to try saving bandwidth on the network. When silence period is detected on the TDM side of a VoIp connection, the packetization engine will encode a special message (SID) telling the remote-end about the silence period and characteristic to generate comfort noise instead of encoding the silence into a regular RTP packet. As this feature affect the voice integrity (compared to the TDM stream), it is not recommended to activate it for data transfer (modem or fax).

6.2.2.5.9 T.38 Fax relay

Although fax data could be transferred over a non-codec (i.e. clear channel) RTP stream without loss of integrity on local networks, there are still parameters that can affect efficient T.30 transfer over a packet network (i.e. latency may affect carrier-frequency negotiation). T.38 allows a fax transmission over a network where it is not possible to guarantee low latency (i.e. when the communication goes over a satellite link). For this reason, it is always recommended to use T.38 instead of clear-channel communication when it is available. T.38 encapsulation allows the transfer of such fax information over a packet network without loss of integrity. In order to transfer the fax, a T.38 fax relay voice processing group first needs to terminate the fax call on the TDM side (i.e. demodulate from V.21 or other speed grade modulation standard). Then it encapsulates fax data into T.38 frames and sends them onto the packet network following RFC2833. The receiving end will retrieve those RFC2833 packets and modulate the data back onto its TDM interface thus relaying the fax.

☞ The TB640 cannot store nor generate fax data to/from the host. The fax data is taken from the TDM side and converted into RFC2833 on the IP network. There is no interaction with the host application other than to notify statistical and state change information. The TB640 does T.38 "fax relay", not "fax termination".

☞ "Fax termination" is accomplished using the FaxServer product integrated with the TBStreamServer. This product is opened-sourced available from TelcoBridges.

Fax and modem calls are traditional equipment that are here to stay and that shouldn't be ignored. With gateway applications, especially when using VoIp, fax transmission need to be dealt with using the proper transmission protocol (i.e. T.38) in order to carry the fax information efficiently across networks. Since there is nothing different between a typical voice and a fax call during its setup phase, the fax transmission need to be detected live by analyzing the content and detecting tone typical to fax and modems. Once the data transmission is detected, both sides of the call must switch from currently used codec either to T.38 protocol or to a clear-channel mode (no codec, fixed jitter-buffer, low latency, no echo depending on the fax negotiated speed). The tricky part is that both sides needs to do this transition on their own as there is no 'standard' in-band protocol telling them to do so. Larger companies such as Cisco inserted their own proprietary protocol embedded between RTP frames so that two Cisco boxes would agree on the new parameters to choose to carry the fax.

Another way to deal with this issue is to be flexible in switching from one codec to the other quickly. For example, once a fax transmission is detected, an application could immediately change the current codec to use clear-channel instead (lowering the packet duration, fixing the jitter-buffers, disabling the echo, etc). Once in that mode, clear-channel RTP packets will be generated toward the other end. The VoIp unit will then alert the host application once it detects that the received payload type has changed (from the 'old' codec type to a new one). Thus, the application will know if the remote side has switched to a clear-channel codec or to an RFC2833 channel by looking at received packets. It can then re-adjust (if necessary) by switching to the same codec type and continue the fax transmission.

☞ A TB640 VpGrp1 Fax resource can only be used to process one fax transmission. Once the transmission is done, the resource needs to be de-allocated. This is due to the Fax state machine handling in the physical voip devices. Thus, an application cannot pre-allocate T.38 VpGrp1 resource as it does for voice-related resources.

6.2.2.5.10 RTCP

The RTCP protocol is an optional protocol that 'controls' a single RTP session. Following RFC3550, this protocol gathers statistical and provisioning information about a specific RTP session. Although it is not mandatory for both sides of a VoIp stream to support RTCP in order to access this statistical information, it is recommended since it will allow the protocol to calculate useful debugging statistics such as the round-trip delay between both VoIp endpoints. The application can decide to activate RTCP on a per-channel basis. The only restriction (enforce by RFC3550) is that an RTP stream controlled by an RTCP stream needs be have an even UDP port number (assuming n)while the RTCP stream will use the next odd UDP port (n+1). Therefore, if the RTCP option is activated, the blade will refuse to open a stream resource on a odd UDP port number.

☞ An RTP stream must always be allocated with an even UDP port number if RTCP is to be used. The blade will then use the following port number (odd) for the RTCP control protocol monitoring that stream.

Aside from statistical information, RTCP also adds new information to be exchanged with the remote VoIp endpoint. A textual name for the VoIp stream will be sent over the RTCP control path within a SDES-CNAME RTCP packet section. This name is used to identify the session in a human-readable fashion. Another RTCP packet named 'BYE' indicates the termination of the RTP session. Yet another RTCP packet named 'APP' can be used to exchange user-to-user information between the two VoIp endpoints. If activated, all of those RTCP packet types may generate notifications toward the host application when they are received. The reader should refer to RFC3550 for further details of RTCP information.

6.2.2.6 VP group1 Applications

6.2.2.6.1 Tdm to VoIp connections

Numerous types of application exist that make use of both TDM and VoIp connections. For example, in a typical signaling gateway application, calls are terminated on either one interface (PSTN for TDM calls and VoIp for packet network) and bridge them to the other. In those cases, multiple types of signaling stacks may be used such as ISDN, CAS or SS7 on the PSTN side and SIP or H.323 on the VoIp side. For the codec usage, it really depends on what type of packet network(s) the application connects to. Wireless networks may require the gateway to support CDMA codecs (EVRC, QCELP8/13, SMV) or AMR while a packet cable network might require G.711 and G.728. Of course, other VoIp features are always required to adapt to the different network conditions (i.e. echo cancellations, jitter buffers, RTCP control protocol, T.38 fax relay, etc).

Other applications such as prepaid or postpaid systems may require connecting to different types of networks, also involving different codecs. These systems usually offer a 'least cost routing' feature which changes the routing of each call according a dial plan and/or different network providers' fees during different time of the day. Routing calls through a VoIp carrier might become cost effective during certain peak period of the day compared to traditional PSTN routing. Furthermore, since other IVR (interactive voice response) services are also required (conferencing, prompt playback, tone generation and detections, etc), the TB640 product is well suited for any of these situation as it supports all the required subsystems, as well as signaling stacks, on the same blade, running concurrently.

6.2.2.6.2 VoIp to VoIp transcoding

One of the simplest applications in VoIp is to do codec transcoding without dropping to the PSTN (TDM) interface. Wireless telephony, due to a limited bandwidth availability (compared to fixed land lines) has no choice but to use compression codecs such as AMR or EVRC. Therefore, when a call needs to be switched to the PSTN, existing mobile switches need to convert the voice stream back to G.711 before routing the calls outside their network. They usually depend on external media gateways to do the processing-intensive conversion.

6.2.2.6.3 Fax relay over VoIp network (RFC2833)

As mentioned before, fax and modem equipment cannot be ignored in a network because of their widely deployment. Almost every Voip system will need to support fax transport either using clear-channel mode or using T.38. The application range goes from standard gateways to fax/modem relay farms. In any cases, supporting fax relay is a must.

6.2.2.6.4 TDM Echo cancellation

As mentioned in section 6.2.2.5.2, echo cancellation may be required when interfacing with network-edge TDM equipment or when routing calls to a packet network. It may also be used in applications where VoIp is not involved to resolve an audible echo issue. Using the pure-TDM voice processing resources, the TB640 VoIp subsystem can create unidirectional or bidirectional echo cancellation resources that can be used in conjunction with other part of the system (e.g. conference, trunking, etc.). Creating high-density echo canceller system (i.e. 15k connections) is therefore possible by combining multiple blades together.

7 CONNECTIONS

Connections on the TelcoBridges family of adapters are possible between any type of resources (channel and processing resources). There is virtually no limitation on which resource can be linked together. The application must previously allocate and/or define resources which need to be connected. A connection needs at least to include a channel resource to get outside of the adapter but can include as many resources as required. The connection manager will automatically allocate resources that have been defined by the application. The connection manager will also automatically de-allocate resources that have been defined by the application. The connection manager programs the connection engine on the adapter following a list of paths describing the connection. A path description can be seen as a finite list of resources that must be linked together.

Every path between all resources must be specifically described. A path is composed of two resources plus a flag indicating whether the two resources must be connected in full-duplex or half-duplex. The first resource listed in a path description is the source and the second is the destination. When a resource is flow through, two path descriptions are required to connect each side of the resource. The first connection listed is always connected to the input logical side of the resource while the second one connects to the other side.

A Higher level view of the connections on the system is show in `Figure 57`.

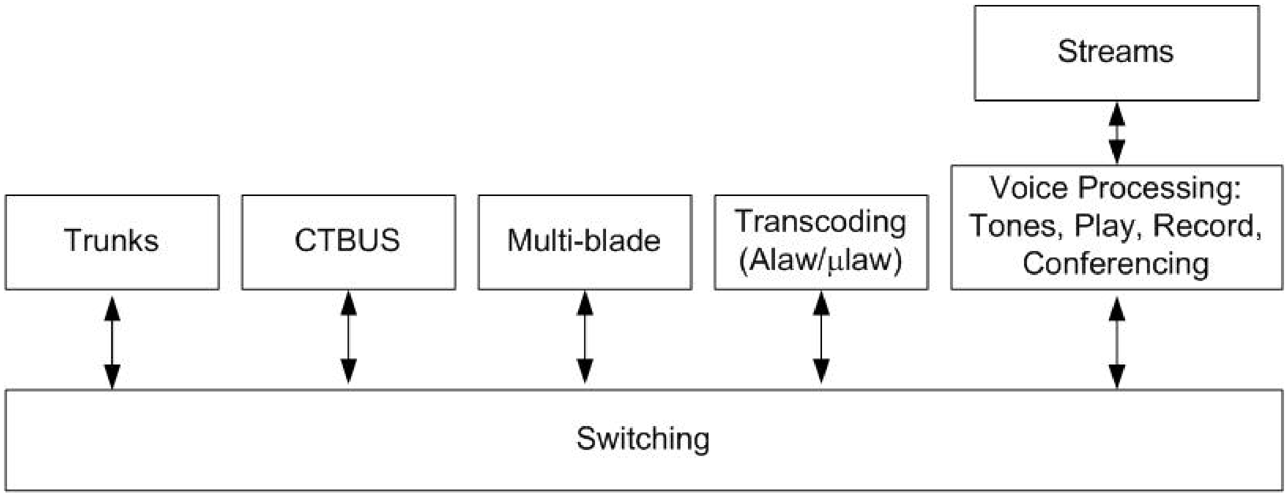


Figure 57: TB640 high-level view of connections

7.1 Path description

Here are three different cases illustrating how to describe a single connection with a path description list:

- **Case 1**

The following connection could be described with two different path description lists.

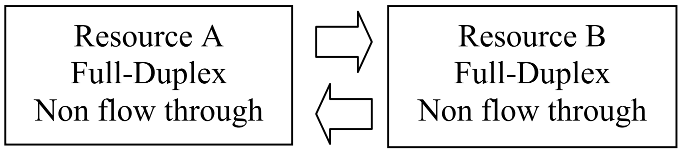


Figure 58: Path description to connect two full-duplex resources

The connection between resources A and B could be described with the following path description list:

1. Link resource A to B, full-duplex

The connection between resources A and B could also be described with the following path description list:

1. Link resource A to B, half-duplex
2. Link resource B to A, half-duplex

Path descriptions are both equivalent.

- **Case 2**

The following connection could be described with four different path description lists.

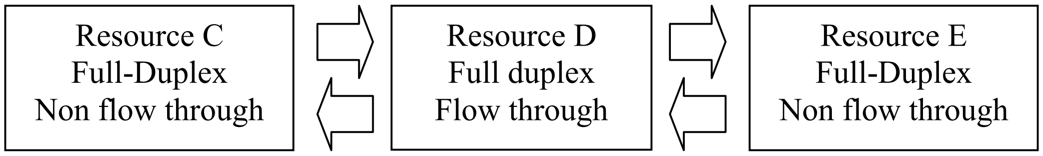


Figure 59: Path description to connect three full duplex resources

The connection between resources C, D and E could be described with the following path description list:

1. Link resource C to D, full-duplex
2. Link resource D to E, full-duplex

The connection between resources C, D and E could also be described with the following path description list:

1. Link resource C to D, half-duplex
2. Link resource D to E, half-duplex
3. Link resource E to D, half-duplex
4. Link resource D to C, half-duplex

The connection between resources C, D and E could also be described with the following path description list:

1. Link resource C to D, full-duplex
2. Link resource D to E, half-duplex
3. Link resource E to D, half-duplex

The connection between resources C, D and E could also be described with the following path description list:

1. Link resource C to D, half-duplex
2. Link resource D to E, full-duplex
3. Link resource D to C, half-duplex

They are all equivalent.

- **Case 3**

It is possible to connection the following asymmetric resources with a single path description list.

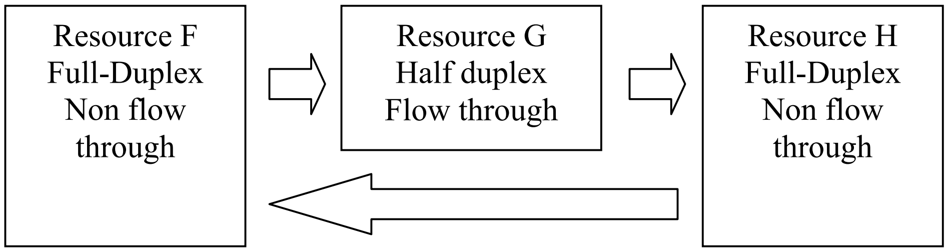


Figure 60: Path description to connect asymmetric resources

The connection between resources F, G and H could be described with the following path description list:

1. Link resource F to G, half-duplex
2. Link resource G to H, half-duplex
3. Link resource H to F, half-duplex

- **Case 4**

It is possible to connect the following resources with a single path description list.

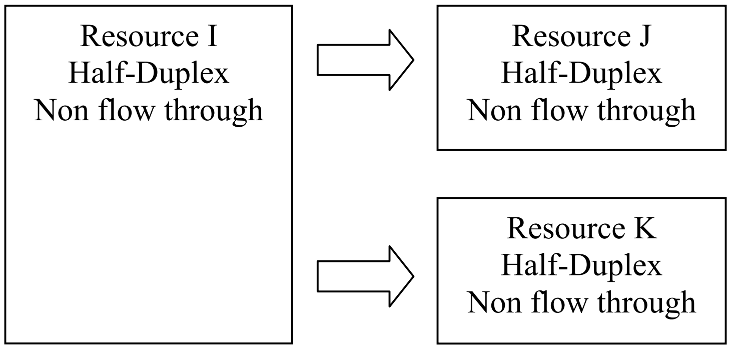


Figure 61: Path description to connect resources (single source to multiple destinations)

The connection between resources I, J and K could be described with the following path description list:

1. Link resource I to J, half-duplex
2. Link resource I to K, half-duplex

- **Case 5**

It is NOT possible to connect the following resources.

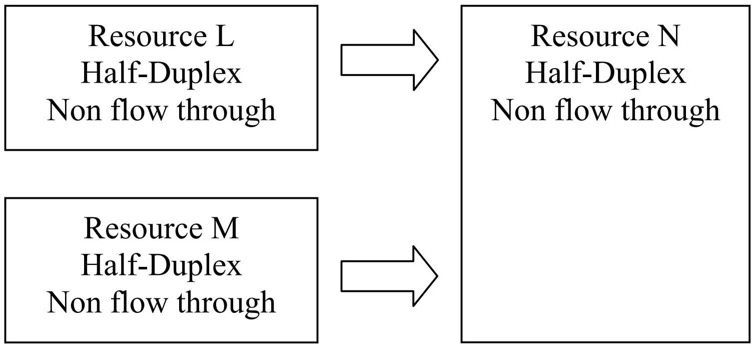


Figure 62: Path description to connect resources (multiple sources to single destination)

7.2 Code example #6: Connect two full-duplex resources using the implicit filter

Here is a code example showing how to connect two full-duplex resources using the implicit filter to retrieve response message in synchronous like behavior:

…

```
TBX_RESULT_API                 APIResult;
TB640_RESULT                   ConnectResult;
TBX_MSG_HANDLE                 hMsg;
TBX_FILTER_HANDLE              hFilter;
TBX_CONNECTION_HANDLE          hConn;
TB640_REQ_CONN_RES_CREATE *    pRequestCreate;
TB640_RSP_CONN_RES_CREATE *    pResponseCreate;

/* Initialize local variables */
hFilter = 0;
```

…

```
/* in_hLib is the library handle returned by TBXOpenLib call
in_hAdapter is an adapter handle returned by TBXGetAdaptersList call
in_hResSrc is the resource handle of the source
in_hResDst is the resource handle of the destination
out_hConn is the connection handle returned by this code */

/* Get message buffer */
APIResult = TBXGetMsg(
  in_hLib,
  sizeof( TB640_MSG_CONN_RES_CREATE ) + sizeof( TB640_PATH_DESCRIPTION ),
  &hMsg );

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Initialize message header */
      TBX_FORMAT_MSG_HEADER (
        hMsg,
        TB640_MSG_ID_CONN_RES_CREATE,
        TBX_MSG_TYPE_REQUEST,
        sizeof (TB640_MSG_CONN_CREATE)
        in_hAdapter,
        0,
        0);

      /* Set the message payload */
      pRequestCreate = (PTB640_REQ_CONN_CREATE)
        TBX_MSG_PAYLOAD_POINTER( hMsg );
      pRequestCreate->un32MsgVersion = 1;
      pRequestCreate->un32PathDescCount = 1;
      pRequestCreate->aPathDesc[0].fFullDuplex = TRUE;
      pRequestCreate->aPathDesc[0].hResSrc = in_hResSrc;
      pRequestCreate->aPathDesc[0].hResDst = in_hResDst;

      /* Send the allocation message to single adapter. Note that the last
      argument is non NULL. This call will return a handle on an implicit
      filter. This filter can be used to retrieve the response to this request.
      */
      APIResult = TBXSendMsg (
        in_hLib,
        in_hAdapter,
        &hMsg,
        &hFilter );
}

/* Insert code here for the operations to be done in parallel */
/* Multi-threading is recommended to obtain best performance and take full
advantage of the asynchronous capabilities */

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Use implicit filter returned by TBXSendMsg call to retrieve response
      message. Call blocks for maximum 5 second. */
      APIResult = TBXReceiveMsg (
        in_hLib,
        hFilter,
        5000,
        &hMsg );
}

if ( TBX_RESULT_SUCCESS( APIResult ) )
{
      /* Retrieve the message payload */
      pResponseCreate = (PTB640_RSP_CONN_CREATE)
        TBX_MSG_PAYLOAD_POINTER( hMsg );
      CreateResult = pResponseCreate->Result;
      out_hConn = pResponseCreate->hConn;

      if ( TBX_RESULT_SUCCESS( CreateResult ) )
      {
            printf( "SUCCESS: Connection of full-duplex resources\n" );
      }
      else
      {
            printf( "FAILURE: Connection of full-duplex resources\n" );
        }

        /* Release message buffer */
        APIResult = TBXReleaseMsg (
          in_hLib,
          hMsg );
}

…

if (hFilter != 0)
{
        /* Destroy the implicit message filter */
        TBXDestroyMsgFilter ( in_hLib, hFilter );
}

…
```

8 SIGNALING

The signaling information/code contained in this document/product are based on the best information we have available. Although it has been tested successfully with other piece of signaling equipment, we cannot guarantee that it will conform to the usage of any particular switch in the field.

8.1 Overview

8.1.1 Architecture

One of the primary functions of the TelcoBridges family of adapters is to provide signaling capabilities to interconnect with the PSTN or other types of systems. In the signaling world, there are many variants of protocols (even standardized ones) depending on the type of equipment the adapter is connected to. TelcoBridges supports many variants of the ISDN Q.931 as well as multiple variants of channel associated signaling such as CAS R1 and R2.

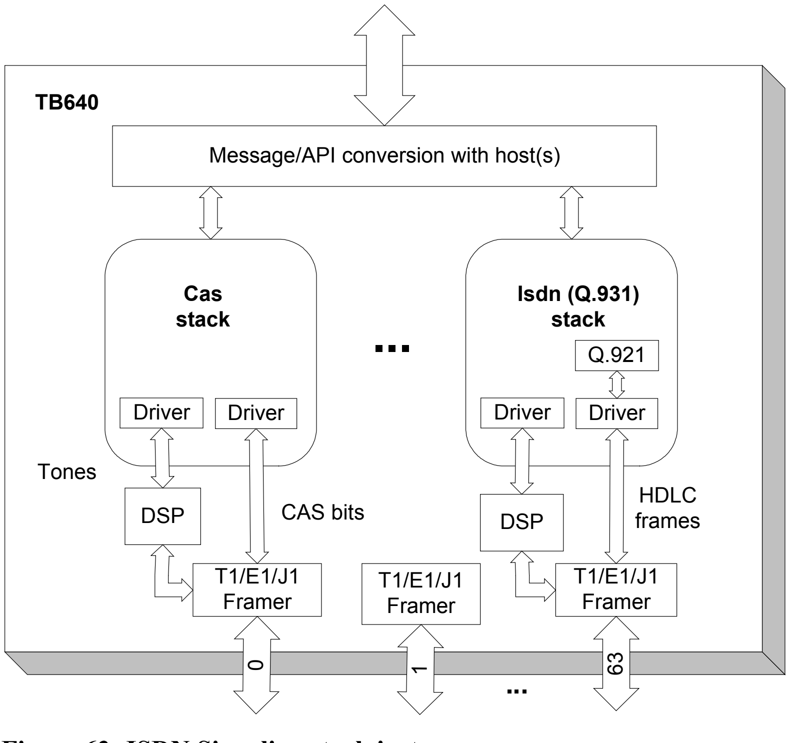


Figure 63: ISDN Signaling stack instances

At this point, nothing is really different from what a customer can find with most telecommunications equipment vendors. The major difference with TelcoBridges family of products is that the signaling stacks and protocols can be individually selected, configured and controlled on any of the trunks available on the adapter while the system is up and running. Resource allocation (such as DSPs for tone detection and generation) is done automatically when selecting the signaling protocol and CAS method desired for a particular trunk (as shown on `Figure 63`). This enables the user to bridge between different types of switches using the same adapter and equipment. Some configurations require a single stack instance to control multiple trunks (such as the ISDN NFAS mode). In this case, the multiple trunks are assigned under the control of that single stack instance and resources are assigned accordingly.

This “isolated-stack” architecture also allows a stack instance to be check-pointed (saving its states regularly) to another instance therefore preventing active call-loss if there is a software failure. Being protected from one-another, the stacks are isolated from the physical devices they control thus reducing the risks of corruption of active voice calls.

The API messages exchanged between the adapter and the host(s) are almost a direct mapping between the Q.931 primitives from the signaling stack and the API messages. This type of methodology is preferred even if it does add a little complexity to the user application because it also gives a tremendous control over the protocol information elements and parameters.

8.1.2 CCS

The common channel signaling associated with every trunk primarily uses Q.921 LAPD frames sent and received over a designated D-channel. All the configuration parameters are specific to every trunk making it possible for the user to individually configure the trunks as he sees fit for his application. Every signaling stack instance have its own parameters including the network termination mode (user versus network), the type of associated signaling, the switching equipment type (e.g. DMS-100, 4ESS, 5ESS, etc.) and other information related to the protocol (e.g. ISDN numbers, information elements, etc).

Multiple types of channel associated signaling can be chosen depending on the selected trunk and configuration options. T1 robbed-bit and E1 CAS signaling information is retrieved from the multi-frame by the framers while R1 tones are processed by DSPs and then forwarded to the signaling stack. Once the information is processed, an appropriate message is sent to the host(s) containing the high-level information gathered. The host(s) won’t receive, for example, the raw ABCD bits from the framers to process them (only for information in cases where they are required). This eliminates a huge exchange flow of messages between the adapters and the host libraries.

Another type of channel associated signaling is the MFC-R2 which uses multi-frequency tones to carry in-band information during connection setup. This type of information is also retrieved/inserted into the data streams by the DSPs under the control of the signaling stack. Again, this allows a greater number of trunks to be processed since there is no need for a huge traffic exchange with the host.

8.1.3 Q.SIG

Q.SIG is an extended version of the ITU-ISDN reference module to allow interconnection between PINX [Private Integrated services Network Exchange] without any references to a “user” or “network” side (the protocol is symmetrical). This extension creates two new endpoints in the network called “Q” and “C” where “Q” is the logical signaling endpoints within the private network and “C” is the physical connection endpoint to the PINXs. The layer 2 is still using LAPD but replace Q.931 layer 3 protocol by its own. This reflects to the host application by other types of messages for Q.SIG signaling. This mode of operation is, again, selectable per trunk independently of others.

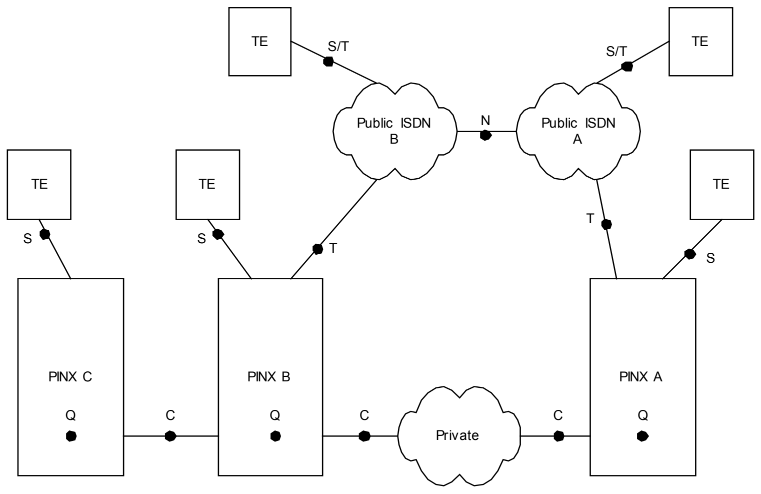


Figure 64: Network with Q.SIG endpoints

8.2 Q.931 ISDN Signaling

8.2.1 Trunk configuration

Before a signaling stack can be started, the underlying trunk must be configured correctly. The configuration depends on the Q.931 switch variant that will be used in the system. When multiple trunks are attached to the same stack instances, all trunks must have the same configuration.

Table 12: Q.931 ISDN variants

Switch variant (and supported deltas)	Trunk type expected
TB640_ISDN_VARIANT_4ESS	T1/J1
TB640_ISDN_VARIANT_AUS_PRI *TS-014* *TS-038*	T1/J1
TB640_ISDN_VARIANT_5ESS_PRI	T1/J1
TB640_ISDN_VARIANT_NET5 *French delta* *German delta* *UK delta* *China delta* *Korea delta* *Singapore delta*	E1
TB640_ISDN_VARIANT_DMS_PRI *Nortel DMS-100* *Nortel DMS-250*	T1/J1
TB640_ISDN_VARIANT_US_NI2_PRI	T1/J1
TB640_ISDN_VARIANT_HONG_KONG_PRI	T1/J1
TB640_ISDN_VARIANT_JAPAN_INS	T1/J1

8.2.2 D-Channel logical status

The signaling stack is decoupled of the transport mechanism (which is a variant of HDLC over the trunks) but is still affected by the trunk state. When the physical line is up and the D-channel goes up, the application will receive two *TB640_MSG_ID_ISDN_NOTIF_STATUS_INDICATION* events, one with the value *TB640_ISDN_STATUS_IND_VALUE_PHYSICAL_LINE_ACTIVATED* and one with the value *TB640_ISDN_STATUS_IND_VALUE_LINE_READY*. When a trunk line goes down, the stack is no longer able to communicate with its peer if its communication channel (also called the D-Channel) was located on that trunk. There are two different levels of error that can occur. The first error level is tied to the physical layer and is affected by the regular alarms of the trunks (Loss-of-signal, Remote-alarm-indication, etc). When such an event occurs, the stack sends an event to the user-application (*TB640_MSG_ID_ISDN_NOTIF_STATUS_INDICATION)* containing the value *TB640_ISDN_STATUS_IND_VALUE_PHYSICAL_LINE_DEACTIVATED*. The second level of error concerns the D-Channel state. The D-Channel state represents the state (up or down) of the logical communication link between the stack and its peer. When this link is lost, the stack sends an event to the user-application (*TB640_MSG_ID_ISDN_NOTIF_STATUS_INDICATION*) containing the value *TB640_ISDN_STATUS_IND_VALUE_LINE_NOT_READY*. Note that a user-application can always read this state using the *TB640_MSG_ID_ISDN_STATES_GET* API message. No connection openings can occur when the D-Channel is down (all attempts will be refused). Pending connections may time-out and be closed by the stack while active calls may stay opened until manually disconnected by the user-application. This behavior is variant-specific. The user application should be careful when closing active calls when the D-Channel is down because it does not mean the peer stack will do it as well. Thus, when the logical link will be up again, the two stacks will not have the same knowledge of which timeslot is free and which one is used. To make sure the user-application knows the exact state of the D-channel at all time, it should first create an event filter to capture ISDN notifications coming from the stack and then use *TB640_MSG_ID_ISDN_STATES_GET* to retrieve the current state. Following that sequence will guarantee the application will have an exact knowledge of the D-Channel state (and never miss the event).

8.2.3 Bring up sequence

Depending on the chosen switch variant, the LAP-D communication between the two peer stacks is either initiated by the user-side, network-side or both. In some switch variants, a D-Channel timeout may occur if one of the two protocol stacks is not "ready" (or the trunk line is down). Some variants do not restart the LAP-D communication establishment until certain events occur (i.e. the user tries to open a connection). If the user-application is not careful in monitoring the D-Channel status (discussed in section 8.2.2), the very first call might be lost because the LAP-D communication wasn't yet established. Thus, the application should respect a sequence before trying to open the very first user call. First, after the physical line is up, the application should monitor the D-Channel status to see if the logical link is up. When the D-channel logical link is down, the application should send a *TB640_MSG_ID_ISDN_CMD_WAKEUP_REQUEST API* message on the trunk to "kick-start" for some switch variant. The application will then receive another event about the D-Channel status (either up or down). The user application should continue executing the wake up command until it sees the event indicating the D-Channel is now up (or declare a fatal error because the peer switch wasn't able to answer after a certain delay). Once the channel is up, the user-application can establish ISDN calls.

This wakeup sequence is only necessary on certain switch variants and protocol side summarized in Table 13. In a telecom network, it is assumed that the network-side (e.g. a DMS-100) is always up before the user-side. But, in smaller networks, it would be possible to have the user-side up first. In those cases, the application controlling the stack should follow the bring up sequence to 'discover' the remote side.

Table 13 - Bring up sequence behavior

Switch variants	Network-side responsibility to discover the remote	User-side responsibility to discover the remote	Behavior
4ESS	No	Yes	Retries indefinitely until up
AUS	No	Yes	(*) Tries only once after the physical line is brought up.
5ESS	No	Yes	Retries indefinitely until up
NET5	No	Yes	(*) Tries only once after the physical line is brought up.
DMS	No	Yes	Retries indefinitely until up
US NI2	No	Yes	Retries indefinitely until up
HONG KONG	No	Yes	(*) Tries only once after the physical line is brought up.
JAPAN INS	No	Yes	Retries indefinitely until up

Important note:

The user application should send one wakeup request only for variants/protocol side marked with a (*) in Table 13. Also, the wake up request must be sent only while in the following state:

TB640_ISDN_STATUS_IND_VALUE_PHYSICAL_LINE_ACTIVATED has been received **AND**
TB640_ISDN_STATUS_IND_VALUE_LINE_NOT_READY has been received.

Failure to follow this sequence will result in the application thinking it has sent the wakeup request but will never see the effect (i.e. wakeup has no effect when the physical line is not activated – the stack will never send another *TB640_ISDN_STATUS_IND_VALUE_LINE_NOT_READY* event).

8.2.4 Call handle and user contexts

Dealing with a signaling protocol stack equals dealing with many asynchronous events related to different contexts. In a typical application, the concept of a “call” represents all the different information and states for a particular communication between two endpoints in a specific system. An application dealing with this “call” usually wants to retain information about the two peers (names, numbers, resources handles to which the call is connected to, billing information, etc.) during the lifetime of the call. This information is kept by the user-application and needs to be referenced each time the call state changes. From the perspective of a protocol stack, a call “lives” from the first request (or notification) to establish the call until the call is terminated by one of the two peers. During the life of the call, it creates a unique handle (a call handle) to identify this call and internal states. The end-user application can then make the correlation between its own call context information (kept in its memory) and this unique identifier. When receiving a call, the hCall is present in the notification message (TB640_EVT_ISDN_NOTIF_CALL_PRESENT_INDICATION). When generating a new call (TB640_REQ_ISDN_CMD_CONNECT_REQUEST), the response (TB640_RSP_ISDN_CMD_CONNECT_REQUEST) will have the hCall.

The easiest (and recommended) way to make this correlation in the user application is to use an hash-table or something similar (depending on the number of calls the application is designed to process). Since every call made from or to the stack is identified by a unique hCall (call handle) allocated by the TB640, the user application can create an hash-table (or do a linear research when the number of calls is not high enough) to match an hCall to call context information (typically referenced by a pointer or a database index). Using this method will ensure the user application never to loose a call context since it can always go back into its hash-table and list the currently opened call(s).

[The section talking about the “user call context” has been removed – the suggested way to reference the call is to always use the hCall (call handle) provided by the TB640 ISDN stack. See the *isdn* sample program state machine for details on how to handle this (isdnstates.pdf)].

8.2.5 API request/response vs ISDN messages

One source of confusion when first developing with the TB640 ISDN API is the difference between the request/response messaging system and the ISDN Q.931 messages. In TB640 terminology, a "request" is a message that is sent from the user application to the blade and will always be answered by a "response" message (see section 2.1). It means that when a user application sends a Q.931 API message to the blade, it will always be answered by a response containing a result code (usually OK or an error code). This result code refers to the delivery of the Q.931 message to the protocol stack and does not imply the success or failure of the actual Q.931 requested operation (e.g. SETUP message being delivered to the peer). A Q.931 protocol stack involves timers and state machines that reflect the progress of a call and cannot always return answers to a request right away. Therefore, the user application needs to send multiple Q.931 requests (that will each be answered by a "response") to establish a call. Many Q.931 API message sequences used during a call establishment are presented in section 8.2.10 but those don't show the responses sent by the board to acknowledge every requests. Therefore, the Q.931 ISDN state changes from the stack will be delivered to the user application through a series of notifications (events) containing new call states.

8.2.6 Restart Procedure

RESTART is an optional Q.931 primitive (depending on the switch variant) that is used to restore a B-channel into the idle state in case of mismatch between the host application, the on-board ISDN stack and the peer ISDN stack. According to the specifications, the RESTART primitive can only be sent when the call is in the 'idle' or in the 'talking' state. Through a mechanism of acknowledgement, a stack (and host) knows if the peer ISDN stack has accepted/processed the RESTART request.

Using the TB640_MSG_ISDN_CMD_RESTART_CHANNEL_REQUEST (i.e. RESTART) message on a channel (trunk resource handle) or a trunk (trunk handle) will have the following effects:

- The specified channel or all the channels in the trunk on the local side will be put to idle if the RESTART operation succeeds
- There will be an Q.931RESTART primitive sent to the remote stack
- The application will receive a TB640_MSG_ISDN_NOTIF_STATUS_INDICATION event with a Status Indication of TB640_ISDN_STATUS_IND_VALUE_RESTART_ACK (i.e. RESTART_ACK). The value can be TBX_TRUE if it succeeded or TBX_FALSE if it fails.
- The ISDN stack has an internal timeout (named T316) which is set at 120 seconds (per specifications). Upon the first expiry of this timeout, the stack will automatically send another RESTART request to the peer stack and restart the T316 timer. If this second timer expires, the RESTART procedure is considered as 'failed' and the B-channel will be put in 'maintenance' mode (no longer accepting calls). This behavior is common to all switch variants and sides (network and user).

The restart procedure is symmetrical and behaves the same from the network-side or user-side. All switch variants supports the RESTART primitive although most of them won't do anything with it. Only switch variants DMS-100 and NI2 make active use of RESTART primitives (this is done automatically). 4ESS and 5ESS have similar functionalities but implemented with SERVICE primitive instead (also done automatically). No specific action is required by the host application when an ISDN stack is brought up but it can make use of the RESTART primitive to recover from state mismatch as mentioned before. It is possible that some ISDN equipment don't implement correctly the requirements of the RESTART primitive. Thus, it is possible for the application to send a RESTART primitive and to get a RESTART_ACK(failed) answer. In those cases, the B-channel becomes locally unavailable ('maintenance mode') but can be recovered with the TB640_MSG_ID_ISDN_CMD_RESET_CHANNEL_REQUEST request. This request will put back the local B-channel to the idle state.

8.2.7 B-Channel status

B-channels (also called bearer channels or simply 'voice timeslots') are the timeslots carrying data or voice when an ISDN call is active. Usually, a B-channel is always considered 'in-service' meaning it is available for a new call or is currently transporting a call. Note that the stack will refuse any incoming or outgoing calls on a B-channel that is not 'in-service'.

In some situations, a B-channel can be switched (manually or automatically) to another state where calls are no longer allowed: out-of-service or maintenance. When a B-channel is in the state ‘out-of-service’, it usually means that it is not used at all by the ISDN stack. Cases where timeslots of a trunk are shared between the ISDN stack and some other applications would put the non-ISDN timeslot as ‘out-of-service’. B-Channels that are in maintenance means they currently cannot carry voice/data traffic as they are ‘under-test’ by some external means (e.g. a bit-error tester).

Automatic state change can occur when a RESTART Q.931 primitive sent to the remote stack is not answered within a timeout period. Upon timeout expiry, the B-channel is set to maintenance mode automatically thus preventing other calls to be made until the B-channel is ‘repaired’. The B-channel can be restored to the ‘in-service’ state by issuing a successful TB640_MSG_ID_ISDN_CMD_RESTART_CHANNEL_REQUEST command message. If, for some reasons, the remote stack always refuses the RESTART Q.931 primitive, the last resort is to use the TB640_MSG_ID_ISDN_CMD_RESET_CHANNEL_REQUEST command message which will reset the local state of the stack (ignoring the states of the remote stack). B-channel state modification commands are also available to return to an ‘in-service’ state but are not available for all stack variants as mentioned below.

The states of each B-channel can be queried at any time using the command message TB640_MSG_ID_ISDN_STATES_BCHANNEL_GET. Notifications are also sent to the host application when a B-channel changes from one state to the other (TB640_MSG_ID_ISDN_NOTIF_STATUS_INDICATION with value TB640_ISDN_STATUS_IND_VALUE_BCHANNEL_STATE_CHANGE). This allows an application to monitor which timeslot can be use to make calls or receive calls. This command is available for all stack variants.

On the other hand, only four stacks variants (DMS, 5ESS, 4ESS and NI2) allows the host application to change the B-channel state through a command message (TB640_MSG_ID_ISDN_STATES_BCHANNEL_SET). The main reason is that only those four variants have implemented a primitive (not part of Q.931 standard) that allow a stack to inform its peer stack that a B-channel is about to change state.

Upon stack bring-up sequence, the user-application will receive a series of B-channel state change events. If the bring-up sequence worked correctly, every B-channel (timeslot) should have its state set to ‘in-service’. Note that the B-channel states (in-service, maintenance or out-of-service) is not related to the D-channel state (up or down). Even if the logical D-channel is down, all B-channels can still be ‘in-service’ meaning that the stacks will be able to issue calls once the D-channel is brought up.

8.2.8 Asynchronous issues

The ISDN stacks, the blade/host communication mechanism and the host application being asynchronous modules in the system, it is almost impossible to avoid race conditions. Such race conditions can be detected and be dealt with by the host application. The two most common race conditions are incoming/outgoing call collisions and disconnection collisions. In all cases, the application designer must always let the ISDN stack be the component that decides which call will win the conflict condition. The call scenarios illustrating these cases are presented in section 8.2.11.

An incoming/outgoing call collision is occurring when the host application tries to make an outgoing call on a B-channel that is, at the very same moment, being reserved by the ISDN stack for an incoming call. The host application won’t know that the B-channel is reserved until it receives the incoming call event. This event will be received even before the outgoing call request’s response. Therefore, the host application must assume that the stack has decided to let the incoming call allocate the B-channel and need to process the B-channel as an incoming call.

Disconnect collision can occur when both the ISDN stack and the host application tries to disconnect the same call at the same time. The host application first sends a disconnection request toward the ISDN stack. Immediately after sending the request, the application receives a disconnection notification because the stack had already disconnected the call. In that specific case, the disconnection request sent will fail because the call handle will have been freed by the ISDN stack upon sending the disconnection event. Therefore, the host application must assume that the stack has cleared the call before and must expect receiving a failure for the disconnection request.

8.2.9 Original vs extended ISDN API message

Over the last years, TelcoBridges' ISDN stack has been deployed in multiple sites and has interoperated with even more different ISDN switches. Although the vast majority of those switches conform to general specifications, some private networks are using local customizations (i.e. information elements relevant only to their network) to achieve some specific features. The original version of the Telcobridges' ISDN API wasn't flexible enough to allow the host application to send and/or receive custom information elements or to even receive the complete content of a primitive. In fact, the ISDN stack, once the primitive validation was over, was extracting the major information elements (i.e. CDN, CGN, cause, etc) and sending them toward the host application. This type of API fits very well for applications that don't need to decipher any of the local information elements since those custom elements cannot influence the Q.931 state machine of a call. On the other hand, this type of API cannot be extended easily to support every possible IEs (as they are locally defined to a specific network). Therefore, an extension API has been designed to support any possible types of IE in both transmission and reception directions.Since not every customer requires to process custom information elements, a key point to the addition of the extended API was that the original API would be working without any modifications to an existing application. The new API is only activated when the TB640_ISDN_STACK_OPTIONS_USE_EXTENDED_ISDN_API option is used upon ISDN stack allocation. If not activated, the ISDN stack behaves exactly as the original API did.

The major difference between the original and extended API is how IE content is passed within every request and notification. Using the extended API, every IE is contained into a 'raw' IE buffer within each. Thus, the host application can now fill/parse any outgoing/incoming Q.931 primitive with any possible information element into/from a raw IE buffer. It is no longer limited to specific known information element. The IEs are formatted as per the ITU Q.931 specification (section 4.5.1). Helping functions have also been created to help the host application to properly format (or parse) these IEs. The ISDN stack will validate the syntax of the IEs but will simply ignore any non-relevant IEs in most situations. There are still cases where certain IEs are invalid or incompatible for a specific variant. For those cases, the ISDN stack will try to map the outgoing potentially invalid IEs to valid information elements. When this is not possible, the stack will process the primitive as if they were invalid.

Section 8.2.9.1 explains how to send request and receive notification using the extended ISDN API. Section 8.2.9.2 covers the usage of the original APIs. For documentation simplicity, all call scenarios from section 8.2.10 and 8.2.11 are expressed in terms of the extended API. API correspondence between the two APIs is summarized in Table 14, Table 15 and Table 16.

Table 14 – Common messages used in both original and extended ISDN API

Common ISDN API messages	Description
TB640_MSG_ID_ISDN_OP_ALLOC	Allocates an ISDN stack instance and attach it to one/many trunk(s)
TB640_MSG_ID_ISDN_OP_FREE	Frees an allocated ISDN stack instance and detach it from one/many trunk(s)
TB640_MSG_ID_ISDN_OP_GET_PARAMS	Retrieves the configuration parameters from an ISDN stack instance
TB640_MSG_ID_ISDN_OP_GET_LIST	Retrieves the list of all allocated ISDN stack handles
TB640_MSG_ID_ISDN_OP_CALL_GET_PARAMS	Retrieves the parameters of a call associated to an ISDN stack instance
TB640_MSG_ID_ISDN_OP_CALL_GET_LIST	Retrieves the list of all calls associated to an ISDN stack instance
TB640_MSG_ID_ISDN_OP_GET_NVPARAMS	Retrieves the global system parameters (stored in non-volatile memory) associated with all ISDN stack instances.
TB640_MSG_ID_ISDN_OP_SET_NVPARAMS	Modifies the global system parameters (stored in non-volatile memory) associated with all ISDN stack instances.
TB640_MSG_ID_ISDN_CMD_RESTART_CHANNEL_REQUEST	Requests a state restart (per Q.931 specifications) for one/many B-channel(s) associated with an ISDN stack instance
TB640_MSG_ID_ISDN_CMD_WAKEUP_REQUEST	Requests a D-channel synchronization with peer ISDN stack
TB640_MSG_ID_ISDN_CMD_RESET_CHANNEL_REQUEST	Requests a local state reset for one/many B-channel(s) associated with an ISDN stack instance
TB640_MSG_ID_ISDN_STATES_GET	Retrieves the state of an allocated ISDN stack instance regarding the D-channel and physical trunk status
TB640_MSG_ID_ISDN_STATES_CALL_GET_STATE	Retrieves the current state of a call associated to a ISDN stack instance
TB640_MSG_ID_ISDN_STATES_BCHANNEL_SET	Modifies the state of a specific B-channel associated to an allocated ISDN stack instance
TB640_MSG_ID_ISDN_STATES_BCHANNEL_GET	Retrieves the current state of a specific B-channel associated to an allocated ISDN stack instance
TB640_MSG_ID_ISDN_NOTIF_STACK_AVAILABILITY	Notification event telling about an ISDN stack's level of availability to process ISDN calls.

TB640_MSG_ID_ISDN_NOTIF_STATUS_INDICATION	Notification event telling about a stack or call specific state change
TB640_ISDN_STATUS_IND_VALUE_PHYSICAL_LINE_ACTIVATED	Trunk is in service (no LOS, no AIS, no RAI)
TB640_ISDN_STATUS_IND_VALUE_PHYSICAL_LINE_DEACTIVATED	Trunk is out-of-service (LOS and/or AIS and/or RAI)
TB640_ISDN_STATUS_IND_VALUE_LINE_READY	D-channel is in service
TB640_ISDN_STATUS_IND_VALUE_LINE_NOT_READY	D-channel is out-of-service
TB640_ISDN_STATUS_IND_VALUE_RESTART_ACK	Result of a previous RESTART request on one/many B-channel(s)
TB640_ISDN_STATUS_IND_VALUE_NO_CHANNEL_ERROR	'No more channel' error notification upon outgoing or incoming call event
TB640_ISDN_STATUS_IND_VALUE_BCHANNEL_STATE_CHANGE	Indication that a B-channel has just changed its state
TB640_ISDN_STATUS_IND_VALUE_BCHANNEL_STATE_CHANGE_ACK	Confirmation of a previous B-channel state change request

Table 15 – Matching between original and extended ISDN messages

Original ISDN API	Extended ISDN API
TB640_MSG_ID_ISDN_CMD_CONNECT_REQUEST	TB640_MSG_ID_ISDN_CMD_INITIATE_CALL
TB640_MSG_ID_ISDN_CMD_CONNECT_RESPONSE	TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_CONNECT_RESPONSE)
TB640_MSG_ID_ISDN_CMD_MORE_INFO_REQUEST	TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_MORE_INFO)
TB640_MSG_ID_ISDN_CMD_CONNECT_ACK_REQUEST	TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_CONNECT_ACK)
TB640_MSG_ID_ISDN_CMD_DISCONNECT_REQUEST	TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_DISCONNECT)
TB640_MSG_ID_ISDN_CMD_KEYPAD_REQUEST	TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_KEYPAD)
TB640_MSG_ID_ISDN_CMD_ALERT_REQUEST	TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_ALERT)
TB640_MSG_ID_ISDN_CMD_CALL_PROCEEDING_REQUEST	TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_CALL_PROCEEDING)
TB640_MSG_ID_ISDN_CMD_PROGRESS_REQUEST	TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_PROGRESS)
TB640_MSG_ID_ISDN_NOTIF_CONNECT_INDICATION TB640_MSG_ID_ISDN_NOTIF_CALL_PRESENT_INDICATION	TB640_MSG_ID_ISDN_NOTIF_INCOMING_CALL
TB640_MSG_ID_ISDN_NOTIF_CONNECT_CONFIRM	TB640_MSG_ID_ISDN_NOTIF_STATE_CHANGE_EVENT (TB640_ISDNMGR_NOTIF_TYPE_CONNECT_CONFIRM)
TB640_MSG_ID_ISDN_NOTIF_DISCONNECT_INDICATION	TB640_MSG_ID_ISDN_NOTIF_STATE_CHANGE_EVENT (TB640_ISDNMGR_NOTIF_TYPE_DISCONNECT)
TB640_MSG_ID_ISDN_NOTIF_DISCONNECT_CONFIRM	TB640_MSG_ID_ISDN_NOTIF_STATE_CHANGE_EVENT (TB640_ISDNMGR_NOTIF_TYPE_DISCONNECT_CONFIRM)
TB640_MSG_ID_ISDN_NOTIF_KEYPAD_INDICATION	TB640_MSG_ID_ISDN_NOTIF_STATE_CHANGE_EVENT (TB640_ISDNMGR_NOTIF_TYPE_KEYPAD)
TB640_MSG_ID_ISDN_NOTIF_CALL_PROCEEDING_INDICATION	TB640_MSG_ID_ISDN_NOTIF_STATE_CHANGE_EVENT (TB640_ISDNMGR_NOTIF_TYPE_CALL_PROCEEDING)
TB640_MSG_ID_ISDN_NOTIF_ALERT_INDICATION	TB640_MSG_ID_ISDN_NOTIF_STATE_CHANGE_EVENT (TB640_ISDNMGR_NOTIF_TYPE_ALERT)
TB640_MSG_ID_ISDN_NOTIF_PROGRESS_INDICATION	TB640_MSG_ID_ISDN_NOTIF_STATE_CHANGE_EVENT (TB640_ISDNMGR_NOTIF_TYPE_PROGRESS)
TB640_MSG_ID_ISDN_NOTIF_STATUS_INDICATION	TB640_MSG_ID_ISDN_NOTIF_STATE_CHANGE_EVENT
TB640_ISDN_STATUS_IND_VALUE_MISC_INFO	IE is received in IE raw buffer in the associated event (no separate event)
TB640_ISDN_STATUS_IND_VALUE_CAUSE_INFO_IE	IE is received in IE raw buffer in the associated event (no separate event)
TB640_ISDN_STATUS_IND_VALUE_DISPLAY_INFO_IE	IE is received in IE raw buffer in the associated event (no separate event)
TB640_ISDN_STATUS_IND_VALUE_SIGNAL_INFO_IE,	IE is received in IE raw buffer in the associated event (no separate event)
TB640_ISDN_STATUS_IND_VALUE_PROGRESS_INDICATOR_IE	IE is received in IE raw buffer in the associated event (no separate event)
TB640_ISDN_STATUS_IND_VALUE_FEATURE_INDICATION_IE	IE is received in IE raw buffer in the associated event (no separate event)
TB640_ISDN_STATUS_IND_VALUE_CONNECT_RECEIVED	TB640_ISDNMGR_NOTIF_TYPE_NETWORK_CONNECT
TB640_ISDN_STATUS_IND_VALUE_DISCONNECT_RECEIVED	TB640_ISDNMGR_NOTIF_TYPE_NETWORK_DISCONNECT
TB640_ISDN_STATUS_IND_VALUE_NOTIFICATION_INDICATOR_IE	IE is received in IE raw buffer in the associated event (no separate event)
TB640_ISDN_STATUS_IND_VALUE_US_NI_INFO_REQUEST_IE	IE is received in IE raw buffer in the associated event (no separate event)
TB640_ISDN_STATUS_IND_VALUE_SENDING_COMPLETED	IE is received in IE raw buffer in the associated event (no separate event)
TB640_ISDN_STATUS_IND_VALUE_MORE_INFO	TB640_ISDNMGR_NOTIF_TYPE_SETUP_ACK

Table 16 - Messages used in Extended ISDN API

Extended ISDN API messages	Description
TB640_MSG_ID_ISDN_CMD_INITIATE_CALL	Requests an outgoing call establishment through a SETUP primitive
TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_CONNECT_RESPONSE)	Requests a CONNECT primitive to be sent
TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_MORE_INFO)	Requests a SETUP_ACK primitive to be sent (in overlap mode).
TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_CONNECT_ACK)	Requests a CONNECT_ACK primitive to be sent
TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_DISCONNECT)	Requests a DISC primitive to be sent
TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_KEYPAD)	Requests an INFO primitive (with a 'CDN' or 'Keypad' IE) to be sent
TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_ALERT)	Requests a ALERT primitive to be sent
TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_CALL_PROCEEDING)	Requests a CALL_PROCEEDING primitive to be sent
TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_PROGRESS)	Requests a PROGRESS primitive to be sent
TB640_MSG_ID_ISDN_NOTIF_INCOMING_CALL	Notification event telling about a received incoming call through a SETUP primitive.
TB640_MSG_ID_ISDN_NOTIF_STATE_CHANGE_EVENT (TB640_ISDNMGR_NOTIF_TYPE_CONNECT_CONFIRM)	Notification event telling about a received CONNECT_ACK primitive
TB640_MSG_ID_ISDN_NOTIF_STATE_CHANGE_EVENT (TB640_ISDNMGR_NOTIF_TYPE_DISCONNECT)	Notification event telling about a received DISC primitive
TB640_MSG_ID_ISDN_NOTIF_STATE_CHANGE_EVENT (TB640_ISDNMGR_NOTIF_TYPE_DISCONNECT_CONFIRM)	Notification event telling about a received RELEASE primitive
TB640_MSG_ID_ISDN_NOTIF_STATE_CHANGE_EVENT (TB640_ISDNMGR_NOTIF_TYPE_KEYPAD)	Notification event telling about a received INFO primitive (with a 'CDN' or 'Keypad' IE)
TB640_MSG_ID_ISDN_NOTIF_STATE_CHANGE_EVENT (TB640_ISDNMGR_NOTIF_TYPE_CALL_PROCEEDING)	Notification event telling about a received CALL_PROCEEDING primitive
TB640_MSG_ID_ISDN_NOTIF_STATE_CHANGE_EVENT (TB640_ISDNMGR_NOTIF_TYPE_ALERT)	Notification event telling about a received ALERT primitive
TB640_MSG_ID_ISDN_NOTIF_STATE_CHANGE_EVENT (TB640_ISDNMGR_NOTIF_TYPE_PROGRESS)	Notification event telling about a received PROGRESS primitive
TB640_MSG_ID_ISDN_NOTIF_STATE_CHANGE_EVENT (TB640_ISDNMGR_NOTIF_TYPE_NETWORK_CONNECT)	Notification event telling about a received CONNECT primitive by a network-side ISDN stack instance
TB640_MSG_ID_ISDN_NOTIF_STATE_CHANGE_EVENT (TB640_ISDNMGR_NOTIF_TYPE_NETWORK_DISCONNECT)	Notification event telling about a received DISC primitive by a network-side ISDN stack instance
TB640_MSG_ID_ISDN_NOTIF_STATE_CHANGE_EVENT (TB640_ISDNMGR_NOTIF_TYPE_SETUP_ACK)	Notification event telling about a received SETUP_ACK primitive

8.2.9.1 Extended ISDN API message

The extended ISDN API mode is activated when the option TB640_ISDN_STACK_OPTIONS_USE_EXTENDED_ISDN_API is configured during the stack instance allocation.

☞ When the extended ISDN API is used, the stack will refuse any requests made with the original API messages. Also, all notifications will be sent using the extended API events messages.

As explained in section 8.2.9, this mode primarily uses a byte buffer pass any information elements content from and to an ISDN stack instance. Within this byte buffer, all IEs are formatted as specified in the Q.931 specification (section 4.5.1). For all ISDN messages coming from the host application, the byte buffer will be verified for syntax correctness and passed to the ISDN stack. Failure to comply with formatting rules of the specification will result in the call refusal. Only relevant information elements will be used by the ISDN stack depending on the message. Some information elements are mandatory depending on the request type and some others are prohibited because it would interfere with the ISDN state machine. Table 17 lists the different IE restrictions depending on the request type. Other information elements (completely ignored by the ISDN stack) will be inserted transparently to the Q.931 data flow going out from the blade. Furthermore, if the ISDN stack generates internally an IE (e.g. bearer capabilities) that is also contained in the byte buffer, the outgoing Q.931 message will contain the information element from the byte buffer. This gives the user application almost complete control over the information elements. This also has the drawback that the application can insert or replace IEs that will make the remote switch to refuse the incoming call because of incompatibility issues. Section 0 explains in detail how to fill specific information element into ISDN messages.

☞ Only insert information elements that are allowed into the targeted ISDN network. Otherwise, the remote switch is likely to refuse or drop the faulty calls. In case of doubt, only use the mandatory information elements specified in Table 17. The ISDN stack will ensure to respect the switch variant restrictions.

As an example, the CMD_INITIATE_CALL message is expected to contain at least the 'called party number' information element. If that specific IE is missing from the byte buffer, the outgoing call will simply be refused. On the other hand, the byte buffer could contain any other IE, including IE from a different codeset (refer to section 0 or to Q.931 section 4.5.2 to learn about ISDN codesets). The ISDN stack will merge its generated information elements with the IEs contained in the byte buffer before sending the SETUP primitive to the network.

In the receive direction, the ISDN stack will include all IEs from the received primitive into the byte buffer included in the notification. Therefore, the user application will actually receive a raw buffer containing all information elements at every steps of a call state. The host application needs to parse through the list of information elements to retrieve the ones that are relevant to the application (e.g. called party number, calling party number, etc). Section 8.2.9.1.2 explains in detail how to search for specific information elements.

Table 17 – Allowed, typical and prohibited IEs depending on request type

Request type	Mandatory IE(s)	Typical IE(s)	Prohibited IE(s)
TB640_MSG_ID_ISDN_CMD_INITIATE_CALL	Called party number	Calling party nunber Bearer capabilities High layer compatibility Low layer compatibility User to user Redirecting Sending complete	Channel identification*
TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_CONNECT_RESPONSE)	None	User to user	Channel identification
TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_MORE_INFO)	None	None	Channel identification
TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_CONNECT_ACK)	None	None	Channel identification
TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_DISCONNECT)	Cause	User to user	Channel identification
TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_KEYPAD)	Keypad facility	Sending complete	Channel identification
TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_ALERT)	None	Progress indicator	Channel identification
TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_CALL_PROCEEDING)	None	None	Channel identification
TB640_MSG_ID_ISDN_CMD_STATE_CHANGE_REQUEST (TB640_ISDNMGR_REQUEST_TYPE_PROGRESS)	Progress indicator	None	Channel identification

* Channel is controlled by hTrunkRes rather than this IE.

8.2.9.1.1 How to fill IE buffer in ISDN request

As mentioned before, every IE within a byte buffer must be formatted according to rules from Q.931 section 4.5.1. This allows the host application to insert IEs that are totally unknown to an ISDN stack as long as it respects those formatting rules. To ease the programming effort, macros are available to the host application to fill the buffers properly. These macros format the IE byte information according to the member of a local structure that is more convenient to work with inside a C/C++ piece of code. Before looking at the helping macros, we need to describe the raw format of a Q.931 IE.

8.2.9.1.1.1 Native format of an IE

There are two types of information elements: single octet and variable-length IEs. The former, as its name implies, is defined by a single octet and does not contain any payload. Since the first octet of an IE is always its identification code, these single-octet IE are totally defined by their ID. The coding format specifies that the most significant bit of an IE identification code indicates if it is a single octet IE (bit8 set to 1) or a variable-length IE (bit8 set to 0). As an example, the single-octet IE 'sending complete' is totally defined its IE identification code (0xA1) as shown below:

8	7	6	5	4	3	2	1	Octet
1	0x21							1

The later type of information element always contains a minimum of 2 bytes. As shown below, the first byte is the IE identification code and is followed by the payload length (which can be zero).

8	7	6	5	4	3	2	1	Octet
0	Information element identifier							1
Length of contents of information element (octets)								2
Contents of information element								3 etc.

The only restriction is that variable-length information elements need to be place in ascending numerical order so that an ISDN stack can easily determine if some mandatory IEs are missing from a primitive without having to scan through the whole IE buffer. This rule from the Q.931 spec also applies for the host application when filling the byte buffer with information element. It is important to note that a single-octet information element can be found anywhere in a byte buffer regardless of it identifier code since the rule only applies to variable-length IEs.

☞ IE within the byte buffer must be written in ascending order based on their information element IE as specified by Q.931 section 4.5.1. This rule applies only for variable-length IE. Single octet IE can be placed anywhere in the byte buffer.

8.2.9.1.1.2 Working with different codesets

Due to the encoding of information element ID field, there is only a limited number of IE identifier that can exist. To enable IE extension, the concept of 'codeset' is defined by the Q.931 specifications (section 4.5.2). Codesets can be viewed as categories of information elements. By default, all information elements described in the Q.931 specifications and this document are from codeset 0. There are 5 different codesets that can be used:

Codeset 0 is the default (and mandatory) codeset and supported IE are defined in Q.931 section 4.5
Codeset 4 is reserved for use by ISO/IEC Standards.
Codeset 5 is reserved for information elements reserved for national use.
Codeset 6 is reserved for information elements specific to the local network (either public or private).
Codeset 7 is reserved for user-specific information elements.

An information element identifier has a specific meaning only into a specific codeset. This means that IE identifier 0xA1 (i.e. sending complete) from codeset 0 does not have the same meaning in codeset 6. It is possible, within the same primitive, to have IEs from different codesets. To do so, a single octet IE exists to switch from one codeset to the other. The base rule is that it is only possible to switch to an higher codeset (i.e. from 0 to 6). The shift can be temporary (i.e. only affect the next IE) or permanent (the rest of the buffer is encoded in the new codeset). The temporary shift is referred to as "non-locking codeset shift". The format of the codeset shift IE is shown below:

8	7	6	5	4	3	2	1	Octet
	Shift identifier							
1	0	0	1	0	New codeset identification			1

↑
"0" in this position
indicates locking shift

Since the ISDN stack has only interest in codeset 0 IEs (because they may be state affecting), all IEs from different codesets are ignored by the stack and sent to the user application (or sent to the network transparently). The extended

ISDN API allows the user to fill or parse these IEs and extract meaningful information. For example, a local ISDN network could use a codeset 6 IE to pass the type of phone terminal that is used when the call is established. This information, not useful for the ISDN stack, could be used to change billing information of the call by the application. Below is an example of a SETUP message containing an IE from codeset 6 (local network):

```
// Time:14:30:10.015#000           -----> length=67 SAPI=0, CR=0, TEI=0 [I] PF= 0 NR= 78 NS= 17
Call reference -> 1
[0x05] 00000101  SETUP
[0x04] 00000100  Bearer capability ->
[0x03] 00000011  IE Length = 3
[0x80] -00-----  Coding Standard: ITU-T
[0x80] ---00000  Information transfer capability: Speech
[0x90] -00-----  Transfer mode: Circuit mode
[0x90] ---10000  Information transfer rate: 64kbps
[0xA3] ---00011  User information layer 1: G.711 Alaw
[0x18] 00011000  Channel identification->
[0x03] 00000011  IE Length = 3
[0xA9] -0------  Implicitly identified
[0xA9] --1-----  Primary rate interface
[0xA9] ----1---  Indicated channel is exclusive
[0xA9] -----0--  Channel identified is not the D channel
[0xA9] ------01  Information channel selection: B1 channel
[0x83] -00-----  Coding Standard: ITU-T
[0x83] ---0----  Channel is indicated by the number
[0x83] ------11  Channel type/map element type: B channel units
[0x81] -0000001  Channel number: 1
[0x6C] 01101100  Calling party number->
[0x06] 00000110  IE Length = 6
[0x21] -010----  Type of number: National number
[0x21] ----0001  Numbering plan identification: ISDN/telephony numbering plan
[0x80] -00-----  Presentation indicator: Presentation allowed
[0x80] ------00  Screening indicator: User-provided, not screened Number: 5555
[0x70] 01110000  Called party number->
[0x01] 00000001  IE Length = 1
[0xA1] -010----  Type of number: National number
[0xA1] ----0001  Numbering plan identification: ISDN/telephony numbering plan Number:
[0x9E] 10011110  Non-locking Shift to codeset 6 -> IE specific to the local network
[0x40] 01000000  IE from codeset 6 ->
[0x04] 00000100  IE Length = 4
[0x01] 00000001
[0x02] 00000010
[0x03] 00000011
[0x04] 00000100
```

Therefore, when filling or parsing a raw IE buffer, the host application must be careful and track in which codeset the filling/parsing process is currently in as an identical IE id may have a different meaning depending on the current active codeset.

8.2.9.1.1.3 Helping macros

To help create human-readable piece of code, helping macros have been designed to ease the IE formatting and inserting. The helping macros (include the source code) are available in the API header file *tb640_isdnmgr_ie.*h from the TB640 package. Currently available macros are summarized in Table 18.

Table 18 - IE formatting helping macros

Information element	Helping macros	C structure associated with the IE
Calling party number	TB640_ISDN_WRITE_CGN	TB640_ISDNMGR_IE_CGN
Called party number	TB640_ISDN_WRITE_CDN	TB640_ISDNMGR_IE_CDN
Keypad facility	TB640_ISDN_WRITE_KEYPAD_FACILITY	TB640_ISDNMGR_IE_KEYPAD_FACILITY
Progress indicator	TB640_ISDN_WRITE_PROGRESS_INDICATOR	TB640_ISDNMGR_IE_PROGRESS_INDICATOR
User to user	TB640_ISDN_WRITE_USER_TO_USER	TB640_ISDNMGR_IE_USER_TO_USER
Cause	TB640_ISDN_WRITE_CAUSE	TB640_ISDNMGR_IE_CAUSE
Redirecting	TB640_ISDN_WRITE_REDIRECTING	TB640_ISDNMGR_IE_REDIRECTING
Not specific to a particular IE	TB640_ISDNMGR_WRITE_SINGLE_OCTET_IE	IE Id provided by the user
Not specific to a particular IE	TB640_ISDNMGR_WRITE_VAR_LENGTH_IE	Any buffer provided by the user
Not specific to a particular IE	TB640_ISDNMGR_COPY_IE	Raw IE buffer provided by the user and its

		associated read offset pointers.

When filling a byte buffer with information elements, the host application can make use of these macros to convert IE content from a C structure into its Q.931 format. The macros also manages the 'write pointer' into the byte buffer so that it is easy to write multiple information elements one after the other. Below is an example of code using the macros to fill the TB640_MSG_ID_ISDN_CMD_INITIATE_CALL message. Looking at the sample code 'isdnext' also gives other sets of examples.

```
/* Fill with IEs */
pReq->un32IeBufferSize = 0;

/* Let's fill the IEs IN THE NUMERICAL ORDER to follow Q.931 section 4.5.1 encoding rules */

/*************************************************
 *        Format 'Calling party number' IE        *
 *************************************************/

Cgn.TypeOfNumber = TB640_ISDNMGR_TYPE_OF_NUMBER_NATIONAL;
Cgn.NumberingPlan = TB640_ISDNMGR_NUMBERING_PLAN_ISDN;

Cgn.PresentationAndScreening.fPresentationAndScreeningPresent = TBX_TRUE;
Cgn.PresentationAndScreening.Presentation = TB640_ISDNMGR_PRESENTATION_INDICATOR_ALLOWED;
Cgn.PresentationAndScreening.Screening = TB640_ISDNMGR_SCREENING_INDICATOR_USER_PROVIDED_NOT_SCREENED;

Cgn.un8NbIA5Characters = strlen(in_pCallContext->aun8OutbandCallingAddress);
strcpy (Cgn.aIA5Chars, in_pCallContext->aun8OutbandCallingAddress);

fResultSuccess = TB640_ISDN_WRITE_CGN (pReq->aun8IeBuffer, &pReq->un32IeBufferSize,
                                      TB640_ISDNMGR_MAXIMUM_IE_BUFFER_LEN, &Cgn);
if (!fResultSuccess)
{
        TBX_EXIT_ERROR(result, 0, "Unable to write CGN IE");
}

/*************************************************
 *        Format 'Called party number' IE         *
 *************************************************/

Cdn.TypeOfNumber = TB640_ISDNMGR_TYPE_OF_NUMBER_NATIONAL;
Cdn.NumberingPlan = TB640_ISDNMGR_NUMBERING_PLAN_ISDN;

Cdn.un8NbIA5Characters = strlen(in_pCallContext->aun8OutbandCalledAddress);
strcpy (Cdn.aIA5Chars, in_pCallContext->aun8OutbandCalledAddress);

fResultSuccess =  TB640_ISDN_WRITE_CDN (pReq->aun8IeBuffer, &pReq->un32IeBufferSize,
                                      TB640_ISDNMGR_MAXIMUM_IE_BUFFER_LEN, &Cdn);
if (!fResultSuccess)
{
        TBX_EXIT_ERROR(result, 0, "Unable to write CDN IE");
}

/*************************************************
 *             Format 'Redirecting' IE            *
 *************************************************/

if (strlen(in_pCallContext->aun8Redirecting) > 0)
{
        Redirecting.TypeOfNumber = TB640_ISDNMGR_TYPE_OF_NUMBER_NATIONAL;
        Redirecting.NumberingPlan = TB640_ISDNMGR_NUMBERING_PLAN_ISDN;

        Redirecting.PresentationAndScreening.fPresentationAndScreeningPresent = TBX_TRUE;
        Redirecting.PresentationAndScreening.Presentation = TB640_ISDNMGR_PRESENTATION_INDICATOR_ALLOWED;
        Redirecting.PresentationAndScreening.Screening =
                                      TB640_ISDNMGR_SCREENING_INDICATOR_USER_PROVIDED_NOT_SCREENED;
        Redirecting.Reason.fReasonPresent = TBX_TRUE;
        Redirecting.Reason.RedirectingReason = TB640_ISDNMGR_REDIRECTING_REASON_CALL_DEFLECTION;

        Redirecting.un8NbIA5Characters = strlen(in_pCallContext->aun8Redirecting);
        strcpy (Redirecting.aIA5Chars, in_pCallContext->aun8Redirecting);

        fResultSuccess = TB640_ISDN_WRITE_REDIRECTING (pReq->aun8IeBuffer, &pReq->un32IeBufferSize,
                                      TB640_ISDNMGR_MAXIMUM_IE_BUFFER_LEN, &Redirecting);
        if (!fResultSuccess)
        {
                TBX_EXIT_ERROR(result, 0, "Unable to write Redirecting IE");
        }
}
```

8.2.9.1.1.4 How to fill new information elements

The formatting macros are only help function to ease the formatting of IE into a raw byte buffer. An application designer could fill the buffer himself with any IE he desires as long as the format respects Q.931 section 4.5.1. He could also extend the helping macros of file *tb640_isdnmrg_ie.h* to support new IEs that are currently listed there. There is no change required to the TB640 firmware in order to support these new IEs.

8.2.9.1.2 How to parse IE buffer in ISDN notification

As mentioned before, every message received from the ISDN stack will contain a buffer with raw IE information (even IEs that were not used by the ISDN stack). Based on the information contained in those IEs, the application can make decisions regarding a specific call (e.g. called party number, user-to-user information, etc). To ease the programming effort, macros are available to the host application to scan the buffers efficiently and find relevant IE for an application. These macros convert the Q.931 raw IE byte into a C structure that is more convenient to work with inside a C/C++ piece of code. The base principle is that any unknown IE should be ignored by the receiving application. This will ensure that this application can work in different ISDN networks.

8.2.9.1.2.1 Helping macros

To help create human-readable piece of code, helping macros have been designed to ease the IE scanning from a received raw IE buffer. The helping macros (include the source code) are available in the API header file *tb640_isdnmgr_ie.*h from the TB640 package. Currently available parsing macros are summarized in Table 19.

Table 19 - IE parsing helping macros

Information element	Helping macros	C structure associated with the IE
Calling party number	TB640_ISDN_READ_CGN	TB640_ISDNMGR_IE_CGN
Called party number	TB640_ISDN_READ_CDN	TB640_ISDNMGR_IE_CDN
Keypad facility	TB640_ISDN_READ_KEYPAD_FACILITY	TB640_ISDNMGR_IE_KEYPAD_FACILITY
Progress indicator	TB640_ISDN_READ_PROGRESS_INDICATOR	TB640_ISDNMGR_IE_PROGRESS_INDICATOR
User to user	TB640_ISDN_READ_USER_TO_USER	TB640_ISDNMGR_IE_USER_TO_USER
Cause	TB640_ISDN_READ_CAUSE	TB640_ISDNMGR_IE_CAUSE
Redirecting	TB640_ISDN_READ_REDIRECTING	TB640_ISDNMGR_IE_REDIRECTING
Not specific to a particular IE	TB640_ISDNMGR_READ_SINGLE_OCTET_IE	Read IE is returned by argument (single octet)
Not specific to a particular IE	TB640_ISDNMGR_READ_VAR_LENGTH_IE	IE content returned in a buffer provided by user
Not specific to a particular IE	TB640_ISDNMGR_GET_IE_ID	Read IE Id is returned by argument
Not specific to a particular IE	TB640_ISDNMGR_GET_IE_LENGTH	Read IE length field is returned by argument
Not specific to a particular IE	TB640_ISDNMGR_SKIP_IE	Read offset pointer of a raw buffer is modified to point to the next IE
Not specific to a particular IE	TB640_ISDNMGR_SEARCH_IE	Read offset pointer of a raw buffer is modified to point to the desired IE if it is contained in the buffer

When parsing a byte buffer for information elements, the host application can make use of these macros to convert from Q.931 format to a C structure usable in C/C++ code. The macros also manages the ‘read pointer’ into the byte buffer so that it is easy to read or search multiple information elements from the same buffer. Below is an example of code using the macros to parse the TB640_MSG_ID_ISDN_NOTIF_INCOMING_CALL message. Looking at the sample code ‘isdnext’ also gives other sets of examples.

```
/******************************************
 * Retrieve mandatory called party number *
 ******************************************/
un32Offset = 0;
fResult = TB640_ISDNMGR_SEARCH_IE (pIsdnIncomingCall->aun8IeBuffer, &un32Offset, pIsdnIncomingCall->un32IeBufferSize,
                                   TB640_ISDNMGR_IE_ID_CALLED_PARTY_NUMBER);
if (fResult != TBX_FALSE)
{
        fResult = TB640_ISDN_READ_CDN (pIsdnIncomingCall->aun8IeBuffer, un32Offset, pIsdnIncomingCall->un32IeBufferSize, &Cdn);
}
if (!fResult)
{
        TBX_EXIT_ERROR(TBX_RESULT_INVALID_PARAM, 0, "Unable to retrieve mandatory 'called party number' IE");
}

/* Copy the 'CDN' IE into our call context */
```

```
memcpy (pCallContext->aun8OutbandCalledAddress, Cdn.aIA5Chars, Cdn.un8NbIA5Characters);
pCallContext->aun8OutbandCalledAddress [Cdn.un8NbIA5Characters] = 0;

/*****************************************
 * Retrieve optional calling party number *
 *****************************************/
un32Offset = 0;
fResult = TB640_ISDNMGR_SEARCH_IE (pIsdnIncomingCall->aun8IeBuffer, &un32Offset, pIsdnIncomingCall->un32IeBufferSize,
                                   TB640_ISDNMGR_IE_ID_CALLING_PARTY_NUMBER);
if (fResult != TBX_FALSE)
{
        fResult = TB640_ISDN_READ_CGN (pIsdnIncomingCall->aun8IeBuffer, un32Offset, pIsdnIncomingCall->un32IeBufferSize, &Cgn);
        if (!fResult)
        {
                TBX_EXIT_ERROR(TBX_RESULT_INVALID_PARAM, 0, "Unable to read optional 'calling party number' IE");
        }

        /* Copy the 'CGN' IE into our call context */
        memcpy (pCallContext->aun8OutbandCallingAddress, Cgn.aIA5Chars, Cgn.un8NbIA5Characters);
        pCallContext->aun8OutbandCallingAddress [Cgn.un8NbIA5Characters] = 0;
}

/***************************************
 * Retrieve optional redirecting number *
 ***************************************/
un32Offset = 0;
fResult = TB640_ISDNMGR_SEARCH_IE (pIsdnIncomingCall->aun8IeBuffer, &un32Offset, pIsdnIncomingCall->un32IeBufferSize,
                                   TB640_ISDNMGR_IE_ID_REDIRECTING);
if (fResult != TBX_FALSE)
{
        fResult = TB640_ISDN_READ_REDIRECTING (pIsdnIncomingCall->aun8IeBuffer, un32Offset, pIsdnIncomingCall->un32IeBufferSize,
                                                   &Redirecting);
        if (!fResult)
        {
                TBX_EXIT_ERROR(TBX_RESULT_INVALID_PARAM, 0, "Unable to read optional 'calling party number' IE");
        }

        /* Copy the 'redirecting' IE into our call context */
        memcpy (pCallContext->aun8OutbandCallingAddress, Redirecting.aIA5Chars, Redirecting.un8NbIA5Characters);
        pCallContext->aun8OutbandCallingAddress [Redirecting.un8NbIA5Characters] = 0;
}
```

8.2.9.1.2.2 How to parse new information elements

The parsing macros are only help function to ease the scanning of IEs from a raw byte buffer. An application designer could scan the buffer himself to search for any IEs he want. The generic macros xx_GET_IE_ID, xx_GET_IE_LENGTH, xx_SKIP_IE and xx_SEARCH_IE can be used regardless of the IE being scanned for. Extracting the raw byte information and formatting it to a C structure can still be done using custom-made helping functions or macros. This means an application designer can extend the helping macros of file *tb640_isdnmrg_ie.h* to support new IEs. There is no change required to the TB640 firmware in order to parse these new IEs.

8.2.9.2 Original ISDN API message

The original ISDN API is the default API mode in which an ISDN stack will start if the host application didn't specified the TB640_ISDN_STACK_OPTIONS_USE_EXTENDED_ISDN_API options upon allocation. As explained in section 8.2.9, this mode primarily uses arrays of bytes to pass specific information elements content from and to an ISDN stack instance. Basically, it means that an ISDN message will have a field named 'called party number' within its structure if the CDN information elements can be sent or received. This method has the advantage to be clear on what IE can be processed with a specific primitive. On the other hand, it has the limitation of only processing known information elements. Furthermore, if a new IE needs to be supported, the API must be changed and TelcoBridges needs to provide a new software release. The extended ISDN API addresses those two last concerns and is explained in section 8.2.9.1.

☞ When the original ISDN API is used, the stack will refuse any requests made with the extended API messages. Also, all notifications will be sent using the original API events messages.

8.2.9.2.1 How to fill or parse Information Elements (IE)

Information elements that are included in an ISDN message need to be filled prior to sending the ISDN messages. This should follow the ISDN user-network interface standard (Q.931). The TelcoBridges API provides individual fields within a message for each supported IE and the size and content must be filled properly. The IE identifier does not need to be specified in the buffer. The length of the payload (not including the IE identifier and the length itself) must be specified in the xxxSize parameter. So filling the IE will start in the Q.931 structures Octet 3. Most of the IEs are presented here with some coding examples.

8.2.9.2.1.1 Bearer Capabilities

Refer to "Bearer capabilities" information element as described in Q.931 section 4.5.5.
If *un32BearerCapabilitiesSize* = 0, the ISDN stack will use default values for this switch variant. If you include this IE, you must specify the octets according to standard in the *aunBearerCapabilities* structure. Invalid entries might result in call refusal. Maximum size of *un32BearerCapabilitiesSize* for this IE is 12 octets.

8	7	6	5	4	3	2	1	Octet
ext. 1	Coding standard		Information transfer capability					3
ext. 1	Transfer mode		Information transfer rate					4
ext. 0/1	Layer 1 ident. 0	1	User information layer 1 protocol					5*

Code example:
```
/* Set structure size */
un32BearerCapabilitiesSize = 3;

/* Octet 3: Coding standard = ITU-T (00), Information Transfer Capability = speech (00000) */
aun8BearerCapabilities[0] = 0x80;

/* Octet 4: Transfer Mode = circuit mode (00), Information Transfer rate = 64kbps (10000)*/
aun8BearerCapabilities[1] = 0x90;

/* Octet 5: ext (1), Layer 1 ident (01), User Information layer 1 protocol = G.711 ulaw (00010)*/
aun8BearerCapabilities[2] = 0xA2;
```

8.2.9.2.1.2 Outband called address

Refer to "Called party number" information element as described in Q.931 section 4.5.8.
Three parameters are provided for the Outband Called Address. *un32OutbandCalledAddressPrefixLength* indicates the length of the prefix bytes (before the actual number) in the IE included in the *aun8OutbandCalledAddress* buffer. If this value is zero, the default values will be used and the buffer must only contains the digits. If this value is set to 1, the first byte of the *aun8OutbandCalledAddress* will be the following structure:

8	7	6	5	4	3	2	1	Octet
ext. 1	Type of number			Numbering plan identification				3
0	Number digits (IA5 characters)							4*

un32OutbandCalledAddressSize is the size of the data within *aun32OutbandCalledAddress* (excluding *un32OutbandCalledAddressPrefixLength*)

Code example:
```
#define PREFIX_LENGTH 1
```

```
/* Use custom prefix */
un32OutbandCalledAddressPrefixLength = PREFIX_LENGTH;

/* Set structure size */
un32OutbandCalledAddressSize = strlen (szCalledPhoneNb) + PREFIX_LENGTH;

If (un32OutbandCalledAddressPrefixLength == PREFIX_LENGTH)
{
        /* Octet 3: Type of number = National, Numbering plan ID = ISDN (E.164) */
        aun8OutbandCalledAddress[0] = 0xA1;

        /* Octet 4: Copy content of IA5 characters */
        strcpy (&aun8OutbandCalledAddress[PREFIX_LENGTH], szCalledPhoneNb);
}
else /* un32OutbandCalledAddressPrefixLength== 0 */
{
        /* Use default stack values for Type of Number and Numbering plan ID */

        /* Octet 4: Copy content of IA5 characters */
        strcpy (aun8OutbandCalledAddress, szCalledPhoneNb);
}
```

8.2.9.2.1.3 Outband called subaddress

Refer to "Called party sub address" information element as described in Q.931 section 4.5.9.
Fill only the subaddress information (octets 4+) in the *aun8OutbandCalledSubaddress* structure. Maximum size for *un32OutbandCalledSubaddressSize* is 20 octets.

8	7	6	5	4	3	2	1	Octet
Subaddress information								4 etc.

Code example:

```
/* Octet 4: Copy content of IA5 characters */
strcpy (&aun8OutbandCalledSubaddress[0], szSubAddressNb);
```

8.2.9.2.1.4 Outband calling address

Refer to "Calling party number" information element as described in Q.931 section 4.5.10.
Three parameters are provided for the Outband Calling Address. *un32OutbandCallingAddressPrefixLength* indicates the length of the prefix bytes (before the actual number) in the IE included in the *aun8OutbandCallingAddress* buffer. If this value is zero, the default values will be used and the buffer must only contains the digits. If this value is set to 1 or 2, the first bytes of the *aun8OutbandCallingAddress* will be the following structure:

8	7	6	5	4	3	2	1	Octet
ext. 0/1	Type of number			Numbering plan identification				3
ext. 1	Presentation indicator		0	0	0	Screening indicator		3a*
0	Number digits (IA5 characters)							4*

Bit 8 (Ext) of octet 3 must be set to 0 if octet 3a is present

un32OutbandCallingAddressSize is the size of the data within *aun32OutbandCallingAddress* (excluding *un32OutbandCallingAddressPrefixLength*)

Code example:

```
#define PREFIX_LENGTH 2

/* Set structure size */
un32OutbandCallingAddressSize = strlen (szCallingPhoneNb) + PREFIX_LENGTH;

if( fUseCustomPrefix == TBX_TRUE )
{
        /* Use custom prefix */
        un32OutbandCallingAddressPrefixLength = PREFIX_LENGTH;

        /* Octet 3: Ext (0), Type of number = National (010), Numbering plan ID = ISDN (E.164) (0001) */
        aun8OutbandCallingAddress[0] = 0x21;

        /* Octet 3a: Ext (1), Presentation Ind. = restricted (01), (000), Screening Ind. = Network provided (11) */
        aun8OutbandCallingAddress[1] = 0xA3;

        /* Octet 4: Copy content of IA5 characters */
        strcpy (&aun8OutbandCallingAddress[PREFIX_LENGTH], szCallingPhoneNb);
}
else /* fUseCustomPrefix == TBX_FALSE */
{
        /* Use default stack values for Type of Number and Numbering plan ID, presentation and screening */
        un32OutbandCallingAddressPrefixLength = 0;

        /* Octet 4: Copy content of IA5 characters */
        strcpy (aun8OutbandCallingAddress, szCallingPhoneNb);
}
```

8.2.9.2.1.5 Outband calling subaddress

"Calling party sub address" information element as described in Q.931 section 4.5.11.
Fill only the subaddress information (octets 4+) in the *aun8OutbandCallingSubaddress* structure. Maximum size of *un32OutbandCallingSubaddressSize* is 20 octets.

8	7	6	5	4	3	2	1	Octet
Subaddress information								4 etc.

Code example:

```
/* Octet 4: Copy content of IA5 characters */
strcpy (&aun8OutbandCallingSubaddress[0], szSubAddressNb);
```

8.2.9.2.1.6 Cause

Refer to "Cause" information element as described in Q.931 section 4.5.12. This refers to Q.850, section 2.
aun8Cause must be defined when included in a structure. You must specify the octets according to standard in the *aun8Cause* structure, but only octets 4 and 5. Invalid entries might result in call refusal.

8	7	6	5	4	3	2	1	Octet Q.931
ext. 1	Cause value							4
Diagnostic(s) (if any)								5*

Code example:
/* Octet 4: Ext (1), Cause value = Normal, unspecified (001 1111) */
aun8Cause [0] = 0x9F;

/* octet 5: Diagnostic value = none (0) */
aun8Cause [1] = 0x0;

8.2.9.2.1.7 High Layer Compatibility

Refer to "High layer compatibility" information element as described in Q.931 section 4.5.17.
If *un32HighLayerCompatibilitySize* = 0 , this IE will not be sent. If you include this IE, you must specify the octets according to standard in the *aun8HighLayerCompatibility* structure. Invalid entries might result in call refusal. Maximum size of *un32HighLayerCompatibilitySize* is 4 octets.

8	7	6	5	4	3	2	1	Octet
ext. 1	Coding standard		Interpretation			Presentation method of protocol profile		3
ext. 0/1	High layer characteristics identification							4

Code example:
/* Set structure size */
un32HighLayerCompatibilitySize = 2;

/* Octet 3: Ext (1), Coding standard = ITU-T (00), Interpretation=primary (100), Presentation=high layer (01)*/
aun8HighLayerCompatibility [0] = 0x91;

/* Octet 4: Ext (1), High layer ID = telephony (0000001)*/
aun8HighLayerCompatibility [1] = 0x81;

8.2.9.2.1.8 Keypad Facility

Refer to "Keypad facility" information element as described in Q.931 section 4.5.18.
un32KeypadFacilitySize specifies the number of digits (IA5 characters) to include in the structure. *aun8KeypadFacility* are the IA5 characters. Maximum size of *un32KeypadFacilitySize* is 16 octets.

8	7	6	5	4	3	2	1	Octet
0	Keypad facility information (IA5 characters)							3 etc.

Code example:
/* Set structure size */
un32KeypadFacilitySize = strlen (szKeypadNb);

/* Octet 3: Copy content of IA5 characters */
strcpy (*aun8KeypadFacility*, szKeypadNb);

8.2.9.2.1.9 Layer Compatibility

Refer to "Low layer compatibility" information element as described in Q.931 section 4.5.19.
If *un32LowLayerCompatibilitySize* = 0 , this IE will not be sent. If you include this IE, you must specify the octets according to standard in the *aun8LowLayerCompatibility* structure. Invalid entries might result in call refusal. Maximum size of *un32HighLayerCompatibilitySize* is 16 octets.

8	7	6	5	4	3	2	1	Octet
ext. 0/1	Coding standard		Information transfer capability					3
ext. 1	Transfer mode		Information transfer rate					4
ext. 0/1	Layer 1 ident. 0	1	User information layer 1 protocol					5*

Code example:
/* Set structure size */
un32LowLayerCompatibilitySize = 2;

/* Octet 3: Ext (1), Coding standard = ITU-T (00), Information Transfer Capability = speech (00000) */
aun8LowLayerCompatibility [0] = 0x80;

/* Octet 4: Ext (1), Transfer Mode = circuit mode (00), Information Transfer rate = 64kbps (10000)*/
aun8LowLayerCompatibility [1] = 0x90;

/* Octet 5: ext (1), Layer 1 ident (01), User Information layer 1 protocol = G.711 ulaw (00010)*/
aun8LowLayerCompatibility [2] = 0xA2;

8.2.9.2.1.10 Progress Indicator

Refer to "Progress indicator" information element as described in Q.931 section 4.5.23.
Fill *un8ProgressIndicator* with progress description parameter.

8	7	6	5	4	3	2	1	Octet
ext. 1	Progress description							4

Code example:
/* Octet 4: Ext (1), Progress Description = Destination address is non-ISDN (0x02) */
un8ProgressIndicator = 0x82;

8.2.9.2.1.11 User to User

Refer to "User user" information element as described in Q.931 section 4.5.30.
If *un32UserToUserSize* = 0, , this IE will not be sent. If you include this IE, you must specify the octets according to standard in the *aun8UserToUser* structure. Invalid entries might result in call refusal. Maximum size of *un32UserToUserSize* for this IE is octets. This IE is carried transparently over the network.

8	7	6	5	4	3	2	1	Octet
Protocol discriminator								3
User information								4 etc.

Code example:
/* Set structure size */
un32UserToUserSize = 10;

```
/* Octet 3: Protocol Discriminator: IA5 characters (0x04) */
aun8UserToUser[0] = 0x04;
strcpy ((PTBX_CHAR) &aun8UserToUser[1], "123456789");
```

8.2.9.2.1.12 User (additionnal IE)

Optional information element(s) the user wants to send as described in Q.931. The only implemented IE is Sending Complete. Other IEs can be implemented upon customer requests.

Code example:

```
/* Send a Sending Complete IE */
un32UserSize = 1;          /* This IE contains only the IE value */
aun8User[0] = 0xA1;        /* Sending Complete IE value */
```

8.2.9.2.1.13 Redirecting

Refer to "Redirecting" information element as described in Q.931 section 4.6.7.
If *un32RedirectingSize* = 0 , this IE will not be sent. If you include this IE, you must specify the octets according to standard in the *aun8Redirecting* structure. Invalid entries might result in call refusal. Maximum size of *un32RedirectingSize* is 16 octets.

8	7	6	5	4	3	2	1	Octet
ext. 0/1		Type of number			Numbering plan identification			3
ext. 0/1	Presentation indicator		0	Spare 0	0	Screening indicator		3a* 1
ext. 1	0	Spare 0	0		Reason for redirection			3b* 1
Spare 0				Number digits (IA5 characters)				4 etc.

Code example:

```
/* Set structure size */
un32RedirectingSize = strlen (szRedirectingPhoneNb);

/* Octet 3: Ext (0), Type of number = International (001), Numbering plan ID = ISDN (E.164) (0001) */
aun8Redirecting [0] = 0x11;

/* Octet 3a: Ext (0), Presentation Ind. = restricted (01), (000), Screening Ind. = Network provided (11) */
aun8Redirecting [1] = 0x23;

/* Octet 3b: Ext (1), (000), Reason for redirection. = Call deflection (0100) */
aun8Redirecting [2] = 0x84;

/* Octet 4: Copy content of IA5 characters */
strcpy (&aun8Redirecting[3], szRedirectingPhoneNb);
```

8.2.10 PRI ISDN Call scenarios (Stack and TB640 APIs)

The following call scenarios show the different Q.931 API message exchange required to establish a call. Every table in the following sections is divided in four columns. The leftmost and rightmost columns describe the actual TB640 API messages that are sent and received by the application. The two middle columns represent the Q.931 primitives exchanged by the Q.931 stacks. Those primitives are never seen by neither user applications. The reader must remember that each user application request is answered individually by the TB640 blade (not the stack) to confirm they all have been delivered to the signaling stack. Those blade "responses" are <u>not</u> shown in the call scenarios. Also note that the call scenarios are shown using the extended API but can be translated to the original API using Table 15.

8.2.10.1 Successful call placed from network-side

Network-side API prim	Network-side Q.931	User-side Q.931	User-side API prim
CMD_INITIATE_CALL ->	SETUP ->	SETUP ->	NOTIF_INCOMING_CALL ->
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_CALL_PROCEEDING]	<- CALL PROC	<- CALL PROC	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_CALL_PROCEEDING] (only required for AUS_PRI switch variant)
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_ALERT]	<- ALERT	<- ALERT	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_ALERT]
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_NETWORK_CONNECT]	<- CONN	<- CONN	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_CONNECT_RESPONSE]
CMD_STATE_CHANGE_REQUEST -> [REQUEST_TYPE_CONNECT_ACK]	CONN_ACK ->	CONN_ACK->	<nothing> if response received within T313 seconds
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_CONNECT_CONFIRM]			

8.2.10.2 Successful call placed from network-side (overlap mode)

Network-side API prim	Network-side Q.931	User-side Q.931	User-side API prim
CMD_INITIATE_CALL -> (must contain at least 1 digit information)	SETUP ->	SETUP ->	NOTIF_INCOMING_CALL ->
		<- SETUP_ACK	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_MORE_INFO]
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_SETUP_ACK]	INFO ->		
			NOTIF_STATE_CHANGE_EVENT -> [NOTIF_TYPE_KEYPAD]
CMD_STATE_CHANGE_REQUEST -> [REQUEST_TYPE_KEYPAD]	INFO ->		
			NOTIF_STATE_CHANGE_EVENT -> [NOTIF_TYPE_KEYPAD]
Digit information exchange with REQUEST_TYPE_KEYPAD and NOTIF_TYPE_KEYPAD			
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_CALL_PROCEEDING]	<- CALL PROC	<- CALL PROC	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_CALL_PROCEEDING] (only required for AUS_PRI switch variant)
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_ALERT]	<- ALERT	<- ALERT	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_ALERT]
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_NETWORK_CONNECT]	<- CONN	<- CONN	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_CONNECT_RESPONSE]
CMD_STATE_CHANGE_REQUEST -> [REQUEST_TYPE_CONNECT_ACK]	CONN_ACK ->	CONN_ACK->	<nothing> if response received within T313 seconds
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_CONNECT_CONFIRM]			

8.2.10.3 Unsuccessful Call placed from Network-side

Network-side API prim	Network-side Q.931	User-side Q.931	User-side API prim
CMD_INITIATE_CALL ->	SETUP ->	SETUP ->	NOTIF_INCOMING_CALL ->
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_CALL_PROCEEDING]	<- CALL PROC	<- CALL PROC	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_CALL_PROCEEDING] (only required for AUS_PRI switch variant)
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_ALERT]	<- ALERT	<- ALERT	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_ALERT]
	<lost>	<- CONN	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_CONNECT_RESPONSE]
	<lost>	T313 expires <- CONN	[only for 4ESS/5ESS PRI]
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_DISCONNECT]	<- REL	T313 expires <- REL	[ALL SWITCHES – for 4ESS/5ESS after retrans.]
	REL_COM ->	-> REL_COM	NOTIF_STATE_CHANGE_EVENT -> [NOTIF_TYPE_DISCONNECT]

8.2.10.4 Unsuccessful Call placed from Network-side (app. Timeout)

Network-side API prim	Network-side Q.931	User-side Q.931	User-side API prim
CMD_INITIATE_CALL ->	SETUP ->	SETUP ->	NOTIF_INCOMING_CALL ->
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_CALL_PROCEEDING]	<- CALL PROC	<- CALL PROC	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_CALL_PROCEEDING] (only required for AUS_PRI switch variant)
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_ALERT]	<- ALERT	<- ALERT	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_ALERT]
	<lost>	<- CONN	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_CONNECT_RESPONSE]
	<lost>	T313 expires <- CONN	[only for 4ESS/5ESS PRI]
<application timeout>			
CMD_STATE_CHANGE_REQUEST -> [REQUEST_TYPE_DISCONNECT]	DISC ->	DISC ->	
<- NOTIF_STATE_CHANGE_EVENT [TYPE_DISCONNECT_CONFIRM]	<- REL	<- REL	
	REL_COM->	REL_COM->	NOTIF_STATE_CHANGE_EVENT -> [NOTIF_TYPE_DISCONNECT] with IE from DISC

8.2.10.5 Refused Call placed from Network-side

Network-side API prim	Network-side Q.931	User-side Q.931	User-side API prim
CMD_INITIATE_CALL ->	SETUP ->	SETUP ->	NOTIF_INCOMING_CALL ->
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_DISCONNECT]	<- REL_COM	<- REL_COM	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_DISCONNECT]
			NOTIF_STATE_CHANGE_EVENT -> [TYPE_DISCONNECT_CONFIRM]

8.2.10.6 Discontinued Call placed from Network-side

Network-side API prim	Network-side Q.931	User-side Q.931	User-side API prim
CMD_INITIATE_CALL ->	SETUP ->	SETUP ->	NOTIF_INCOMING_CALL ->
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_CALL_PROCEEDING]	<- CALL PROC	<- CALL PROC	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_CALL_PROCEEDING] (only required for AUS_PRI switch variant)
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_ALERT]	<- ALERT	<- ALERT	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_ALERT]
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_NETWORK_CONNECT]	<- CONN	<- CONN	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_CONNECT_RESPONSE]
CMD_STATE_CHANGE_REQUEST -> [REQUEST_TYPE_DISCONNECT]	DISC ->	DISC ->	
<- NOTIF_STATE_CHANGE_EVENT [TYPE_DISCONNECT_CONFIRM]	<- REL	<- REL	
	REL_COM->	REL_COM->	NOTIF_STATE_CHANGE_EVENT -> [NOTIF_TYPE_DISCONNECT] with IE from DISC

8.2.10.7 Call placed from Network-side, discontinued by User-side

Network-side API prim	Network-side Q.931	User-side Q9.31	User-side API prim
CMD_INITIATE_CALL ->	SETUP ->	SETUP ->	NOTIF_INCOMING_CALL ->
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_CALL_PROCEEDING]	<- CALL PROC	<- CALL PROC	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_CALL_PROCEEDING] (only required for AUS_PRI switch variant)
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_ALERT]	<- ALERT	<- ALERT	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_ALERT]
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_NETWORK_DISCONNECT]	<- DISC	<- DISC	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_DISCONNECT]
CMD_STATE_CHANGE_REQUEST -> [REQUEST_TYPE_DISCONNECT]	REL ->	REL ->	NOTIF_STATE_CHANGE_EVENT -> [TYPE_DISCONNECT_CONFIRM]
<- NOTIF_STATE_CHANGE_EVENT [TYPE_DISCONNECT_CONFIRM]	<- REL_COM	<- REL_COM	

8.2.10.8 Call collision (same B-channel)

Network-side API prim	Network-side Q.931	User-side Q9.31	User-side API prim
CMD_INITIATE_CALL ->	SETUP ->		
		<- SETUP	<- CMD_INITIATE_CALL
	<- SETUP	SETUP ->	
	REL_COM ->	<- REL_COM	
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_DISCONNECT]	<- REL_COM	REL_COM ->	NOTIF_STATE_CHANGE_EVENT -> [NOTIF_TYPE_DISCONNECT]

8.2.10.9 Successful Call placed from User-side

Network-side API prim	Network-side Q.931	Q9.31	User-side API prim
<- `NOTIF_INCOMING_CALL`	<- SETUP	<- SETUP	<- `CMD_INITIATE_CALL`
`CMD_STATE_CHANGE_REQUEST` -> `[REQUEST_TYPE_CALL_PROCEEDING]`	CALL_PROC ->	CALL_PROC ->	`NOTIF_STATE_CHANGE_EVENT` -> `[NOTIF_TYPE_CALL_PROCEEDING]`
`CMD_STATE_CHANGE_REQUEST` -> `[REQUEST_TYPE_ALERT]`	ALERT ->	ALERT ->	`NOTIF_STATE_CHANGE_EVENT` -> `[NOTIF_TYPE_ALERT]`
`CMD_STATE_CHANGE_REQUEST`-> `[REQUEST_TYPE_CONNECT_RESPONSE]`	CONN ->	CONN ->	`NOTIF_STATE_CHANGE_EVENT` -> `[NOTIF_TYPE_CONNECT_CONFIRM]`
	<- CONN_ACK	<- CONN_ACK	[optional for network-side]

8.2.10.10 Successful Call placed from User-side (overlap mode)

Network-side API prim	Network-side Q.931	User-side Q9.31	User-side API prim
<- `NOTIF_INCOMING_CALL`	<- SETUP	<- SETUP	<- `CMD_INITIATE_CALL` (must contain at least 1 digit information)
`CMD_STATE_CHANGE_REQUEST` -> `[REQUEST_TYPE_MORE_INFO]`	SETUP_ACK ->		
		<-INFO	`NOTIF_STATE_CHANGE_EVENT` -> `[NOTIF_TYPE_SETUP_ACK]`
<- `NOTIF_STATE_CHANGE_EVENT` `[NOTIF_TYPE_KEYPAD]`			
		<-INFO	<- `CMD_STATE_CHANGE_REQUEST` `[REQUEST_TYPE_KEYPAD]`
<- `NOTIF_STATE_CHANGE_EVENT` `[NOTIF_TYPE_KEYPAD]`			
Digit information exchange with `REQUEST_TYPE_KEYPAD` and `NOTIF_TYPE_KEYPAD`			
`CMD_STATE_CHANGE_REQUEST` -> `[REQUEST_TYPE_CALL_PROCEEDING]`	CALL_PROC ->	CALL_PROC ->	`NOTIF_STATE_CHANGE_EVENT` -> `[NOTIF_TYPE_CALL_PROCEEDING]`
`CMD_STATE_CHANGE_REQUEST` -> `[REQUEST_TYPE_ALERT]`	ALERT ->	ALERT ->	`NOTIF_STATE_CHANGE_EVENT` -> `[NOTIF_TYPE_ALERT]`
`CMD_STATE_CHANGE_REQUEST`-> `[REQUEST_TYPE_CONNECT_RESPONSE]`	CONN ->	CONN ->	`NOTIF_STATE_CHANGE_EVENT` -> `[NOTIF_TYPE_CONNECT_CONFIRM]`
	<- CONN_ACK	<- CONN_ACK	[optional for network-side]

8.2.10.11 Unsuccessful Call placed from User-side

Network-side API prim	Network-side Q.931	User-side Q9.31	User-side API prim
<- NOTIF_INCOMING_CALL	<- SETUP	<- SETUP	<- CMD_INITIATE_CALL
CMD_STATE_CHANGE_REQUEST -> [REQUEST_TYPE_CALL_PROCEEDING]	CALL_PROC ->	CALL_PROC ->	NOTIF_STATE_CHANGE_EVENT -> [NOTIF_TYPE_CALL_PROCEEDING]
CMD_STATE_CHANGE_REQUEST -> [REQUEST_TYPE_ALERT]	ALERT ->	ALERT ->	NOTIF_STATE_CHANGE_EVENT -> [NOTIF_TYPE_ALERT]
CMD_STATE_CHANGE_REQUEST-> [REQUEST_TYPE_CONNECT_RESPONSE]	CONN ->	<lost>	
			<application timer expires>
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_NETWORK_DISCONNECT]	<- DISC	<- DISC	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_DISCONNECT]
CMD_STATE_CHANGE_REQUEST -> [REQUEST_TYPE_DISCONNECT]	REL ->	REL ->	NOTIF_STATE_CHANGE_EVENT -> [TYPE_DISCONNECT_CONFIRM]
<- NOTIF_STATE_CHANGE_EVENT [TYPE_DISCONNECT_CONFIRM]	<- REL_COM	<- REL_COM	

8.2.10.12 Refused Call placed from User-side

Network-side API prim	Network-side Q.931	User-side Q9.31	User-side API prim
<- NOTIF_INCOMING_CALL	<- SETUP	<- SETUP	<- CMD_INITIATE_CALL
CMD_STATE_CHANGE_REQUEST -> [REQUEST_TYPE_DISCONNECT]	REL_COM ->`	REL_COM ->	NOTIF_STATE_CHANGE_EVENT -> [NOTIF_TYPE_DISCONNECT]
<- NOTIF_STATE_CHANGE_EVENT [TYPE_DISCONNECT_CONFIRM]			

8.2.10.13 Discontinued Call placed from User-side

Network-side API prim	Network-side Q.931	User-side Q9.31	User-side API prim
<- NOTIF_INCOMING_CALL	<- SETUP	<- SETUP	<- CMD_INITIATE_CALL
CMD_STATE_CHANGE_REQUEST -> [REQUEST_TYPE_CALL_PROCEEDING]	CALL_PROC ->	CALL_PROC ->	NOTIF_STATE_CHANGE_EVENT -> [NOTIF_TYPE_CALL_PROCEEDING]
CMD_STATE_CHANGE_REQUEST -> [REQUEST_TYPE_ALERT]	ALERT ->	ALERT ->	NOTIF_STATE_CHANGE_EVENT -> [NOTIF_TYPE_ALERT]
<- NOTIF_STATE_CHANGE_EVENT [NOTIF_TYPE_NETWORK_DISCONNECT]	<- DISC	<- DISC	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_DISCONNECT]
CMD_STATE_CHANGE_REQUEST -> [REQUEST_TYPE_DISCONNECT]	REL ->	REL ->	NOTIF_STATE_CHANGE_EVENT -> [TYPE_DISCONNECT_CONFIRM]
<- NOTIF_STATE_CHANGE_EVENT [TYPE_DISCONNECT_CONFIRM]	<- REL_COM	<- REL_COM	

8.2.10.14 Call placed from User-side, discontinued by Network-side

Network-side API prim	Network-side Q.931	User-side Q9.31	User-side API prim
<- NOTIF_INCOMING_CALL	<- SETUP	<- SETUP	<- CMD_INITIATE_CALL
CMD_STATE_CHANGE_REQUEST -> [REQUEST_TYPE_CALL_PROCEEDING]	CALL_PROC ->	CALL_PROC ->	NOTIF_STATE_CHANGE_EVENT -> [NOTIF_TYPE_CALL_PROCEEDING]
CMD_STATE_CHANGE_REQUEST -> [REQUEST_TYPE_ALERT] <optional>	ALERT -> <optional>	ALERT -> <optional>	NOTIF_STATE_CHANGE_EVENT -> [NOTIF_TYPE_ALERT] <optional>
(1) CMD_STATE_CHANGE_REQUEST -> [REQUEST_TYPE_DISCONNECT]	DISC ->	DISC ->	**(1)**
(1) <- NOTIF_STATE_CHANGE_EVENT [TYPE_DISCONNECT_CONFIRM]	<- REL	<- REL	**(1)**
(1)	REL_COM ->	REL_COM ->	**(1)** NOTIF_STATE_CHANGE_EVENT -> [NOTIF_TYPE_DISCONNECT] with IE from DISC
(2) CMD_STATE_CHANGE_REQUEST -> [REQUEST_TYPE_DISCONNECT]	REL ->	REL ->	**(2)** NOTIF_STATE_CHANGE_EVENT -> [NOTIF_TYPE_DISCONNECT]
(2) <- NOTIF_STATE_CHANGE_EVENT [TYPE_DISCONNECT_CONFIRM]	<- REL_COM	<- REL_COM	**(2)**

(1) These are applicable only when configured with NI2 variant.
(2) These are applicable only when configured with DMS, NET5, HK, 4ESS, 5ESS, AUS PRI variants.

8.2.10.15 Disconnect collision (scenario starts in active state)

Network-side API prim	Network-side Q.931	User-side Q9.31	User-side API prim
CMD_STATE_CHANGE_REQUEST -> [REQUEST_TYPE_DISCONNECT]	DISC ->		
		<- DISC	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_DISCONNECT]
	<- DISC	DISC ->	
	REL ->	REL ->	
<- NOTIF_STATE_CHANGE_EVENT [TYPE_DISCONNECT_CONFIRM]	<- REL_COM	REL_COM ->	NOTIF_STATE_CHANGE_EVENT -> [TYPE_DISCONNECT_CONFIRM]

8.2.11 PRI ISDN Call collision scenarios (TB640 and user application)

The following call scenarios show the different API message exchanges between the TB640 adapter (board or simulator) and the user applications required to establish/tear down a call. Every table in the following sections is divided in four columns. The leftmost column describes the Q.931 stacks residing on the TB640 adapter. The middle column represents the TB640 software creating the interface between the stack and user application. The rightmost column represents the user application. Thus, the user application will see the message sequence shown in the rightmost column. It is important to understand that since all messages are asynchronous, the delay between the time a request has been made from the application and the time that same request is received by the TB640, it is possible to have some "collisions" between calls. The following scenarios describe most of those corner cases. In those scenarios, the "responses" for the user application requests are shown in ***<bold italic>***.

8.2.11.1 Connect collision (ISDN call arrived first)

For this collision to occur, there most be only one available timeslot left on the trunk before the scenario begins. In this case, the call coming from the network gets the last resource assigned while the call initiated by the user application is refused.

Q.931 stack	TB640	User application
SETUP->		
	Generates `NOTIF_INCOMING_CALL` to the user application (in response to the SETUP) ->	<- `CMD_INITIATE_CALL`
		`NOTIF_INCOMING_CALL` ->
	Generates ***RSP_CMD_INITIATE_CALL*** (ok) to the user application in response to the `CMD_INITIATE_CALL`.	
		RSP_CMD_INITIATE_CALL (ok) ->
	Generates a `NOTIF_STATE_CHANGE_EVENT` (`NOTIF_TYPE_DISCONNECT`) for the call generated from the application	
		`NOTIF_STATE_CHANGE_EVENT` -> `[NOTIF_TYPE_DISCONNECT]`

8.2.11.2 Connect collision (User application call arrived first)

For this collision to occur, there most be only one available timeslot left on the trunk before the scenario begins. In this case, the call coming from the network is refused while the last resource is assigned to the call initiated by the user application.

Q.931 stack	TB640	User application
		<- `CMD_INITIATE_CALL`
SETUP->	Generates ***RSP_CMD_INITIATE_CALL*** (ok) to the user application in response to the `CMD_INITIATE_CALL`.	
	<- REL_COM	***RSP_CMD_INITIATE_CALL*** (ok) ->
<- REL_COM		

8.2.11.3 Connect collision (both call received at the same time in the stack)

This is the same scenario as in section 8.2.10.8. Both sides will receive a disconnect indication.

8.2.11.4 Disconnect collision (stack disconnect first)

For this collision to occur, the stack (or remote caller) disconnects the call at the same time the user application sends a message for disconnecting the same call. The two messages “cross each other” over the API transport path (i.e. the ethernet link).

Q.931 stack	TB640	User application
REL->		
	Generates NOTIF_DISCONNECT_INDICATION to the user application (in response to the REL)	<- CMD_STATE_CHANGE_REQUEST [REQUEST_TYPE_DISCONNECT]
		NOTIF_STATE_CHANGE_EVENT -> [NOTIF_TYPE_DISCONNECT]
	Generates ***RSP_CMD_STATE_CHANGE_REQUEST*** (not found) to the user application in response to the REQUEST_TYPE_DISCONNECT.	
		RSP_CMD_STATE_CHANGE_REQUEST (with an error code NOT_FOUND) ->

8.3 CAS Signaling

Most of the R1 and R2 information contained in this section comes from the *"930i Protocol User's Manual"* from PaloVerde Internacional, Inc.

8.3.1 Trunk configuration

Before a signaling stack can be started, the underlying trunk must be configured correctly. The configuration depends on the CAS switch variant that will be used in the system.

Table 20: CAS variants

Switch variant	Trunk type expected
TB640_CAS_VARIANT_WINK_START	T1/J1
TB640_CAS_VARIANT_FXS_GROUND_START	T1/J1
TB640_CAS_VARIANT_FXS_LOOP_START (must be paired with FXO)	T1/J1
TB640_CAS_VARIANT_FXO (must be paired with FXS_LOOP_START)	T1/J1
TB640_CAS_VARIANT_TAIWAN_R1 (also named « Taiwan modified R1 »)	T1/J1 or E1
TB640_CAS_VARIANT_R2_CHINA	E1
TB640_CAS_VARIANT_R2_KOREA	E1
TB640_CAS_VARIANT_R2_SINGAPORE	E1

Note: FXS_LOOP_START switch variant is a "user-side" protocol only and must be used with "network-side" configured as FXO switch variant.

8.3.2 Physical link status

The signaling stack is decoupled of the transport mechanism (both ABCD bits and tones) When a trunk line goes down, the stack is no longer able to communicate with its peer. When regular alarms of the trunks (Loss-of-signal, Remote-alarm-indication, etc) occur, the stack sends an event to the user-application (*TB640_MSG_ID_CAS_NOTIF_STATUS_INDICATION)* containing the value *TB640_CAS_STATUS_IND_VALUE_PHYSICAL_LINE_DEACTIVATED*. Note that a user-application can always read this state using the *TB640_MSG_ID_CAS_STATES_GET* API message. No connection openings can occur when the stack is down (all attempts will be refused). Pending connections will probably time-out and be closed by the stack while active calls will stay opened until manually disconnected by the user-application (remember that it is not because the stack is down that the already established connections are down as well). The user application should be careful when closing active calls when the stack is down because it does not mean the peer stack will do it as well. Thus, when the physical link will be up again, the two stacks will not have the same knowledge of which timeslot is free and which one is used. To make sure the user-application knows the exact state of the stack at all time, it should first create an event filter to capture CAS events coming from the stack and then use *TB640_MSG_ID_CAS_STATES_GET* to retrieve the current state. Following that sequence will guarantee the application will have an exact knowledge of the stack state (and never miss the event).

8.3.3 Call handle and user contexts

The CAS stack provides the same user contexts functionality as the ISDN stack as described in section 8.2.4.

8.3.4 CAS Basic knowledge

As in almost every standardized protocol, there is a multitude of variants and special utilization that makes it very hard to describe precisely message and event exchanges between two signaling entities. By opposition to Q.931 ISDN where all exchanges are message-based, R1 and R2 signaling protocols make use of a combination of AB/ABCD bits tied to the physical transport layer (T1/J1/E1 trunks) and tones.

The AB/ABCD bits usually represent a state changes (OFF-HOOK, ON-HOOK) which translate to a call state such as “line seizure”, “seizure acknowledge”, “call acceptance”, “call clearance”, etc. Each physical channel (usually referred to as a “timeslot”) owns a set of those signaling bits. Most of the protocols only make uses of the AB bits but some variants do implement some charging information into the two remaining bits. Signaling bits transport is different whether the protocol runs over T1/J1 or E1. When the trunk is configured in T1 or J1, the ABCD bits transport mechanism is called “robbed-bit signaling” since least significant bits of a timeslot are stolen once in a while to store the ABCD bits instead of voice data. This makes usual 64kbps channels to go down to 56kbps (but this degradation is not audible for human hearing). Even if signaling isn’t used during normal lifetime of a voice conversation, they are still required to detect the “call release” event (when one of the talkers hangs up). Also, depending on the framing choice (SF/D4, ESF) the T1/J1 frame restricts the number of signaling bits available. In older framing (SF/D4) only AB bits are available while ESF allows all 4 bits. E1, by opposition, uses a well-known timeslot to carry all ABCD bits for every voice timeslots. This method is therefore more efficient since it leaves the complete 64kbps channel to carry voice.

Tones are transmitted in-band between the two signaling entities to exchange other types of information such as “called number”, “calling number”, etc. As for the signaling bits, the tones are tied to a particular “timeslot” being transferred into the same channel (or timeslot) as the final voice call. Since those tones are transmitted during call setup phase, the end-user never gets to hear those tones. The frequencies and duration of tones are generally very well accepted in standards and localized variants of the protocols. What changes is the meaning of those tones (also referred to as “digits” sometimes. Those are different for R1 and R2 protocols and even vary amongst R2 variants depending on the state of the signaling sequence. It would be possible to establish an R1 call only using the ABCD bits but the receiving end wouldn’t have any information about the “called number”. In R2 CAS signaling, the tones are mandatory since the receiving end sends commands to the originating end of the information to be transmitted.

Generation and detection of those tones uses internal hardware resources to be connected on the timeslot and be ready to listen or play frequencies when calls are requested or received. These resource allocations are hidden from the user-application and dealt-with automatically by the signaling stack. But in order for those resources to be connected and disconnected automatically, the user application has some responsibilities to follow. The end-user cannot use a trunk resource (timeslot) returned by the signaling stack until a call is properly connected. Also, the end-user MUST disconnect any connections to the trunk resource (timeslot) before calling the TB640_MSG_CAS_CMD_RELEASE message. This is shown in the call scenarios in sections 8.3.5.1 and 0.

The two following sections will explain respectively the R1 and R2 types of channel associated signaling more in details with the meaning of the tones/digits.

Important note 1: CAS protocols are first initiated by CAS bits variations. Thus, these protocols are extremely sensitive to glitches in the framing encoding used (T1 SF/ESF, E1 double or multi-frame). Therefore, it is absolutely critical to have the clocks configured properly in a system to avoid those situations. Make sure your system takes the clock from a reliable and unique source (e.g. a specific T1/E1 line connected to a network switch of a provider). Having more than one clock source can also be used in case the primary clock source fails.

Important note 2: T1/J1 CAS protocols have no way to detect the presence of a remote CAS-enabled peer other than checking the status of the line (LOS/LOF errors) by opposition to E1 that can detect framing on the signaling timeslot. Since the seizure of a line (i.e. start of a call) is the variation of CAS bits, “false seizure” condition can occur if one side “unblocks” its respective timeslots before the other stack was able to set the proper idle patterns on the CAS bits. Usually, false seizured calls will be closed automatically after a while because the proper seizure time will not be respected, but this varies from one CAS protocol to the other. Therefore, the user should enable the CAS protocol on the remote side (usually a switch) after having properly setup the CAS stack on the TB640.

8.3.4.1 R1 CAS Basics

As mentioned before, an R1 CAS call could be established using the ABCD bits alone but wouldn't carry any call information (and therefore wouldn't be much useful except on dedicated lines). In its simplest form, R1 CAS uses ABCD bits to signal the "line seizure" (call requested), "seizure acknowledge" (peer stack acknowledge the timeslot is now being used), "call acceptance" (call is answered by peer), "call clearance" (call is either refused or is terminated) states of a calls. After the peer stack has acknowledged the line seizure, tones (also called DTMF for "dual tone multi frequency or MF for "multi-frequency") are played on the selected timeslot to transmit the "called number" to the peer stack. Basically, each tone represents a digit (similar to the buttons of a touch-tone phone). The peer stack records the received digits until it decides it has enough information (or until a timeout occurs) and continues with the ABCD bits signaling to establish the call. Depending on the variant, one or both the call originator and receiver transmit digits over the channel. It is also possible on certain R1 switch variants, when the call is established from the network side, for the parties not to send any digit at all. This is due to the R1 specifications making this portion optional from the network-side for those variants. Once the call is established, the DTMF tones are no longer used for the call. Table 21 and

Table 22 define the meaning of those “digit” in R1 CAS and those are common to all currently supported variants of R1.

Table 21: R1 CAS DTMF digits

Digit	Frequencies	Meaning
‘0’	941Hz + 1336Hz	Digit ‘0’
‘1’	697Hz + 1209Hz	Digit ‘1’
‘2’	697Hz + 1336Hz	Digit ‘2’
‘3’	697Hz + 1477Hz	Digit ‘3’
‘4’	770Hz + 1209Hz	Digit ‘4’
‘5’	770Hz + 1336Hz	Digit ‘5’
‘6’	770Hz + 1477Hz	Digit ‘6’
‘7’	852Hz + 1209Hz	Digit ‘7’
‘8’	852Hz + 1336Hz	Digit ‘8’
‘9’	852Hz + 1477Hz	Digit ‘9’
‘A’	697Hz + 1633Hz	
‘B’	770Hz + 1633Hz	
‘C’	852Hz + 1633Hz	
‘D’	941Hz + 1633Hz	
‘*’	941Hz + 1209Hz	Digit ‘*’
‘#’	941Hz + 1477Hz	Digit ‘#’

Table 22: MFR1 CAS digits

Digit	Frequencies	Meaning
‘0’	1300Hz + 1500Hz	Digit ‘0’
‘1’	700Hz + 900Hz	Digit ‘1’
‘2’	700Hz + 1100Hz	Digit ‘2’
‘3’	900Hz + 1100Hz	Digit ‘3’
‘4’	700Hz + 1300Hz	Digit ‘4’
‘5’	900Hz + 1300Hz	Digit ‘5’
‘6’	1100Hz + 1300Hz	Digit ‘6’
‘7’	700Hz + 1500Hz	Digit ‘7’
‘8’	900Hz + 1500Hz	Digit ‘8’
‘9’	1100Hz + 1500Hz	Digit ‘9’
‘A’	900Hz + 1700Hz	(*end-of-pulsing for Taiwan modified R1)
‘B’	1300Hz + 1700Hz	
‘C’	700Hz + 1700Hz	
‘D’	--	
‘E’	1500Hz + 1700Hz	End-of-pulsing (*not used for Taiwan modified R1)
‘F’	1100Hz + 1700Hz	Start-of-pulsing

One mode of operation of the R1 CAS (except for Taiwan modified R1) is called “Direct inward dialing” (or DID in short). Basically, this mode is usually used when two PBXs are connected together (no human intervention). Since the protocol conversation occurs between two machines, the digits are expected to be much faster than when a human presses the buttons of his touch-tone phone. Thus, the protocol delays are much smaller and the protocol stack expects all digits to be received from its peer before even informing the application of the new call presence. When not using DID, the stack expects the digits to come one by one at a slow speed. Therefore, it informs the application about the new call before the first digit is received. After that, it forwards the digits to the application one by one (as they are received) for the application to decide, at any moment, to accept or refuse the call. This is shown in the different call scenarios described in sections 8.3.5.1

8.3.4.2 R2 CAS Basics

R2 signaling has been designed to be more flexible than R1 regarding the amount and type of information transmitted between the two peers. Although the ABCD bits are different from R1, they mostly reflect the same type of call state transition. The real difference reside in the signaling information transmitted using tones. First, the number of digits has been increased compared to R1 signaling. This method is called MFCR2 (multi frequency compelled R2). There are 15 digits (from ‘1’ to ‘F’ as with the hexadecimal code where ‘0’ stands for 10 – there is no ‘A’) used during tone exchange. Also, we can really consider the tone portion of a call to be an “exchange of tones” in R2 since both ends of the calls sends information using those tones.

At this point, we need to define two terms widely used in R2 CAS: the call originator is called “forward” and the call receiver is called “backward”. These definitions are not to be confused with “network” or “user” side. Both protocol sides can be either forward or backward and they can be both at the same time since this appellation is only relative to a specific call. Thus, a signaling stack configured in “network-side” is considered the “forward-end” on an outgoing call and be considered a “backward-end” on a incoming call (same concept applies for the “user-side”).

We needed to define this terminology because, in R2 signaling, the “backward-end” of a call has the capability of requesting whatever information is required to complete the call. In short, the backward-end tells to the forward-end what information to send. After the initial call setup sequence using ABCD bits, the forward end starts sending the first “called number” digits using MFC-R2 “forward” allocated frequencies (they are different for forward and backward digits). The forward-end continues playing the tone until it sees a response from the backward-end or until it times-out. Once the receiving end “sees” the tone, it sends a backward tone telling the forward-end, first, that it can stop sending the tone and, second, what information to send in the next digit. Thus, the “backward digit string” is a series of digits that the receiving-end will send to the originating-end that tells, digit by digit, what to send next (calling number, category of the call, etc).

This is actually the strength of R2 signaling because the information requested can change depending on the capabilities or needs of the receiving end. One application of this protocol is when a call is routed from one switch to the other during call setup (e.g. for long distance call). Every new switch that enters the call flow (thus considered then backward-end) can ask the originating end (forward-end) to replay all or part of the destination number for routing purposes.

If R2 would be limited to the capabilities mentioned until now, the protocol would be rather straightforward. But, the R2 specifications added different meaning for the tones depending on a call state. For example, the forward-end states can be divided in up to three groups (called I, II and III) while the backward-end states can be divided in up to three groups as well (called A,B and C). Those groups (or states) are used during the tones exchange portion of the call only. The decision to advance from one group to the other is decided by the "backward digit string" (meaning that one of the digit in the string tells that further digits will be part of the following group). Thus, the backward-end can request information such as "called number", "calling number", "country code" and so on, as part of the first group and can then instruct the forward-end to send other type of information. What makes the R2 protocol hard to follow is that the meaning of those digits is different from one switch variant to the other. For example, there are some special digits that change the meaning of the digits depending on the command that was requested by the backward-end (i.e. in R2 China).

The following tables from section 8.3.4.2.1 to 8.3.4.2.4 list the meaning of the MFC-R2 digits for each variant. These won't tell how to use them in what order for a specific switch. The end-user would have to refer to the particular switch configuration to make sure. For example, using the R2 China switch variant, a backward digit string of '1116111131' tells the forward end to send 4 digits of the "called number" followed by the category digit, then the four digits of the "calling number" (remember that the forward-end always start the "dialog" by sending the first digit of the "called number" which is then answered by the first digit of the backward digit string). The signaling stack also has the possibility to automatically request digits until the forward-end has no more to send using the 'R' value in the backward digit string. This value is not part of the R2 protocol and is added to avoid the need to dynamically change the backward digit string. For example, the string "11161R31" does the same thing as the previous one except the backward-end will repeat the '1' digits until it receives a terminating digit 'F' from the forward-end. In this case, it allows the backward-end to accumulate a "calling number" of an unknown size.

Those previous string works for R2 China variant but wouldn't work for R2 Korea or Singapore. A backward string equivalent for R2 Korea would look like "1115555536" (or "1115R36") and "1116666631" (or "1116R31") for R2 Singapore. The following tables only give a summary of the meaning of each digit. The end-user **must** understand how the particular switch works in order to make a successful call with this product.

8.3.4.2.1 R2 China digits

Table 23: R2 CAS China, Group I (forward) digits

Signal	Digit	Frequencies	Meaning
I-1	‘1’	1380Hz + 1500Hz	Digit ‘1’
I-2	‘2’	1380Hz + 1620Hz	Digit ‘2’
I-3	‘3’	1500Hz + 1620Hz	Digit ‘3’
I-4	‘4’	1380Hz + 1740Hz	Digit ‘4’
I-5	‘5’	1500Hz + 1740Hz	Digit ‘5’
I-6	‘6’	1620Hz + 1740Hz	Digit ‘6’
I-7	‘7’	1380Hz + 1860Hz	Digit ‘7’
I-8	‘8’	1500Hz + 1860Hz	Digit ‘8’
I-9	‘9’	1620Hz + 1860Hz	Digit ‘9’
I-10	‘0’	1740Hz + 1860Hz	Digit ‘0’
I-11	‘B’	1380Hz + 1980Hz	Unused
I-12	‘C’	1500Hz + 1980Hz	No origin information is available
I-13	‘D’	1620Hz + 1980Hz	Access to test equipment
I-14	‘E’	1740Hz + 1980Hz	Unused
I-15	‘F’	1860Hz + 1980Hz	End of origin number

Table 24: R2 CAS China, Special signals

Signal group	Meaning
KA	Category code sent in response to group A-6 backward digit
KB	Group B backward digit (subscriber status), sent after group A-3 backward digit. Not available
KC	Category code sent in response to group A-6 backward digit
KD	Originating call type sent in response to group A-3 backward digit
KE	Category code sent sent in response to group A-6 backward digit (toll-to-local and local-to-local)

Table 25: R2 CAS China, Group I (forward) KA digits

Signal	Digit	Frequencies	Meaning
I-1	‘1’	1380Hz + 1500Hz	Ordinary subscriber, monthly charging
I-2	‘2’	1380Hz + 1620Hz	Ordinary subscriber, immediate charging
I-3	‘3’	1500Hz + 1620Hz	Ordinary subscriber, traffic service hall
I-4	‘4’	1380Hz + 1740Hz	Priority class 1, immediate charging
I-5	‘5’	1500Hz + 1740Hz	Priority class 2, free of charge
I-6	‘6’	1620Hz + 1740Hz	Unused
I-7	‘7’	1380Hz + 1860Hz	Priority class 1, monthly charging
I-8	‘8’	1500Hz + 1860Hz	Priority class 2, monthly charging
I-9	‘9’	1620Hz + 1860Hz	Priority class 1, traffic service hall
I-10	‘0’	1740Hz + 1860Hz	Toll free of charge
I-11	‘B’	1380Hz + 1980Hz	Unused
I-12	‘C’	1500Hz + 1980Hz	Unused
I-13	‘D’	1620Hz + 1980Hz	Test call
I-14	‘E’	1740Hz + 1980Hz	Unused
I-15	‘F’	1860Hz + 1980Hz	Unused

Table 26: R2 CAS China, Group I (forward) KC digits

Signal	Digit	Frequencies	
I-11	‘B’	1380Hz + 1980Hz	Priority class 1, use microwave transmission/International operator position
I-12	‘C’	1500Hz + 1980Hz	“Z” calls with specific number
I-13	‘D’	1620Hz + 1980Hz	Test call
I-14	‘E’	1740Hz + 1980Hz	Priority class 2, use superior quality cable
I-15	‘F’	1860Hz + 1980Hz	Unused

Table 27: R2 CAS China, Group I (forward) KE digits

Signal	Digit	Frequencies	Meaning
I-11	‘B’	1380Hz + 1980Hz	“H”-Tandem mark (local calls only(
I-12	‘C’	1500Hz + 1980Hz	Unused
I-13	‘D’	1620Hz + 1980Hz	Test call
I-14	‘E’	1740Hz + 1980Hz	Unused
I-15	‘F’	1860Hz + 1980Hz	Unused

Table 28: R2 CAS China, Group II (forward) KD digits

Signal	Digit	Frequencies	Meaning
II-1	‘1’	1380Hz + 1980Hz	Semi-automatic toll call from operator
II-2	‘2’	1500Hz + 1980Hz	Automatic toll call
II-3	‘3’	1620Hz + 1980Hz	Local call
II-4	‘4’	1740Hz + 1980Hz	Local data call
II-5	‘5’	1860Hz + 1980Hz	Space
II-6	‘6’	1620Hz + 1740Hz	Test call

Table 29: R2 CAS China, Group A (backward) digits

Signal	Digit	Frequencies	Meaning
A-1	‘1’	1140Hz + 1020Hz	Send next digit (n+1)
A-2	‘2’	1140Hz + 900Hz	Send first destination digit
A-3	‘3’	1020Hz + 900Hz	Send KD category, expects group B
A-4	‘4’	1140Hz + 780Hz	Congestion, abort call
A-5	‘5’	1020Hz + 780Hz	Unassigned number, abort call
A-6	‘6’	900Hz + 780Hz	Send calling party category KA, KC or KE

Table 30: R2 CAS China, Group B (backward) digits

Signal	Digit	Frequencies	Meaning
B-1	‘1’	1140Hz + 1020Hz	Subscriber line free
B-2	‘2’	1140Hz + 900Hz	Subscriber busy in a local call
B-3	‘3’	1020Hz + 900Hz	Subscriber busy in a toll call
B-4	‘4’	1140Hz + 780Hz	Congestion, abort call
B-5	‘5’	1020Hz + 780Hz	Unassigned number, abort call
B-6	‘6’	900Hz + 780Hz	Unused for toll calls, PBX line free for local calls

8.3.4.2.2 R2 Korea digits

Table 31: R2 CAS Korea, Group I (forward) digits

Signal	Digit	Frequencies	Meaning
I-1	‘1’	1380Hz + 1500Hz	Digit ‘1’
I-2	‘2’	1380Hz + 1620Hz	Digit ‘2’
I-3	‘3’	1500Hz + 1620Hz	Digit ‘3’
I-4	‘4’	1380Hz + 1740Hz	Digit ‘4’
I-5	‘5’	1500Hz + 1740Hz	Digit ‘5’
I-6	‘6’	1620Hz + 1740Hz	Digit ‘6’
I-7	‘7’	1380Hz + 1860Hz	Digit ‘7’
I-8	‘8’	1500Hz + 1860Hz	Digit ‘8’
I-9	‘9’	1620Hz + 1860Hz	Digit ‘9’
I-10	‘0’	1740Hz + 1860Hz	Digit ‘0’
I-11	‘B’	1380Hz + 1980Hz	Unused
I-12	‘C’	1500Hz + 1980Hz	No origin information is available
I-13	‘D’	1620Hz + 1980Hz	Access to test equipment
I-14	‘E’	1740Hz + 1980Hz	Unused
I-15	‘F’	1860Hz + 1980Hz	End of origin number

Table 32: R2 CAS Korea, Group II (forward) digits

Signal	Digit	Frequencies	Meaning
II-1	‘1’	1380Hz + 1500Hz	Regular subscriber without priority (default)
II-2	‘2’	1380Hz + 1620Hz	Subscriber with priority
II-3	‘3’	1500Hz + 1620Hz	Test or maintenance equipment
II-4	‘4’	1380Hz + 1740Hz	Pay telephone
II-5	‘5’	1500Hz + 1740Hz	Operator originated call
II-6	‘6’	1620Hz + 1740Hz	Subscriber using data communication equipment
II-7	‘7’	1380Hz + 1860Hz	Unused
II-8	‘8’	1500Hz + 1860Hz	Unused
II-9	‘9’	1620Hz + 1860Hz	Unused
II-10	‘0’	1740Hz + 1860Hz	Unused
II-11	‘B’	1380Hz + 1980Hz	Unused
II-12	‘C’	1500Hz + 1980Hz	Unused
II-13	‘D’	1620Hz + 1980Hz	Unused
II-14	‘E’	1740Hz + 1980Hz	Unused
II-15	‘F’	1860Hz + 1980Hz	Unused

Table 33: R2 CAS Korea, Group A (backward) digits

Signal	Digit	Frequencies	Meaning
A-1	‘1’	1140Hz + 1020Hz	Send next digit (n+1)
A-2	‘2’	1140Hz + 900Hz	Send last but one digit (n-1)
A-3	‘3’	1020Hz + 900Hz	Send KD category, expect group B
A-4	‘4’	1140Hz + 780Hz	Congestion, abort call
A-5	‘5’	1020Hz + 780Hz	Send category, origin if repeated
A-6	‘6’	900Hz + 780Hz	Address complete, connect call and charge
A-7	‘7’	1140Hz + 660Hz	Send last but two digits (n-2)
A-8	‘8’	1020Hz + 660Hz	Send last but three digits (n-3)
A-9	‘9’	900Hz + 660Hz	Send first destination digit
A-10	‘0’	780Hz + 660Hz	Unused
A-11	‘B’	1140Hz + 540Hz	Unused
A-12	‘C’	1020Hz + 540Hz	Unused
A-13	‘D’	900Hz + 540Hz	Unused
A-14	‘E’	780Hz + 540Hz	Unused
A-15	‘F’	660Hz + 540Hz	Unused

Table 34: R2 CAS Korea, Group B (backward) digits

Signal	Digit	Frequencies	Meaning
B-1	‘1’	1140Hz + 1020Hz	Called party free, last party release
B-2	‘2’	1140Hz + 900Hz	Subscriber number has changed
B-3	‘3’	1020Hz + 900Hz	Called party busy
B-4	‘4’	1140Hz + 780Hz	Congestion, abort call
B-5	‘5’	1020Hz + 780Hz	Unassigned number
B-6	‘6’	900Hz + 780Hz	Called party free, charge on answer
B-7	‘7’	1140Hz + 660Hz	Called party free, no charge
B-8	‘8’	1020Hz + 660Hz	Subscriber line out of order
B-9	‘9’	900Hz + 660Hz	Unused
B-10	‘0’	780Hz + 660Hz	Unused
B-11	‘B’	1140Hz + 540Hz	Unused
B-12	‘C’	1020Hz + 540Hz	Unused
B-13	‘D’	900Hz + 540Hz	Unused
B-14	‘E’	780Hz + 540Hz	Unused
B-15	‘F’	660Hz + 540Hz	Unused

8.3.4.2.3 R2 Singapore digits

Table 35: R2 CAS Singapore, Group I (forward) digits

Signal	Digit	Frequencies	Meaning
I-1	‘1’	1380Hz + 1500Hz	Digit ‘1’
I-2	‘2’	1380Hz + 1620Hz	Digit ‘2’
I-3	‘3’	1500Hz + 1620Hz	Digit ‘3’
I-4	‘4’	1380Hz + 1740Hz	Digit ‘4’
I-5	‘5’	1500Hz + 1740Hz	Digit ‘5’
I-6	‘6’	1620Hz + 1740Hz	Digit ‘6’
I-7	‘7’	1380Hz + 1860Hz	Digit ‘7’
I-8	‘8’	1500Hz + 1860Hz	Digit ‘8’
I-9	‘9’	1620Hz + 1860Hz	Digit ‘9’
I-10	‘0’	1740Hz + 1860Hz	Digit ‘0’
I-11	‘B’	1380Hz + 1980Hz	Access to centralized intercept service
I-12	‘C’	1500Hz + 1980Hz	Request by A-x signal not accepted
I-13	‘D’	1620Hz + 1980Hz	Unused
I-14	‘E’	1740Hz + 1980Hz	Unused
I-15	‘F’	1860Hz + 1980Hz	End of origin/destination number

Table 36: R2 CAS Singapore, Group II (forward) digits

Signal	Digit	Frequencies	Meaning
II-1	‘1’	1380Hz + 1500Hz	Operator with trunk offering facility
II-2	‘2’	1380Hz + 1620Hz	Ordinary subscriber or operator without trunk offering facility (default)
II-3	‘3’	1500Hz + 1620Hz	Payphone
II-4	‘4’	1380Hz + 1740Hz	Ex-CLI display
II-5	‘5’	1500Hz + 1740Hz	Coinafon
II-6	‘6’	1620Hz + 1740Hz	Test equipment
II-7	‘7’	1380Hz + 1860Hz	Line test desk
II-8	‘8’	1500Hz + 1860Hz	Interception operator
II-9	‘9’	1620Hz + 1860Hz	Call from transit exchange which normally does not have the calling subscriber number information (e.g. trunk/gateway)
II-10	‘0’	1740Hz + 1860Hz	Indication of a transferred call.
II-11	‘B’	1380Hz + 1980Hz	Unused
II-12	‘C’	1500Hz + 1980Hz	Unused
II-13	‘D’	1620Hz + 1980Hz	Unused
II-14	‘E’	1740Hz + 1980Hz	Unused
II-15	‘F’	1860Hz + 1980Hz	Unused

Table 37: R2 CAS Singapore, Group III (forward) digits

Signal	Digit	Frequencies	Meaning
III-1	‘1’	1380Hz + 1500Hz	Digit ‘1’
III-2	‘2’	1380Hz + 1620Hz	Digit ‘2’
III-3	‘3’	1500Hz + 1620Hz	Digit ‘3’
III-4	‘4’	1380Hz + 1740Hz	Digit ‘4’
III-5	‘5’	1500Hz + 1740Hz	Digit ‘5’
III-6	‘6’	1620Hz + 1740Hz	Digit ‘6’
III-7	‘7’	1380Hz + 1860Hz	Digit ‘7’
III-8	‘8’	1500Hz + 1860Hz	Digit ‘8’
III-9	‘9’	1620Hz + 1860Hz	Digit ‘9’
III-10	‘0’	1740Hz + 1860Hz	Digit ‘0’
III-11	‘B’	1380Hz + 1980Hz	Unused
III-12	‘C’	1500Hz + 1980Hz	CLI not available
III-13	‘D’	1620Hz + 1980Hz	Unused
III-14	‘E’	1740Hz + 1980Hz	Unused
III-15	‘F’	1860Hz + 1980Hz	End of calling number

Table 38: R2 CAS Singapore, Group A (backward) digits

Signal	Digit	Frequencies	Meaning
A-1	‘1’	1140Hz + 1020Hz	Send next digit (n+1)
A-2	‘2’	1140Hz + 900Hz	Send first destination (restart)
A-3	‘3’	1020Hz + 900Hz	Send category, expect group B
A-4	‘4’	1140Hz + 780Hz	Congestion, abort call
A-5	‘5’	1020Hz + 780Hz	Unused
A-6	‘6’	900Hz + 780Hz	Send category, origin if repeated
A-7	‘7’	1140Hz + 660Hz	Send tarif zone
A-8	‘8’	1020Hz + 660Hz	Send last but one digit (n-1)
A-9	‘9’	900Hz + 660Hz	Send last but two digits (n-2)
A-10	‘0’	780Hz + 660Hz	Unused
A-11	‘B’	1140Hz + 540Hz	Unused
A-12	‘C’	1020Hz + 540Hz	Unused
A-13	‘D’	900Hz + 540Hz	Unused
A-14	‘E’	780Hz + 540Hz	Unused
A-15	‘F’	660Hz + 540Hz	Unused

Table 39: R2 CAS Singapore, Group B (backward) digits

Signal	Digit	Frequencies	Meaning
B-1	‘1’	1140Hz + 1020Hz	Called party free, chargeable
B-2	‘2’	1140Hz + 900Hz	Called party busy
B-3	‘3’	1020Hz + 900Hz	Number requiring re-routing at the originating local exchange or outgoing exchange
B-4	‘4’	1140Hz + 780Hz	Congestion, abort call
B-5	‘5’	1020Hz + 780Hz	Called party free, non-chargeable
B-6	‘6’	900Hz + 780Hz	Last party release
B-7	‘7’	1140Hz + 660Hz	Unallocated number
B-8	‘8’	1020Hz + 660Hz	Unused
B-9	‘9’	900Hz + 660Hz	Unused
B-10	‘0’	780Hz + 660Hz	Unused
B-11	‘B’	1140Hz + 540Hz	Unused
B-12	‘C’	1020Hz + 540Hz	Unused
B-13	‘D’	900Hz + 540Hz	Unused
B-14	‘E’	780Hz + 540Hz	Unused
B-15	‘F’	660Hz + 540Hz	Unused

8.3.4.2.4 R2 Bangladesh digits

Table 40: R2 CAS Bangladesh, Group I (forward) digits

Signal	Digit	Frequencies	Meaning
I-1	‘1’	1380Hz + 1500Hz	Digit ‘1’
I-2	‘2’	1380Hz + 1620Hz	Digit ‘2’
I-3	‘3’	1500Hz + 1620Hz	Digit ‘3’
I-4	‘4’	1380Hz + 1740Hz	Digit ‘4’
I-5	‘5’	1500Hz + 1740Hz	Digit ‘5’
I-6	‘6’	1620Hz + 1740Hz	Digit ‘6’
I-7	‘7’	1380Hz + 1860Hz	Digit ‘7’
I-8	‘8’	1500Hz + 1860Hz	Digit ‘8’
I-9	‘9’	1620Hz + 1860Hz	Digit ‘9’
I-10	‘0’	1740Hz + 1860Hz	Digit ‘0’
I-11	‘B’	1380Hz + 1980Hz	Unused
I-12	‘C’	1500Hz + 1980Hz	Unused
I-13	‘D’	1620Hz + 1980Hz	Access to test equipment
I-14	‘E’	1740Hz + 1980Hz	Unused
I-15	‘F’	1860Hz + 1980Hz	End of identification, end of pulsing

Table 41: R2 CAS Bangladesh, Group II (forward) digits

Signal	Digit	Frequencies	Meaning
II-1	‘1’	1380Hz + 1500Hz	Subscriber without priority
II-2	‘2’	1380Hz + 1620Hz	Subscriber with priority
II-3	‘3’	1500Hz + 1620Hz	Maintenance equipment
II-4	‘4’	1380Hz + 1740Hz	Unused
II-5	‘5’	1500Hz + 1740Hz	Operator
II-6	‘6’	1620Hz + 1740Hz	Data transmission
II-7	‘7’	1380Hz + 1860Hz	Used for international working
II-8	‘8’	1500Hz + 1860Hz	Used for international working
II-9	‘9’	1620Hz + 1860Hz	Used for international working
II-10	‘0’	1740Hz + 1860Hz	Used for international working
II-11	‘B’	1380Hz + 1980Hz	Used for international working
II-12	‘C’	1500Hz + 1980Hz	Spare for national use
II-13	‘D’	1620Hz + 1980Hz	Spare for national use
II-14	‘E’	1740Hz + 1980Hz	Spare for national use
II-15	‘F’	1860Hz + 1980Hz	Spare for national use

Table 42: R2 CAS Bangladesh, Group A (backward) digits

Signal	Digit	Frequencies	Meaning
A-1	‘1’	1140Hz + 1020Hz	Send next digit (n+1)
A-2	‘2’	1140Hz + 900Hz	Send last but one digit (n -1)
A-3	‘3’	1020Hz + 900Hz	Send category, change to B signal
A-4	‘4’	1140Hz + 780Hz	Congestion national network
A-5	‘5’	1020Hz + 780Hz	Send calling party’s category
A-6	‘6’	900Hz + 780Hz	Address complete, charge, set-up, speech conditions
A-7	‘7’	1140Hz + 660Hz	Send last but two digits (n – 2)
A-8	‘8’	1020Hz + 660Hz	Send last but three digits (n – 3)
A-9	‘9’	900Hz + 660Hz	Send calling’s party number
A-10	‘0’	780Hz + 660Hz	Unused

Table 43: R2 CAS Bangladesh, Group B (backward) digits

Signal	Digit	Frequencies	Meaning
B-1	‘1’	1140Hz + 1020Hz	Connection under control called subscriber
B-2	‘2’	1140Hz + 900Hz	Send special information tone
B-3	‘3’	1020Hz + 900Hz	Subscriber’s line busy
B-4	‘4’	1140Hz + 780Hz	Congestion (encountered after A -> B)
B-5	‘5’	1020Hz + 780Hz	Unassigned number
B-6	‘6’	900Hz + 780Hz	Subscriber’s line free, charge
B-7	‘7’	1140Hz + 660Hz	Subscriber’s line free, no charge
B-8	‘8’	1020Hz + 660Hz	Subscriber’s line out-of-order
B-9	‘9’	900Hz + 660Hz	Unused
B-10	‘0’	780Hz + 660Hz	Unused

8.3.4.2.5 R2 Generic digits

Table 44: R2 CAS Generic, Group I (forward) digits

Signal	Digit	Frequencies	Meaning
I-1	‘1’	1380Hz + 1500Hz	Digit ‘1’
I-2	‘2’	1380Hz + 1620Hz	Digit ‘2’
I-3	‘3’	1500Hz + 1620Hz	Digit ‘3’
I-4	‘4’	1380Hz + 1740Hz	Digit ‘4’
I-5	‘5’	1500Hz + 1740Hz	Digit ‘5’
I-6	‘6’	1620Hz + 1740Hz	Digit ‘6’
I-7	‘7’	1380Hz + 1860Hz	Digit ‘7’
I-8	‘8’	1500Hz + 1860Hz	Digit ‘8’
I-9	‘9’	1620Hz + 1860Hz	Digit ‘9’
I-10	‘0’	1740Hz + 1860Hz	Digit ‘0’
I-11	‘B’	1380Hz + 1980Hz	Unused
I-12	‘C’	1500Hz + 1980Hz	No origin information available
I-13	‘D’	1620Hz + 1980Hz	Access to test equipment
I-14	‘E’	1740Hz + 1980Hz	Unused
I-15	‘F’	1860Hz + 1980Hz	End of number

Table 45: R2 CAS Generic, Group II (forward) digits

Signal	Digit	Frequencies	Meaning
II-1	‘1’	1380Hz + 1500Hz	Regular subscriber without priority
II-2	‘2’	1380Hz + 1620Hz	Subscriber with priority
II-3	‘3’	1500Hz + 1620Hz	Test or maintenance equipment
II-4	‘4’	1380Hz + 1740Hz	Unused
II-5	‘5’	1500Hz + 1740Hz	Operator originated call
II-6	‘6’	1620Hz + 1740Hz	Subscriber using data communication equipment
II-7	‘7’	1380Hz + 1860Hz	Subscriber (or operator without forward transfer capability)
II-8	‘8’	1500Hz + 1860Hz	Unused
II-9	‘9’	1620Hz + 1860Hz	Unused
II-10	‘0’	1740Hz + 1860Hz	Unused
II-11	‘B’	1380Hz + 1980Hz	Unused
II-12	‘C’	1500Hz + 1980Hz	Unused
II-13	‘D’	1620Hz + 1980Hz	Unused
II-14	‘E’	1740Hz + 1980Hz	Unused
II-15	‘F’	1860Hz + 1980Hz	Unused

Table 46: R2 CAS Generic, Group A (backward) digits

Signal	Digit	Frequencies	Meaning
A-1	‘1’	1140Hz + 1020Hz	Send next digit (n+1)
A-2	‘2’	1140Hz + 900Hz	Send last but one digit (n -1)
A-3	‘3’	1020Hz + 900Hz	Send category, expect group B signal
A-4	‘4’	1140Hz + 780Hz	Congestion, abort call
A-5	‘5’	1020Hz + 780Hz	Send category, origin if repeated
A-6	‘6’	900Hz + 780Hz	Address complete, charge, set-up, speech conditions
A-7	‘7’	1140Hz + 660Hz	Send last but two digits (n – 2)
A-8	‘8’	1020Hz + 660Hz	Send last but three digits (n – 3)

A-9	‘9’	900Hz + 660Hz	Error, abort call
A-10	‘0’	780Hz + 660Hz	Error, abort call
A-11	‘B’	1140Hz + 540Hz	Error, abort call
A-12	‘C’	1020Hz + 540Hz	Error, abort call
A-13	‘D’	900Hz + 540Hz	Error, abort call
A-14	‘E’	780Hz + 540Hz	Error, abort call
A-15	‘F’	660Hz + 540Hz	Congestion, abort call

Table 47: R2 CAS Generic, Group B (backward) digits

Signal	Digit	Frequencies	Meaning
B-1	‘1’	1140Hz + 1020Hz	Called party free, answer and connect call
B-2	‘2’	1140Hz + 900Hz	Number unreachable, abort call
B-3	‘3’	1020Hz + 900Hz	Called party busy, abort call
B-4	‘4’	1140Hz + 780Hz	Congestion, abort call
B-5	‘5’	1020Hz + 780Hz	Unassigned number, abort call
B-6	‘6’	900Hz + 780Hz	Called party free, answer and connect call
B-7	‘7’	1140Hz + 660Hz	Called party free, answer and connect call
B-8	‘8’	1020Hz + 660Hz	Error, abort call
B-9	‘9’	900Hz + 660Hz	Error, abort call
B-10	‘0’	780Hz + 660Hz	Error, abort call
B-11	‘B’	1140Hz + 540Hz	Error, abort call
B-12	‘C’	1020Hz + 540Hz	Error, abort call
B-13	‘D’	900Hz + 540Hz	Error, abort call
B-14	‘E’	780Hz + 540Hz	Error, abort call
B-15	‘F’	660Hz + 540Hz	Error, abort call

8.3.5 PRI CAS Call scenarios (Stack and TB640 APIs)

The following call scenarios show the different CAS API message exchange required to establish a call. Every table in the following sections are divided in four columns. The leftmost and rightmost columns describe the actual TB640 API messages that are sent and received by the user applications (both user and network side are shown). The two middle columns represent the ABCD bits and tones exchanged by the CAS stacks. This information is never seen by neither user applications. The reader must remember that each request from the user application is answered individually by the TB640 board (not the stack) to confirm they all have been delivered to the signaling stack. Those board "responses" are not shown in the call scenarios.

8.3.5.1 R1 CAS (except Taiwan modified R1)

8.3.5.1.1 Successful call placed from network-side (non Direct-Inward-Dialing)

Network-side API prim	Physical Network-side	Physical User-side	User-side API prim
CMD_CONNECT_REQUEST->			
	ABCD bits exchanges		
			NOTIF_CONNECT_INDICATION->
	Tone (digit #1)->		
			NOTIF_KEYPAD_INDICATION->
	Tone (digit #x)->		
			NOTIF_KEYPAD_INDICATION->
	Tone (final digit)->		
			NOTIF_KEYPAD_INDICATION->
			<- CMD_ACCEPT_INCOMING_CALL (optional). <if sent, the user application can use the timeslot when the response is received>
		←ABCD bits	<- CMD_CONNECT_RESPONSE
<- `N_CONN_CF`			<user application can use the timeslot when the response is received>
<user application can use the timeslot>			

8.3.5.1.2 Successful call placed from network-side (Direct-Inward-Dialing)

Network-side API prim	Physical Network-side	Physical User-side	User-side API prim
CMD_CONNECT_REQUEST->			
	ABCD bits exchanges		
			NOTIF_CONNECT_INDICATION-> <user application can use the timeslot >
		←ABCD bits	<- CMD_CONNECT_RESPONSE
<- `N_CONN_CF`			
<user application can use the timeslot>			

8.3.5.1.3 Successful call placed from user-side (non Direct-Inward-Dialing)

Network-side API prim	Physical Network-side	Physical User-side	User-side API prim
			<-CMD_CONNECT_REQUEST
	ABCD bits exchanges		
<-NOTIF_CALL_PRESENT_INDICATION			
		<- Tone (digit #1)	
<-NOTIF_KEYPAD_INDICATION			
		<- Tone (digit #x)	
<-NOTIF_KEYPAD_INDICATION			
		<- Tone (final digit)	
<-NOTIF_KEYPAD_INDICATION			
CMD_ACCEPT_INCOMING_CALL -> (optional). <if sent, the user application can use the timeslot when the response is received			
CMD_CONNECT_RESPONSE->	ABCD bits →		
<user application can use the timeslot when the response is received>			`N_CONN_CF->`
			<user application can use the timeslot>

8.3.5.1.4 Successful call placed from user-side (Direct-Inward-Dialing)

Network-side API prim	Physical Network-side	Physical User-side	User-side API prim
			<-CMD_CONNECT_REQUEST
	ABCD bits exchanges		
<-NOTIF_CALL_PRESENT_INDICATION <user application can use the timeslot>			
CMD_CONNECT_RESPONSE->	ABCD bits →		
			`N_CONN_CF->`
			<user application can use the timeslot>

8.3.5.1.5 Call refused to network/user-side (timeslot busy or physical line down)

Network-side API prim	Physical Network-side	Physical User-side	User-side API prim
CMD_CONNECT_REQUEST->			
<- NOTIF_STATUS_INDICATION (VALUE_DISPLAY_INFO)			
<- NOTIF_DISCONNECT_INDICATION			
CMD_RELEASE->			
			<- CMD_CONNECT_REQUEST
			NOTIF_STATUS_INDICATION (VALUE_DISPLAY_INFO) ->
			NOTIF_DISCONNECT_INDICATION->
			<-CMD_RELEASE

8.3.5.1.6 Call stopped to network/user-side because of protocol error

Network-side API prim	Physical Network-side	Physical User-side	User-side API prim
CMD_CONNECT_REQUEST->			
	ABCD bits exchanges		
<- NOTIF_DISCONNECT_INDICATION			
CMD_RELEASE->			
			<- CMD_CONNECT_REQUEST
	ABCD bits exchanges		
			NOTIF_DISCONNECT_INDICATION->
			<-CMD_RELEASE

8.3.5.1.7 Call refused by user-side (non Direct-Inward-Dialing)

Network-side API prim	Physical Network-side	Physical User-side	User-side API prim
CMD_CONNECT_REQUEST->			
	ABCD bits exchanges		
			NOTIF_CONNECT_INDICATION->
	Tone (digit #1)->		
			NOTIF_KEYPAD_INDICATION->
	Tone (digit #x)->		
			NOTIF_KEYPAD_INDICATION->
			<user application does not need to wait for all digits to be received before calling CMD_DISCONNECT_REQUEST >
		←ABCD bits	<- CMD_DISCONNECT_REQUEST
<- NOTIF_DISCONNECT_INDICATION	ABCD bits→		
<user application MUST disconnect the timeslot if it was connected>			NOTIF_DISCONNECT_CONFIRM->
CMD_RELEASE->			<-CMD_RELEASE

8.3.5.1.8 Call refused by user-side (Direct-Inward-Dialing)

Network-side API prim	**Physical Network-side**	**Physical User-side**	**User-side API prim**
CMD_CONNECT_REQUEST->			
	ABCD bits exchanges		
			NOTIF_CONNECT_INDICATION->
		←ABCD bits	<- CMD_DISCONNECT_REQUEST
<- NOTIF_DISCONNECT_INDICATION	ABCD bits→		
<user application MUST disconnect the timeslot if it was connected>			NOTIF_DISCONNECT_CONFIRM->
CMD_RELEASE->			<-CMD_RELEASE

8.3.5.1.9 Call refused by network-side (non Direct-Inward-Dialing)

Network-side API prim	**Physical Network-side**	**Physical User-side**	**User-side API prim**
			<-CMD_CONNECT_REQUEST
	ABCD bits exchanges		
<-NOTIF_CALL_PRESENT_INDICATION			
		<-Tone (digit #1)	
<-NOTIF_KEYPAD_INDICATION			
		<- Tone (digit #x)	
<-NOTIF_KEYPAD_INDICATION			
<user application does not need to wait for all digits to be received before calling CMD_CONNECT_RESPONSE>			
CMD_DISCONNECT_REQUEST->	ABCD bits→		
		←ABCD bits	NOTIF_DISCONNECT_INDICATION->
<- NOTIF_DISCONNECT_CONFIRM			<user application MUST disconnect the timeslot if it was connected>
CMD_RELEASE->			<-CMD_RELEASE

8.3.5.1.10 Call refused by network-side (Direct-Inward-Dialing)

Network-side API prim	**Physical Network-side**	**Physical User-side**	**User-side API prim**
			<-CMD_CONNECT_REQUEST
	ABCD bits exchanges		
<-NOTIF_CALL_PRESENT_INDICATION			
CMD_DISCONNECT_REQUEST->	ABCD bits→		
		←ABCD bits	NOTIF_DISCONNECT_INDICATION->
<- NOTIF_DISCONNECT_CONFIRM			<user application MUST disconnect the timeslot if it was connected>
CMD_RELEASE->			<-CMD_RELEASE

8.3.5.1.11 Call cleared by network

Network-side API prim	**Physical Network-side**	**Physical User-side**	**User-side API prim**
Call already established			
CMD_DISCONNECT_REQUEST->	ABCD bits→		
		← ABCD bits	NOTIF_DISCONNECT_INDICATION->
<- NOTIF_DISCONNECT_CONFIRM			<user application MUST disconnect the timeslot if it was connected>
<user application MUST disconnect the timeslot if it was connected>			<-CMD_RELEASE
CMD_RELEASE->			

8.3.5.1.12 Call cleared by/user-side

Network-side API prim	**Physical Network-side**	**Physical User-side**	**User-side API prim**
Call already established			
		←ABCD bits	<-CMD_DISCONNECT_REQUEST
<-NOTIF_DISCONNECT_INDICATION	ABCD bits→		
<user application MUST disconnect the timeslot if it was connected>			NOTIF_DISCONNECT_CONFIRM->
CMD_RELEASE->			<user application MUST disconnect the timeslot if it was connected>
			<-CMD_RELEASE

8.3.5.1.13 Disconnect collision (both sides disconnect at the same time)

Network-side API prim	**Physical Network-side**	**Physical User-side**	**User-side API prim**
Call already established			
CMD_DISCONNECT_REQUEST->	ABCD bits →	←ABCD bits	<-CMD_DISCONNECT_REQUEST
<-NOTIF_DISCONNECT_CONFIRM			NOTIF_DISCONNECT_CONFIRM->
<user application MUST disconnect the timeslot if it was connected>			<user application MUST disconnect the timeslot if it was connected>
CMD_RELEASE->			<-CMD_RELEASE

8.3.5.1.14 Disconnect collision (disconnect almost at the same time)

Network-side API prim	**Physical Network-side**	**User-side**	**User-side API prim**
Call already established			
		←ABCD bits	<-CMD_DISCONNECT_REQUEST
CMD_DISCONNECT_REQUEST->	ABCD bits →		
<-NOTIF_DISCONNECT_INDICATION			NOTIF_DISCONNECT_CONFIRM->
<user application MUST disconnect the timeslot if it was connected>			<user application MUST disconnect the timeslot if it was connected>
CMD_RELEASE->			<-CMD_RELEASE
<response to CMD_DISCONNECT_REQUEST will be an error code)			

8.3.5.2 Taiwan modified R1

Within this section, we no longer refer to the ‘network’ nor ‘user’ sides as the state machine only depends on which side initiated the call. ‘Forward-side’ refers to the originating side and ‘backward-side’ refers to the receiving side. The same Taiwan modified R1 CAS stack can be forward and backward at the same time on different timeslots.

8.3.5.2.1 Successful call placed from forward-side

Forward-side API prim	**Physical Forward-side**	**Physical Backward-side**	**Backward-side API prim**
CMD_CONNECT_REQUEST->			
	ABCD bits exchanges		
	MRF1 tone exchanges		
<- NOTIF_STATUS_IND (MDR1_DIALING_DONE with status = 0x01)			NOTIF_CONNECT_INDICATION->
		←ABCD bits	<- CMD_CONNECT_RESPONSE
<- N_CONN_CF			<user application can use the timeslot when the response is received>
<user application can use the timeslot>			

8.3.5.2.2 Call refused to forward-side (timeslot busy or physical line down)

Forward-side API prim	**Physical Forward-side**	**Physical Backward-side**	**Backward-side API prim**
CMD_CONNECT_REQUEST->			
<- NOTIF_DISCONNECT_INDICATION			
CMD_RELEASE->			

8.3.5.2.3 Call stopped to forward-side because of protocol error

Forward-side API prim	**Physical Forward-side**	**Physical Backward-side**	**Backward-side API prim**
CMD_CONNECT_REQUEST->			
	ABCD bits exchanges		
	MRF1 tone exchanges		
<- NOTIF_STATUS_IND (MDR1_DIALING_DONE with status = 0x05			
<- NOTIF_DISCONNECT_INDICATION			
CMD_RELEASE->			

8.3.5.2.4 Call refused by backward-side

Forward-side API prim	**Physical Forward-side**	**Physical Backward-side**	**Backward-side API prim**
CMD_CONNECT_REQUEST->			
	ABCD bits exchanges		
	MRF1 tone exchanges		
<- NOTIF_STATUS_IND (MDR1_DIALING_DONE with status = 0x01)			NOTIF_CONNECT_INDICATION->
		←ABCD bits	<- CMD_DISCONNECT_REQUEST
<- NOTIF_DISCONNECT_INDICATION	ABCD bits →		
CMD_RELEASE->			NOTIF_DISCONNECT_CONFIRM ->
			<-CMD_RELEASE

8.3.5.2.5 Call cleared by forward -side

Forward-side API prim	Physical Forward-side	Physical Backward-side	Backward-side API prim
Call already established			
CMD_DISCONNECT_REQUEST->	ABCD bits→		
		← ABCD bits	NOTIF_DISCONNECT_INDICATION->
<- NOTIF_DISCONNECT_CONFIRM			<user application MUST disconnect the timeslot if it was connected>
<user application MUST disconnect the timeslot if it was connected>			<-CMD_RELEASE
CMD_RELEASE->			

8.3.5.2.6 Call cleared by backward-side

Forward-side API prim	Physical Forward-side	Physical Backward-side	Backward-side API prim
Call already established			
		←ABCD bits	<-CMD_DISCONNECT_REQUEST
<-NOTIF_DISCONNECT_INDICATION	ABCD bits→		
<user application MUST disconnect the timeslot if it was connected>			NOTIF_DISCONNECT_CONFIRM->
CMD_RELEASE->			<user application MUST disconnect the timeslot if it was connected>
			<-CMD_RELEASE

8.3.5.2.7 Disconnect collision (both sides disconnect at the same time)

Forward-side API prim	Physical Forward-side	Backward-side	Backward-side API prim
Call already established			
CMD_DISCONNECT_REQUEST->	ABCD bits →	←ABCD bits	<-CMD_DISCONNECT_REQUEST
<-NOTIF_DISCONNECT_CONFIRM			NOTIF_DISCONNECT_CONFIRM->
<user application MUST disconnect the timeslot if it was connected>			<user application MUST disconnect the timeslot if it was connected>
CMD_RELEASE->			<-CMD_RELEASE

8.3.5.2.8 Disconnect collision (disconnect almost at the same time)

Forward-side API prim	Physical Forward-side	Physical Backward-side	Backward-side API prim
Call already established			
		←ABCD bits	<-CMD_DISCONNECT_REQUEST
CMD_DISCONNECT_REQUEST->	ABCD bits →		
<-NOTIF_DISCONNECT_INDICATION			NOTIF_DISCONNECT_CONFIRM->
<user application MUST disconnect the timeslot if it was connected>			<user application MUST disconnect the timeslot if it was connected>
CMD_RELEASE->			<-CMD_RELEASE
<response to CMD_DISCONNECT_REQUEST will be an error code)			

8.3.5.3 R2 CAS

Within this section, we no longer refer to the 'network' nor 'user' sides as the state machine only depends on which side initiated the call. 'Forward-side' refers to the originating side and 'backward-side' refers to the receiving side. The same R2 CAS stack can be forward and backward at the same time on different timeslots.

8.3.5.3.1 Successful call placed from forward-side

Forward-side API prim	**Physical Forward-side**	**Physical Backward-side**	
CMD_CONNECT_REQUEST->			
	ABCD bits exchanges		
	Tone exchange starting from forward-side answered by backward-side		
			NOTIF_CONNECT_INDICATION->
			<- CMD_CONNECT_RESPONSE (with a group B 'accept' value)
		←Group B tone	
<- NOTIF_STATUS_INDICATION (VALUE_CAS_R2_DIALING_DONE) <result code is 0x01 or 0x02> + <group B digit>		←ABCD bits	
			-> NOTIF_STATUS_INDICATION (VALUE_CAS_R2_DIALING_DONE)
<- `N_CONN_CF`			<user application can use the timeslot when the response is received>
<user application can use the timeslot>			

8.3.5.3.2 Call refused to forward-side (timeslot busy or physical line down)

Forward-side API prim	**Physical Forward-side**	**Backward-side**	**Backward-side API prim**
CMD_CONNECT_REQUEST->			
<- NOTIF_STATUS_INDICATION (VALUE_DISPLAY_INFO)			
<- NOTIF_DISCONNECT_INDICATION			
CMD_RELEASE->			

8.3.5.3.3 Call stopped to forward-side because of protocol error/network congestions or busy condition

Forward-side API prim	**Physical Forward-side**	**Physical Backward-side**	**Backward-side API prim**
CMD_CONNECT_REQUEST->			
	ABCD bits exchanges		
	Tone exchange starting from forward-side answered by backward-side		
<-NOTIF_STATUS_INDICATION (VALUE_CAS_R2_DIALING_DONE) <result code is 0x3, 0x4 or 0x5>			
<- NOTIF_DISCONNECT_INDICATION			
CMD_RELEASE->			

8.3.5.3.4 Call refused by backward-side

Forward-side API prim	Physical Forward-side	Physical Backward-side	Backward-side API prim
CMD_CONNECT_REQUEST->			
	ABCD bits exchanges		
	Tone exchange starting from forward-side answered by backward-side		
			NOTIF_CONNECT_INDICATION->
			<- CMD_DISCONNECT_REQUEST (with a group B 'refusal' value)
		←Group B tone	
<- NOTIF_STATUS_INDICATION (VALUE_CAS_R2_DIALING_DONE) <result code 0x03, 0x04 or 0x05> + <group B digit>			
CMD_DISCONNECT_REQUEST ->	ABCD bits →		
			-> NOTIF_DISCONNECT_CONFIRM
<- NOTIF_DISCONNECT_CONFIRM			
CMD_RELEASE->			<-CMD_RELEASE

8.3.5.3.5 Call cleared by forward-side

Forward-side API prim	Physical Forward-side	Physical Backward-side	Backward-side API prim
Call already established			
CMD_DISCONNECT_REQUEST->	ABCD bits→		
		← ABCD bits	NOTIF_DISCONNECT_INDICATION->
<- NOTIF_DISCONNECT_CONFIRM			<user application MUST disconnect the timeslot if it was connected>
<user application MUST disconnect the timeslot if it was connected>			<-CMD_RELEASE
CMD_RELEASE->			

8.3.5.3.6 Call cleared by backward-side

Forward-side API prim	Physical Forward-side	Physical Backward-side	
Call already established			
		←ABCD bits	<-CMD_DISCONNECT_REQUEST
<-NOTIF_DISCONNECT_INDICATION	ABCD bits→		
<user application MUST disconnect the timeslot if it was connected>			NOTIF_DISCONNECT_CONFIRM->
CMD_RELEASE->			<user application MUST disconnect the timeslot if it was connected>
			<-CMD_RELEASE

8.3.5.3.7 Disconnect collision (both sides disconnect at the same time)

Forward-side API prim	**Physical Forward-side**	**Physical Backward-side**	**Backward-side API prim**
Call already established			
CMD_DISCONNECT_REQUEST->	ABCD bits →	←ABCD bits	<-CMD_DISCONNECT_REQUEST
<-NOTIF_DISCONNECT_CONFIRM			NOTIF_DISCONNECT_CONFIRM->
<user application MUST disconnect the timeslot if it was connected>			<user application MUST disconnect the timeslot if it was connected>
CMD_RELEASE->			<-CMD_RELEASE

8.3.5.3.8 Disconnect collision (disconnect almost at the same time)

Forward-side API prim	**Forward-side**	**Physical Backward-side**	**Backward-side API prim**
Call already established			
		←ABCD bits	<-CMD_DISCONNECT_REQUEST
CMD_DISCONNECT_REQUEST->	ABCD bits →		
<-NOTIF_DISCONNECT_INDICATION			NOTIF_DISCONNECT_CONFIRM->
<user application MUST disconnect the timeslot if it was connected>			<user application MUST disconnect the timeslot if it was connected>
CMD_RELEASE->			<-CMD_RELEASE
<response to CMD_DISCONNECT_REQUEST will be an error code)			

9 CLOCK CONFIGURATIONS

The TB640 adapter can receive its clock from a trunk, a network reference, the H.110 bus, the Multi-Blades Link port or the local oscillator. The TB-MB adapter can receive its clock from a network reference, the H.110 bus, a Multi-Blades Link port, a BITS port or the local oscillator. For details concerning the Multi-Blades system clock synchronization, refer to section 13.2 (Blades synchronization).

9.1 Clocking description

The application must first select the local references (LocalRef1 and LocalRef2) from different clock reference source types. The active local reference will be used to drive the on-board clock, the network reference and the H.110 clocks. The TB640 and the TB-MB support clock fallback. When the primary clock reference (LocalRef1) fails and local reference fallback is set to true, the secondary clock reference (LocalRef2) will be selected as current clock reference and an event will be sent to the host application. The host application can then reconfigure clock and set a new primary clock reference or restore the primary clock reference and get ready to handle secondary clock reference failure. An option in the clock configuration lets host application decide to switch back from secondary to primary automatically or manually.
The local clock reference of a blade can be shared with other blades via H.110 bus signal if in the same cPCI chassis or via MBL ports if blades have MBL capability option. Refer to section 13.2 (Blades synchronization) for details concerning the Multi-Blades system clock synchronization via MBL ports. By example, when using H.110 signal to share clock reference, a second blade can drive CT8B while the first is driving CT8A. The second blade could have CT8A as primary clock reference and select a local secondary clock source with fallback enabled. All other cards in a cPCI system should slave on the same H.110 clock (either CT8A or CT8B) driven by the primary adapter in the system.
The status of the H.110 clock signals can be seen using the TB640_MSG_CLK_SYNC_STATES_GET message and will tell the host application if CT8A, CT8B, NetRef1 and NetRef2 are driven by at least one adapter. It does not cover the case for multiple adapters driving the H.110 bus master clock. TelcoBridges has not seen any implementation of the NetRef1 and NetRef2 signals and suggest using default values whenever they need to be configured.
When configuring a trunk, you have the option of setting the fLooptime parameter (TB640_TRUNK_CFG). If set to TRUE, this has for effect to take the receive clock from the trunk and use it as the transmit clock (instead of using the internal clock of the TB640). This can be useful to remove undesirable error alarms when the clocks from different sources are not synchronized. In general, the fLooptime parameter should be set to FALSE.

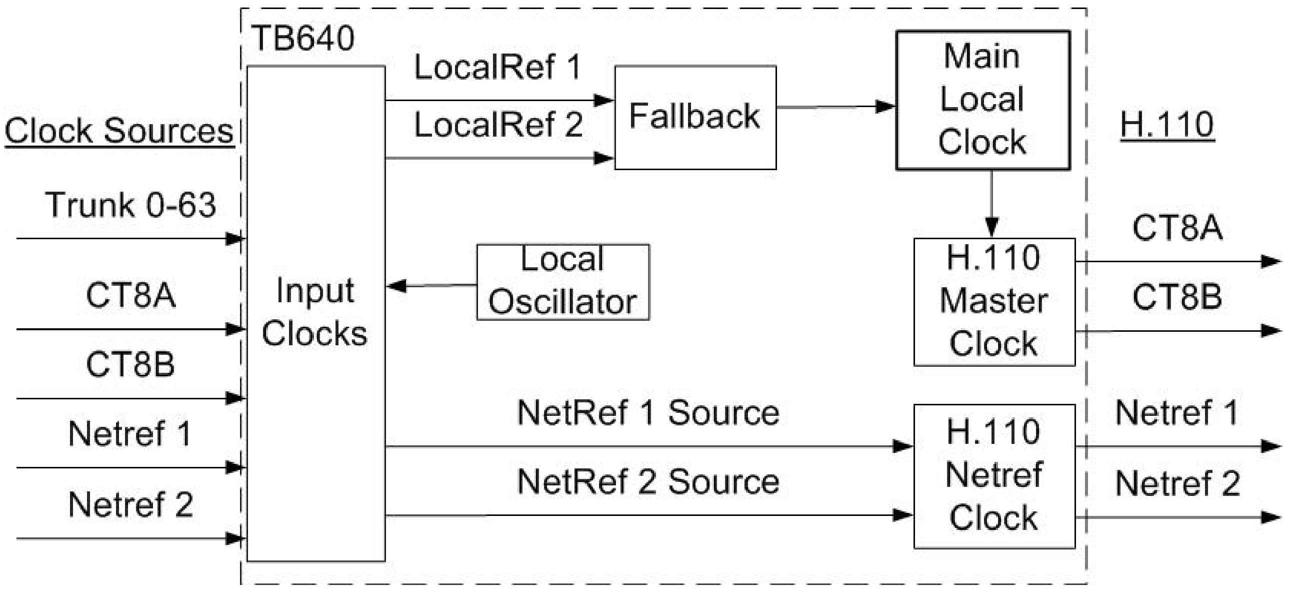
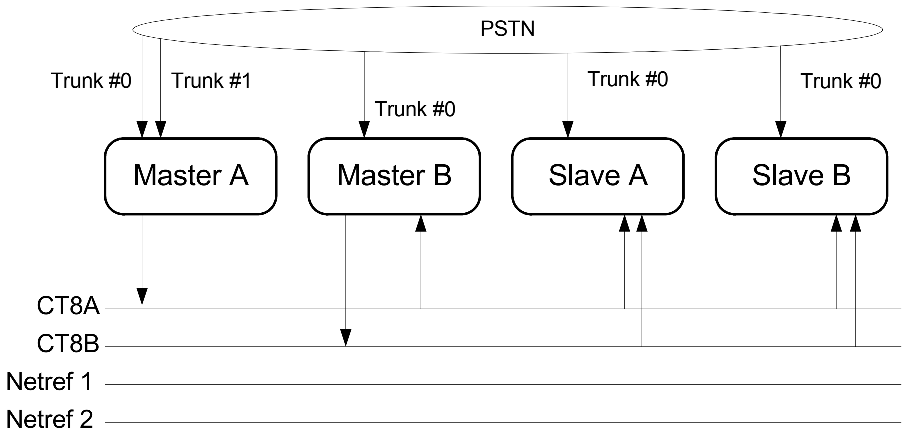


Figure 65: TB640 internal clock configuration

Common configurations are shown below. The tb640clock tool can be used to set the H.110 clock of any adapters in the system.

9.2 Primary, Secondary Master and Slave



	Master A	Master B	Slave A	Slave B
Local Ref 1	Trunk #0	CT8A	CT8A	CT8A
Local Ref 2	Trunk #1	Trunk#0	CT8B	CT8B
Fallback	True	True	True	True
Master Clock src	Local Ref 1	Local Ref 1	-	-
Master Clock	CT8A	CT8B	-	-
Init Master Role	Primary	Secondary	-	-
Drive Netref 1	False	False	False	False
Netref 1 src	-	-	-	-
Drive Netref 2	False	False	False	False
Netref 2 src	-	-	-	-

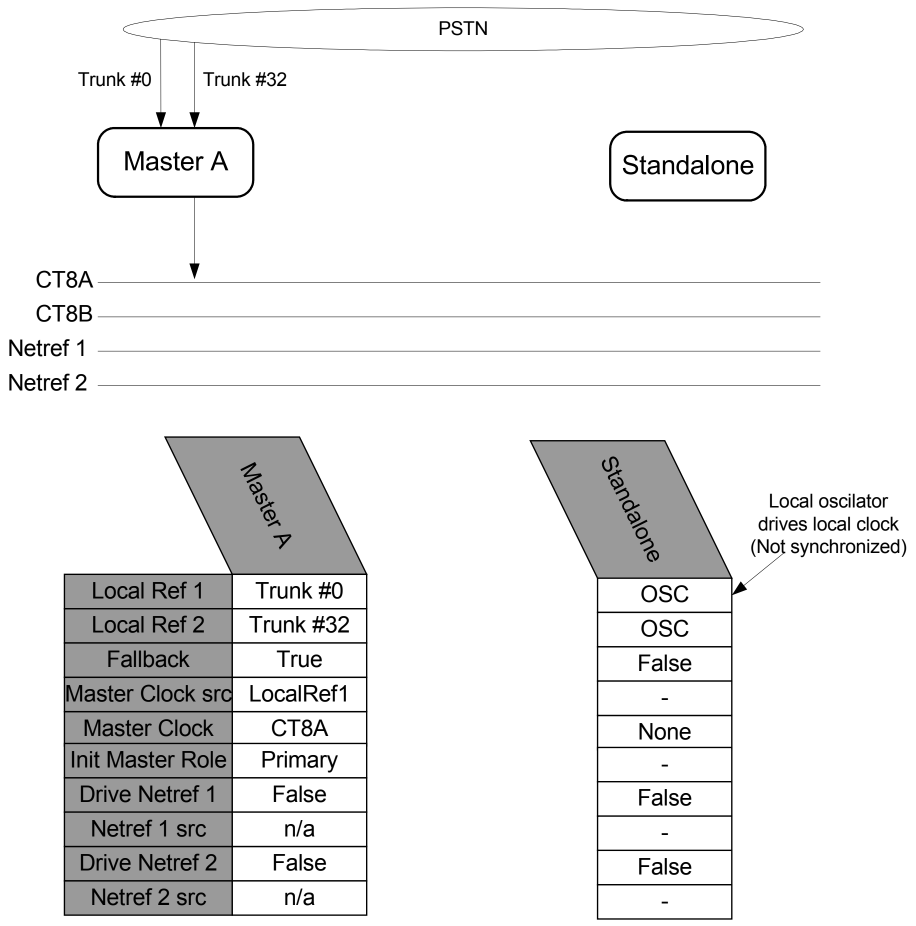
Master A Drives CT8A, Source from Trunk #0, Fallback to Trunk#1
Master B Drives CT8B, Source from CT8A, fallback to Trunk#0
Slave A, Slave to CT8A, fallback to CT8B
Slave B, Slave to CT8A, fallback to CT8B,

Figure 66: Master clock configuration

This configuration shows a primary master driven from two trunks (#0 and #1) and a secondary master ready to take over from his own trunk #0 if Master A fails.

9.3 *Primary Master and Standalone*

This configuration shows a primary master driven from two trunks (#0 and #32) and a standalone, taking its clocks from the local oscillators.



	Master A	Standalone
Local Ref 1	Trunk #0	OSC
Local Ref 2	Trunk #32	OSC
Fallback	True	False
Master Clock src	LocalRef1	-
Master Clock	CT8A	None
Init Master Role	Primary	-
Drive Netref 1	False	False
Netref 1 src	n/a	-
Drive Netref 2	False	False
Netref 2 src	n/a	-

Master A Drives CT8A, Source from Trunk #0, fallback to Trunk #32
Standalone, Local oscillator drives local clocks (does not drive/slave to H.110 bus)

Figure 67: Master and standalone configuration

10 DS3 AND STM-1 CLOCK CONFIGURATIONS

On TB-DS3 and TB-STM1 boards, the clocks are generated using a holdover chip. This chip generates clocks that are locked and synchronized on clock reference (LocalRef1). Note that no secondary clock reference (LocalRef2) is available on these kinds of board. When configuring clock, the LocalRef2 parameter must be set to TB640_CLK_SRC_NONE and fallback indication must be set to TBX_FALSE.

When clock reference is lost, the TB-DS3 and TB-STM1 clocks stay synchronized to previous clock reference and will slowly drift from its original position and eventually will get desynchronized to previous clock reference. When clock reference is lost, application gets notified of the event. The holdover chip gives time to application to reconfigure clock and reduce the effect of losing the clock reference.

On TB-DS3, the clock reference could come from a trunk, a network reference, the H.110 bus, the Multi-Blades Link port or the local oscillator. On TB-STM1, the clock reference could come from a trunk, a SONET_SDH line interface, a network reference, the H.110 bus, the Multi-Blades Link port or the local oscillator.

For SONET_SDH line interface configuration, take care of the loop time setting (fLoopTime parameter). The SONET_SDH payload is floating. I mean there is payload pointer adjustment that allows the SONET_SDH envelop and payload to be unsynchronized. If the clock reference is a trunk from a SONET_SDH line interface then loop time should be enabled to make transmission in sync with SONET_SDH reception. If the clock reference is a SONET_SDH line interface then it possible to use the retrieved clock reference for transmission but it is recommended to enable the SONET_SDH interface loop time setting. In resume, the SONET_SDH interface loop time setting should be set to TBX_TRUE for all cases except when we are master clock reference.

11 ADAPTER MANAGEMENT

11.1 Resetting the adapter

The TB640 adapter can be reset in many ways. Resetting the TB640 will restart the board local processors and reload the software for the “next boot directory” (can be checked using install.exe tool).

- Send a message using the API
  - Send the “TB640_MSG_ADAPTER_OP_RESTART” message with the adapter handle will reset the adapter
- Install tool
  - Option [99]
  - In fact, this tool is using the message “Restart”
- push button
  - At this time, the “sw2” push button on the front panel resets the TB640 adapter
- In-chassis CPU reset
  - The host CPU in the cPCI chassis must generate a Reset on the cPCI chassis backplane (if a host CPU is present in the system)
- Hot swap
- Power down/up the cPCI chassis

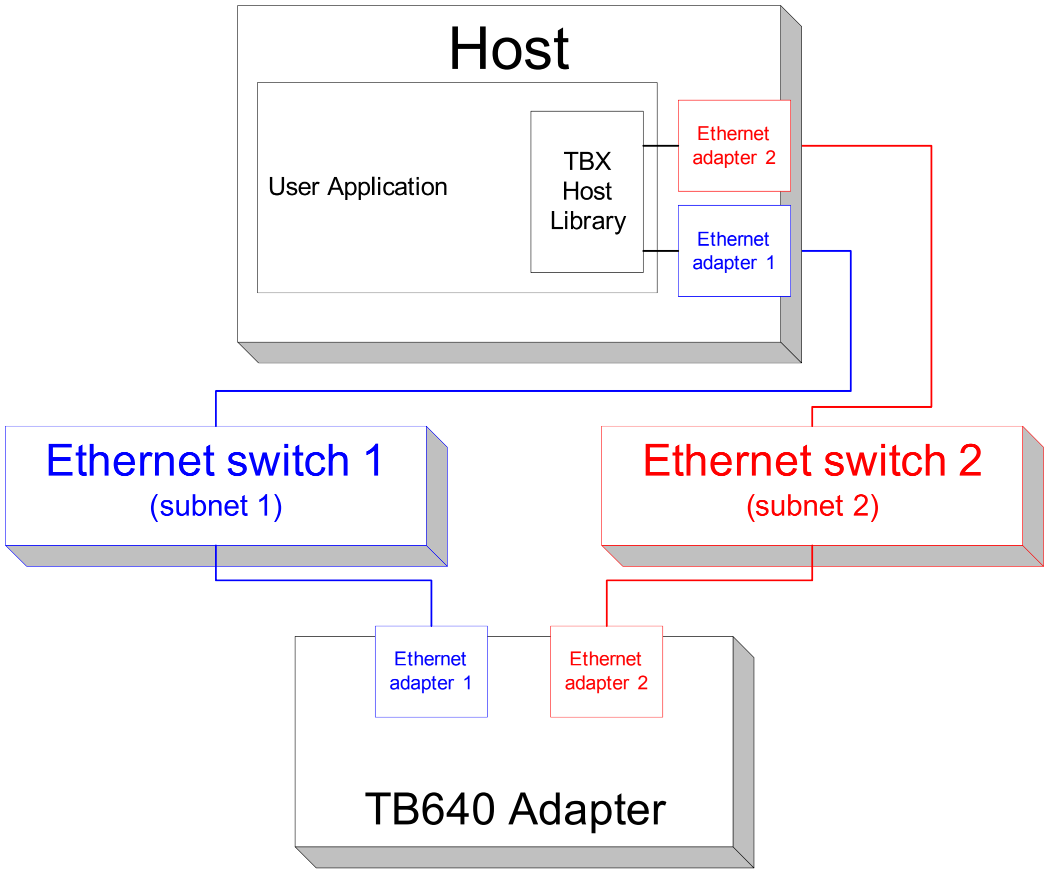
12 FAULT TOLERANCE AND HIGH AVAILABILITY

12.1 Network redundancy

The TB640 adapter supports network redundancy. Each TB640 adapter has two Ethernet ports, each with a separate IP address that can be on the same subnet, or on different subnets. The TBX Host library, used by applications that control the TB640 adapter, connects by default to an adapter using network redundancy mode (using both Ethernet ports). To disable this function or change network redundancy parameters, the application can call the `TBXConfigureNetworkRedundancy` API call.

In network redundancy mode, the host library connects to each of the adapter's Ethernet ports. If network connection with one of the ports is broken (unplugged or failed cable, failed Ethernet switch, failed network), the application will continue to control the adapter seamlessly. You may have a 2 seconds delay (configurable) in the communication while the failure is detected, but no messages will be lost. The application will be notified of the Ethernet port failure by an event of type TBX_MSG_ID_API_NOTIF_ADAPTER_ETH_DOWN, but don't need to take any specific action since everything will continue to work properly. There will be no loss of anything: messages, calls or states. When the failed connection is re-established, the application will be notified by an event of type TBX_MSG_ID_API_NOTIF_ADAPTER_ETH_UP. Again, no specific action is required, everything will continue to work properly. At any time, the application can call the function TBXGetNetworkRedundancyState to query the network redundancy state.

We highly recommend using network redundancy through two separate networks (thus using a host that has two network adapters connected to separate subnets). Duplicating all elements along the path between the host and the adapter protects against the failure of any element (cable, switch, Ethernet adapter) in one of the two redundant networks. Otherwise, the failure of any element common to the network path between the host and each of the adapter's Ethernet ports will cause the application to be disconnected from the adapter. For example, the TB640 adapter supports having its two Ethernet ports on the same subnet through one or two separate switches, but this scheme will be redundant only in regard to the Ethernet cables (and the switches if two separate switches are used). Thus we highly recommend using two separate networks.



12.2 NP1 facility protection

12.2.1 Overview

TelcoBridges supports board level redundancy also called N+1 (NP1) redundancy scheme. Before describing this protection scheme, let first define a Facility as a TB640 adapter and its corresponding RPIO (Rear Panel IO). To achieve NP1 protection, N primary facilities are connected to NP1IO boards in a NP1 chassis, with one redundant facility, ready to take over one of the primary facility.

The TelcoBridges NP1 solution protects N facilities against equipment failure. It is the responsibility of the user application to monitor adapter faults (see 12.3 Fault Detection) and perform the switch over sequence of the faulty primary facility to the redundant facility. This sequence should be the following, first, reprogramming all the resources of the failed adapter to the redundant adapter and, only after this step, sending a *takeover* operation message API to the redundant facility telling which primary facility to protect. The NP1 solution also permits live adapter software update by the host application.

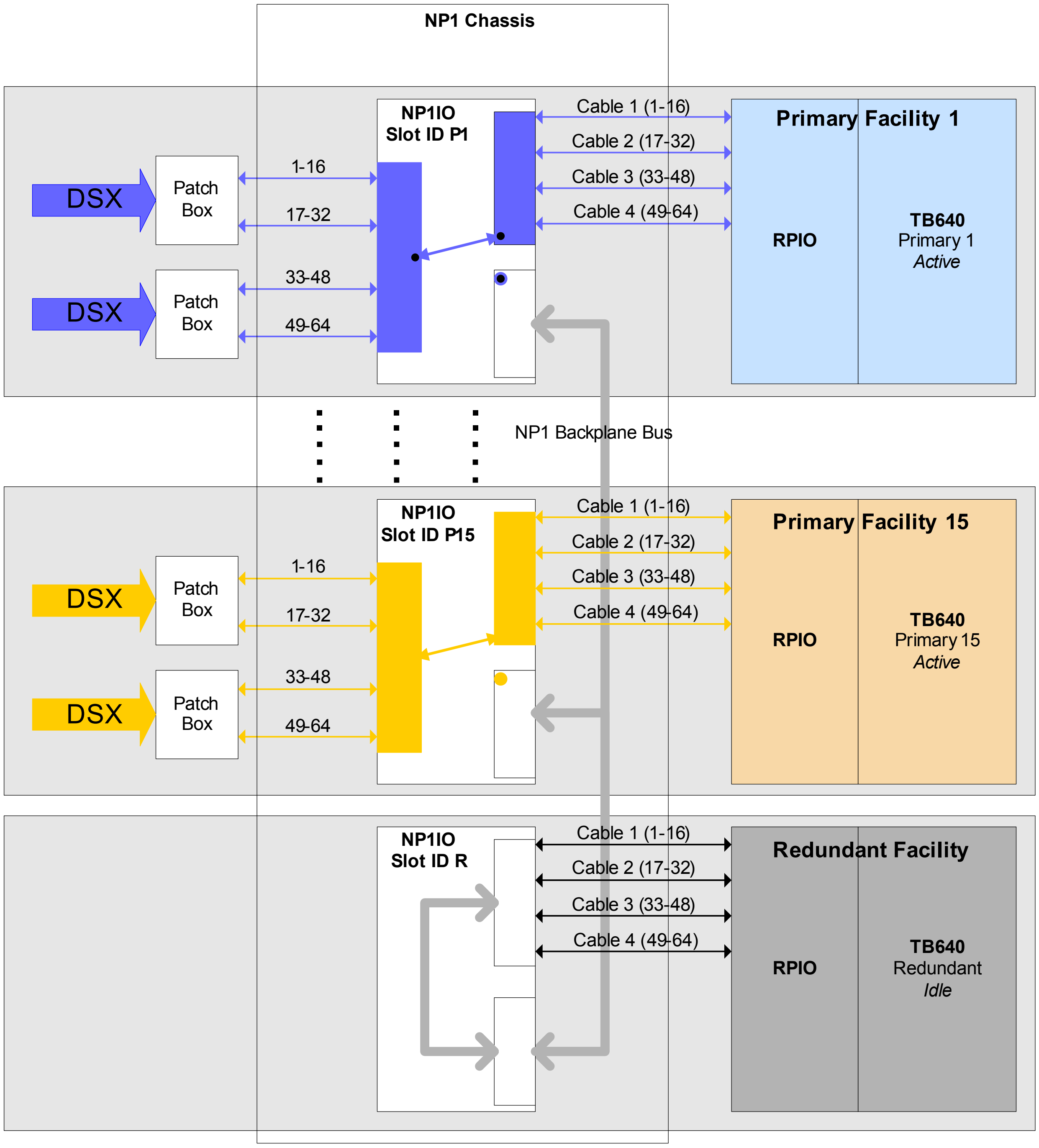


Figure 68: NP1 normal primary facility traffic flow – Redundant Released

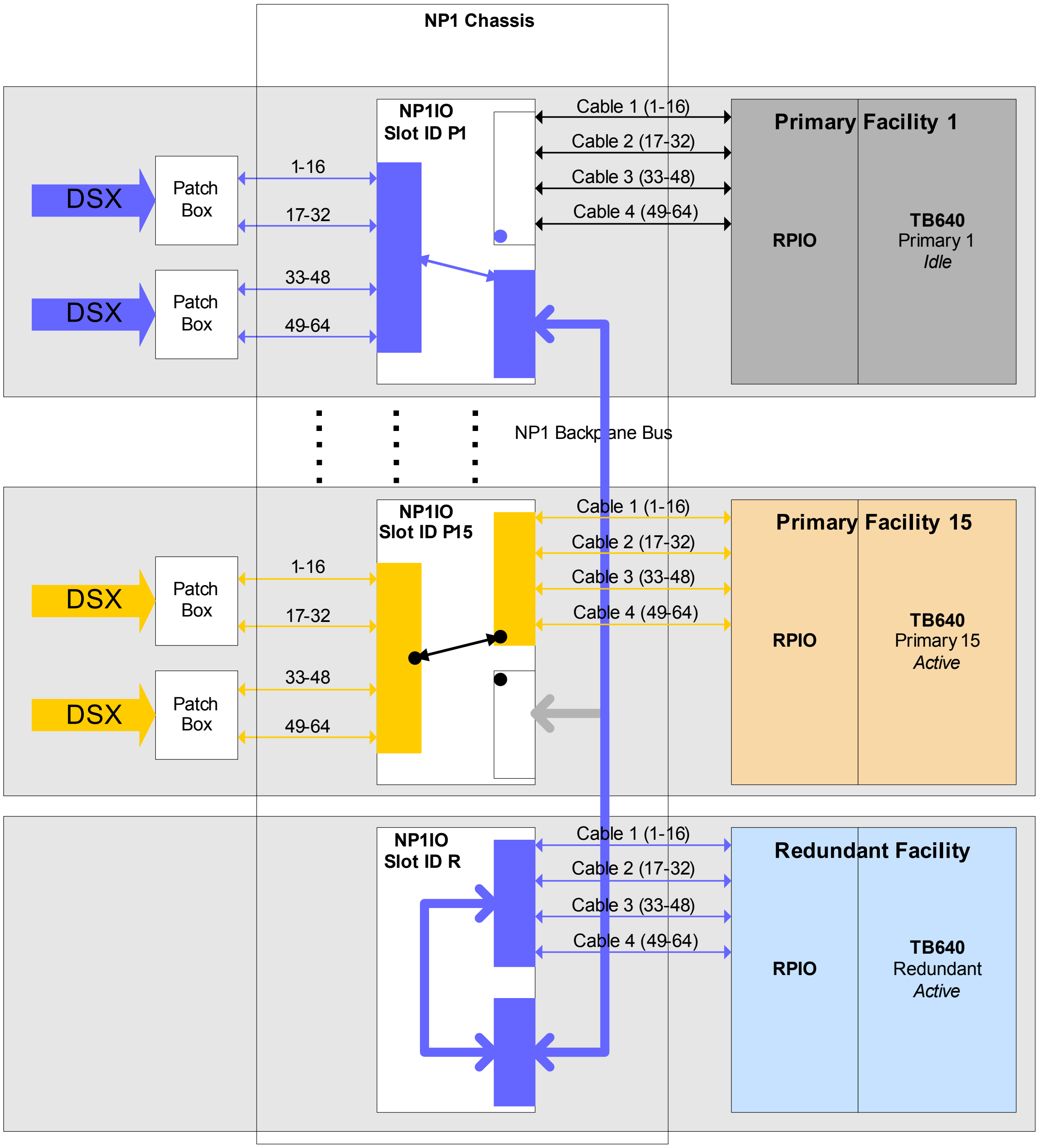


Figure 69: NP1 switched primary 1 facility traffic flow – Redundant Takeover

12.2.2 Cable Management

As seen on the above figures, the tributaries coming from the DSX patch box are connected to the NP1IO board in the NP1 chassis (4 cables of 16 trunks). The NP1IO board is connected to a RPIO of a facility through 4 cables of 16 trunks. The RPIO cables are also transporting control signals (slot ID, connector ID, switch status) and power.

12.2.2.1 Power Sharing

The adapter of a facility is providing the power to the NP1IO card though the RPIO cables 1 and 2 (trunk 1-32). However, the NP1IO cards share their power through the NP1 backplane. If the adapter of a facility is removed or the RPIO cable 1 or 2 disconnected, the NP1IO board will remain powered if at least one NP1IO in the NP1 chassis is powered by its facility adapter. Typically, the redundant facility adapter will always remain and power its NP1IO because it is controlling every NP1IO primary switch (discussed in section 12.2.4).

12.2.2.2 Slot ID and connector ID

To help the cable management between the facilities RPIO and their respective NP1IO and to help reporting misconnections between them (see cable inconsistency alarm), the facility adapter can read, for each cable, the NP1IO slot ID and the NP1IO connector ID on which the cable is connected. For example, the cable 1 of the Primary facility 1 should report a slot ID of 1 and a connector ID of 1, the cable 2 of the same facility should report a slot ID of 1 and a connector ID of 2 and so on. This information can be retrieved on each facility with the TB640_MSG_ID_NP1_OP_FACILITY_GET_STATE message API. The NP1IO slot ID determines the Facility ID of the adapter connected to it.

12.2.2.3 Alarms

There are four alarms to indicate a communication or cable connection problem with the NP1IO card:
CABLE_1_2_RED_ALARM:
This alarm indicates a LOS on the cables 1 and 2. In such a situation, the communication is totally lost with the NP1IO, there is either a hardware or cable connection problem. This is a trunk traffic affecting situation (64 trunks).
CABLE_1_2_YELLOW_ALARM:
This alarm indicates a partial lost of communication of the RPIO with the NP1IO on cables 1 and 2. This is a remote LOS indication from the NP1IO. In such a situation, the information read still valid but there is either a hardware or cable connection problem. This might lead to a trunk traffic affecting situation.
CABLE_3_4_RED_ALARM:
This alarm indicates a LOS on the cables 3 and 4. In such a situation, the communication isn’t lost with the NP1IO. However, the information concerning cable 3 and 4 becomes invalid. There is either a hardware or cable connection problem affecting the traffic on the trunk 32 to 64.
CABLE_INCONSISTENCY:
This alarm is raised when the cable information are inconsistent. This situation means there is a cable misconnection.

Alarm indication changes are reported by the adapter through TB640_MSG_ID_NP1_NOTIF_ALARMS event indication messages.

The alarm status can also be retrieved using TB640_MSG_ID_NP1_OP_FACILITY_GET_STATE message API.

12.2.3 Chassis Management

To ease the chassis management, the NP1 chassis can be uniquely identified by setting a shelf ID on the redundant facility. To do so, the host application must send a TB640_MSG_ID_NP1_OP_REDUNDANT_SET_SHELFID message API to the redundant facility adapter. This API is only available on an adapter connected to an NP1IO in a redundant slot. This shelf ID is propagated to all other NP1IO cards into the same NP1 chassis. This shelf ID can be retrieved from any facility adapter using the TB640_MSG_ID_NP1_OP_FACILITY_GET_STATE message API. The shelf ID is not a persistent configuration, meaning that under a reboot or reset of the redundant adapter, this value is reset to an invalid value until it is reprogrammed by the host application.

12.2.4 Takeover operation

The operation of switching over a primary facility to the redundant facility is called a Takeover operation. This operation instructs the redundant facility to take over the traffic of a primary facility adapter. The redundant NP1IO card controls the primary NP1IO cards switch through the NP1 backplane. The Takeover operation is performed by sending a TB640_MSG_ID_NP1_OP_REDUNDANT_TAKEOVER message API to the redundant facility adapter. Consecutive Takeover operations can be performed without performing release operation.

12.2.5 Release operation

The release operation instructs the redundant facility NP1IO to release any switch over, therefore, the redundant facility becomes Idle, meaning there is no more primary facility traffic switched over it. This operation is performed by sending a TB640_MSG_ID_NP1_OP_REDUNDANT_RELEASE message API to the redundant facility adapter.

12.2.6 Traffic Status

On a primary facility, the traffic status represents the NP1IO switch state and is reported by the NP1IO card to the adapter. It can be either *Active* or *Idle*. The *Active* state means that traffic is flowing through the NP1IO to the primary facility adapter (see Figure 68). The *Idle* state means that traffic is switched on the NP1 backplane to the redundant facility (see Figure 69).

On the redundant facility, the traffic status represents the status of the redundant NP1IO. If the redundant facility is taking over a primary facility, the traffic state reported is *Active*. Otherwise, when the redundant facility is released, the traffic status reported is *Idle*.

The traffic status can be retrieved on any facility by sending a TB640_MSG_ID_NP1_OP_FACILITY_GET_STATE message API.

Traffic change (Active/Idle) notifications are also reported by the facility adapter through TB640_MSG_ID_NP1_NOTIF_TRAFFIC_STATE event indication messages.

12.2.7 Primary Presence

The primary presence is a state that is given by the redundant facility NP1IO. The redundant NP1IO card monitors activity of the primary NP1IO cards in the NP1 chassis. When a primary NP1IO has proper communication with its facility adapter, the redundant NP1IO see its presence. The primary NP1IO presence can be retrieved by sending a TB640_MSG_ID_NP1_OP_REDUNDANT_GET_STATE message API to the redundant facility adapter.

The primary presence change (join/leave) notifications are also reported by the redundant adapter through TB640_MSG_ID_NP1_NOTIF_NP1IO_PRESENT event indication messages.

12.2.8 NP1IO LED status

The left NP1IO LED indicates the card status. A red LED means that the NP1IO experience cable problems (any cable alarm raised), otherwise, the LED is green.

The right NP1IO LED indicates the traffic status. A green LED means the traffic status is Active and a red LED means the traffic status is Idle.

12.3 Fault detection

In order to perform N+1 switchover (use the extra adapter to replace one of the N adapter that failed), the host application must detect the failure.
Failure conditions may vary from one application to the other. Thus it is the responsibility of the application to implement appropriate fault detection schemes using the APIs provided by the TB640 adapters.

12.3.1 Events to monitor for fault detection

- TBX_MSG_ID_API_NOTIF_ADAPTER_ETH_DOWN:
  Communication with the adapter was interrupted through one of the redundant Ethernet ports. Because of network redundancy, this is a non fatal error and should be ignored in most situations.
- TBX_MSG_ID_API_NOTIF_ADAPTER_REMOVED:
  Communication was lost with the adapter. This is a fatal error if the application does not expect the adapter to be rebooted or disconnected from the network.

- TB640_MSG_ID_PMALARMMGR_NOTIF_ALARM:
  Trunk alarm has been detected. This is a fatal error if the application does not expect the trunk to be disconnected from the adapter.
- TB640_MSG_ID_PMALARMMGR_NOTIF_THRESHOLD:
  A trunk error counter has reached the threshold specified by the host application. The application will judge if this is a fatal error or not. (This event requires the host to configure the adapter using message TB640_MSG_ID_PMALARMMGR_OP_SET_THRESHOLD).
- TB640_MSG_ID_ADAPTER_STATES_GET:
  The application can regularly poll the adapter to query its states. The response to this query contains a lot of useful information to monitor board health. The application may declare adapter failure in case some of these states show unexpected values (board temperature, MMC disk problems, Ethernet link states, unexpected number of resources allocated or connected for example).
- TB640_MSG_ID_ADAPTER_STATISTICS_GET:
  The application can regularly poll the adapter to query its statistics. The response to this query contains a lot of useful information to monitor board health. The application may declare adapter failure in case some of these statistics show unexpected values (many unexpected Ethernet packet errors that, for example, may show an intermittent of failed network connection).
- TB640_MSG_ID_ADAPTER_NOTIF_CPU_REPORT:
  The adapter regularly reports its CPU usage to the host through this event. Using this event, the host can monitor if the adapter is overloaded and, for example, perform load balancing across multiple adapters.
- Watchdog:
  The adapter has a built-in hardware watchdog that will automatically reboot the adapter's software fails to write to it. To enable this feature, the host sends a message type TB640_MSG_ID_ADAPTER_OP_KEEP_ALIVE, giving a keep alive timeout value to the adapter. The host application must send this message again before this timeout is reached, otherwise the adapter will automatically reboot. This feature may help detect a failed adapter, failed host application or failed network.

13 MULTI-BLADES SYSTEM

13.1 Overview

It's possible to interconnect up to 1024 trunks from 16x TB640 blades (64 trunks by blade) to build large system with the TB-MB blade. The interconnection of trunks from different TB640 blades can be achieved using single TB-MB blade. TB-MB has the capability to interconnect up to 16x TB640 blades. Each TB640 blade must have the hardware capability (Multi-Blade Link (MBL) mezzanine) and valid license that allows usage of MBL port. The second TB-MB blade is optional and allows building of redundant system (redundant TB-MB blades). Each MBL port on TB640 blade has two links (link A and link B). Only one link is active by port. These links allow connection of TB640 blade to redundant TB-MB blades. Differently from TB640 blade, each MBL port of a TB-MB blade has a single link. See following Figure 70 for details:

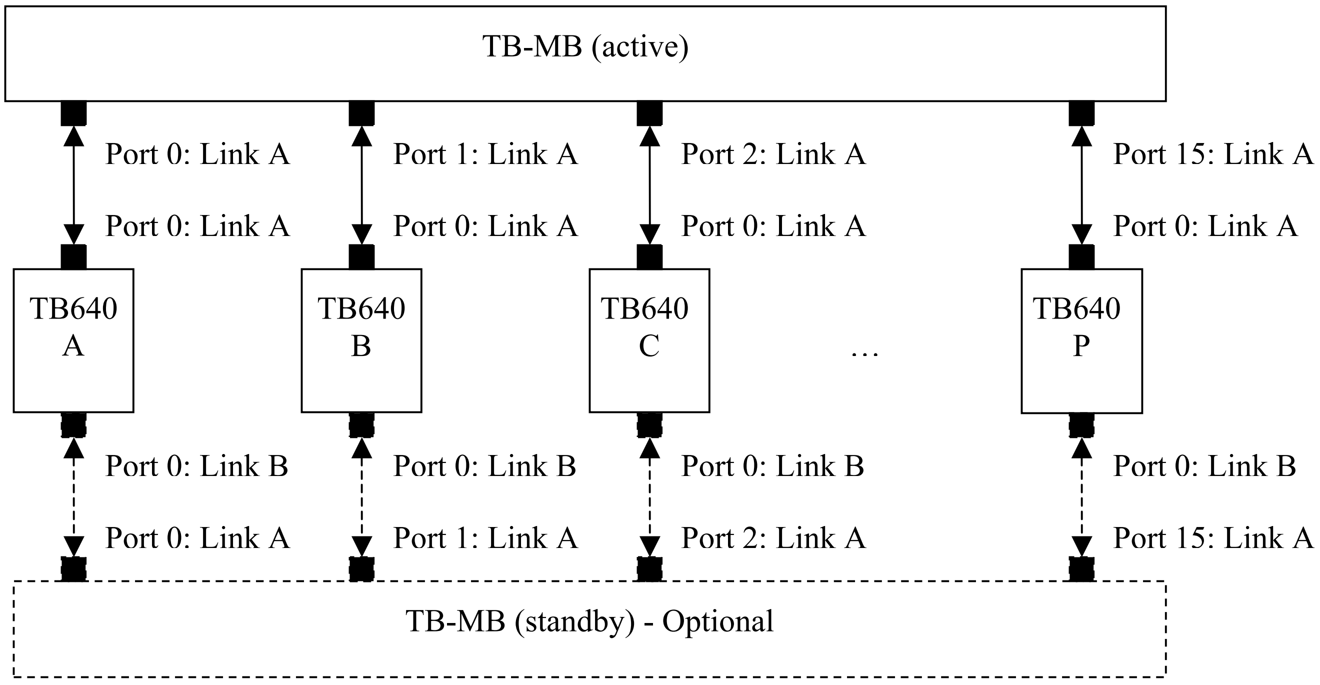


Figure 70: Multi-Blades system scheme with redundant TB-MB option.

The links that interconnect TB640 blades with the upper TB-MB in Figure 70 are connecting an active TB-MB to TB640 blades. Therefore, they are active links. The links that interconnect TB640 blades with the lower TB-MB in Figure 70 are connecting a standby TB-MB to TB640 blades. Therefore, they are inactive links.

The link switchover facility lets control traffic from the active TB-MB to the standby one. To get full TB-MB redundancy, host application is responsible to synchronize the standby TB-MB blade with the active TB-MB. By example, the host application could synchronize the standby TB-MB with the active TB-MB constantly by doing configuration request on both TB-MB (active redundancy option) or could synchronize the standby TB-MB with the active TB-MB only just before the switchover request (passive redundancy option). The first option minimizes the switchover delay but it is more complex to implement then the second one.

Each link physically corresponds to two cables connection from a MBL port to an others one. The first cable of a link is for clock and frame signal and the second one for data/voice signal. Losing any one of the cable will result in getting link alarms.

TB640 blades redundancy can be achieve using the N+1 scheme and the TelcoBridges N+1 backplane.

The MBL management API is the same for TB640 and TB-MB blade types.

13.2 Blades synchronization

The host application must set clock configuration of all blades individually. The host application should start with TB640 blades clock configuration first then terminate with TB-MB blade.

The TB640 clock configuration should be set to retrieve clock from trunk, CTBus, local oscillator… except MBL port. All system blades must use the system clocking mode and let TB-MB select the best clock reference available according to the clock configuration of all system blades. If TB640 clock reference is trunk then TB-MB may select this blade as system clock reference only if trunk is active. Trunk can be active or in maintenance mode. Refer to trunk API messages (TB640_MSG_ID_TRUNK_ACTIVE / TB640_MSG_ID_TRUNK_MAINTENANCE) for details concerning trunk states modification. Refer to MBL API messages (TB640_MSG_ID_MBL_STATES_GET) for details concerning how to check if clock coming from TB640 is valid and can become the system blades clock reference (check "fMasterClockReady" parameter of TB640_MBL_REMOTE_INFO structure of TB640_MBL_LINK_STATES structure.

The TB-MB will select clock reference from TB640 blades via MBL ports, its own BITS port or its own local oscillator depending on the TB-MB clock configuration. The TB-MB will automatically synchronize all system blades to unique clock reference. When all system blades are synchronized, one blade is the clock reference (master of clock reference) and all other blades of the system are retrieving clock from that blade (slave on clock reference) via MBL ports. When the selected clock reference gets invalid, the TB-MB automatically selects a new clock reference matching as much as possible the clock configuration. The TB-MB will stay to the selected clock reference as long as the current clock reference is valid. The host application can force clock selection by resetting TB-MB clock configuration.

Blades that allow MBL system clocking mode are dynamically changing clock reference according to TB-MB clock selection and will always get synchronized to all system blades. Blades that do not allow MBL system clocking mode may need host intervention to synchronize to all system blades after TB-MB clock selection. The following Figure 71 (Clock sources of the system blades) shows the different clock sources of the system blades.

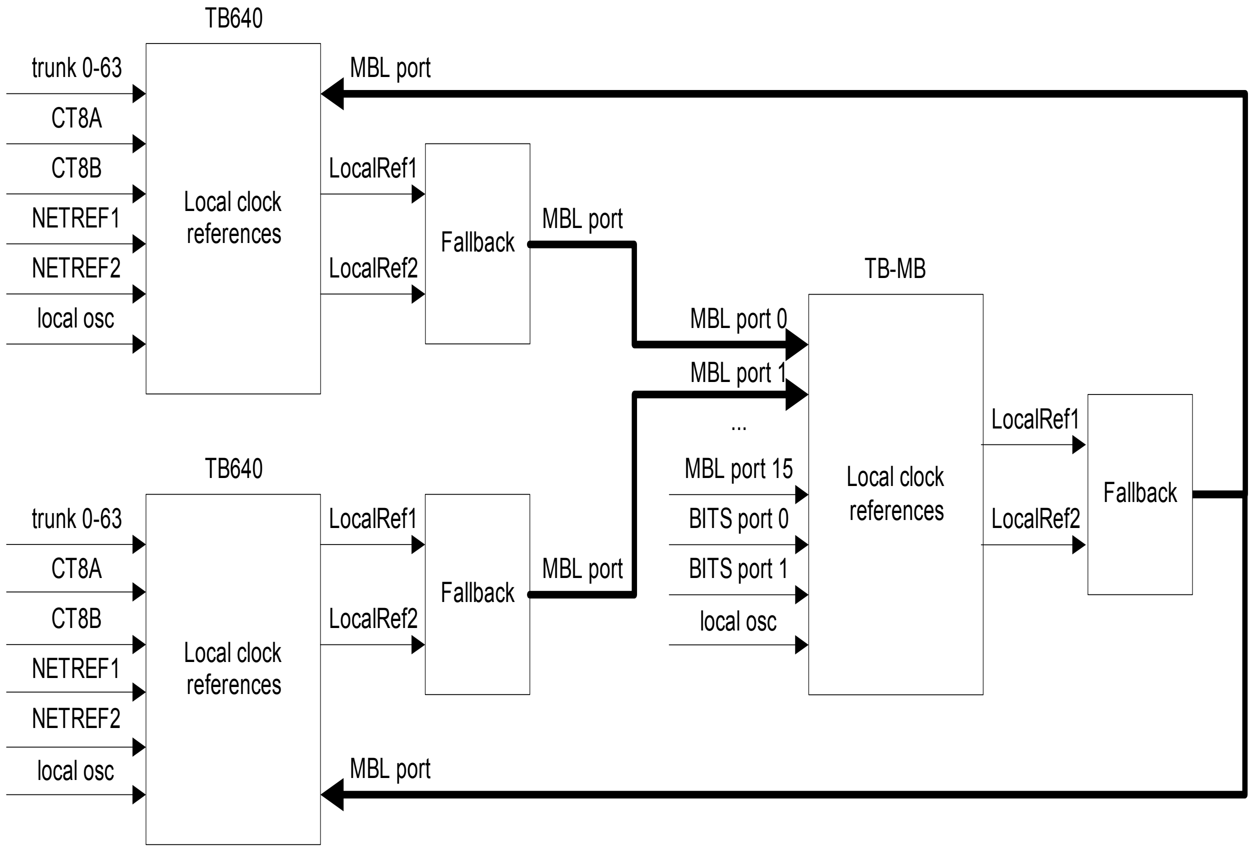


Figure 71: Clock sources of the system blades

13.2.1 System clocking mode

The system clocking mode should be enabled on all blades of the system except for some particular and unusual applications otherwise clock selection mechanism will not work to its full capability and host application will have to reconfigure clock setting on some TB640 blades when losing the primary clock reference. The system clocking mode allow TB640 to change its own clock configuration dynamically (inhibit clock configuration) to retrieve clock reference from TB-MB blade (from MBL port) when ever it is required. Therefore, the system clocking mode makes all TB640 blades available to become the system clock reference and makes all TB640 blades changing clock reference automatically when system clock reference changes. If system clocking mode is enabled on TB-MB side only (not enabled on TB640 side), the TB-MB will behave like there is no system clocking mode. If system clocking mode is enabled on TB640 side only (not enabled on TB-MB side), the TB640 will behave like there is no system clocking mode. The system clocking mode must be enabled on both sides (TB-MB and TB640) of the link to operate normally.

13.2.2 Clock selection mechanism

The TB-MB will select clock reference that is of the same type as the primary clock reference first then select clock reference that is of the same type as the secondary clock reference secondly. If both are of the same type then the TB-MB will select the primary clock reference first then the secondary clock reference secondly. If both are of the same type but none is valid then the TB-MB will select the first available clock reference that is available of the same type as the primary and secondary clock reference. Following Figure 72 shows clock selection mechanism flowchart. The behavior of the clock selection mechanism depends also on system clocking mode setting of all blades of the system. Refer to following clock selection scenarios and section 13.2.1 (System clocking mode) for details concerning clock selection behavior and system clocking mode setting.

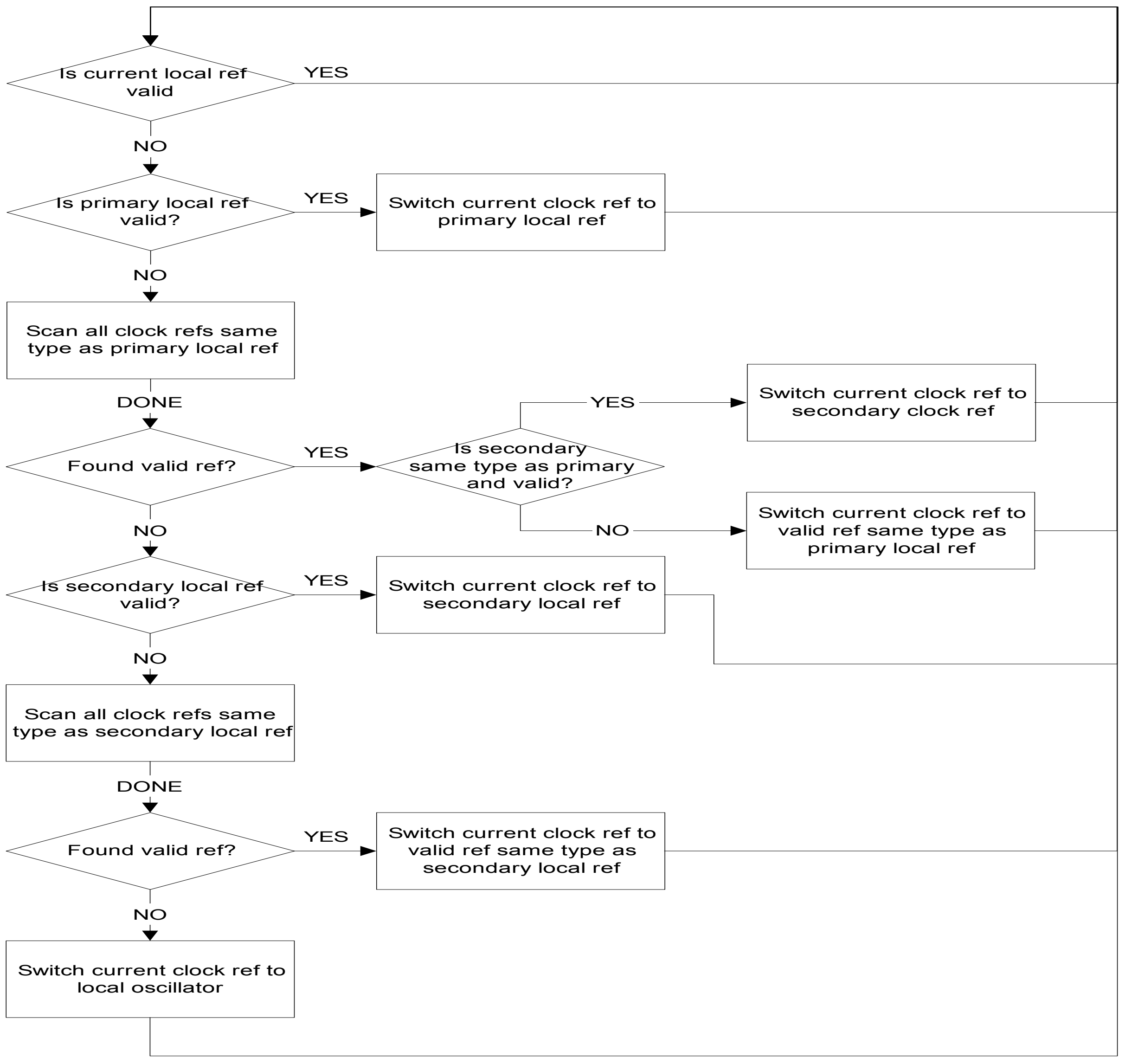

Is current local ref valid
YES
NO
Is primary local ref valid?
YES
Switch current clock ref to primary local ref
NO
Scan all clock refs same type as primary local ref
DONE
Found valid ref?
YES
Is secondary same type as primary and valid?
YES
Switch current clock ref to secondary clock ref
NO
Switch current clock ref to valid ref same type as primary local ref
NO
Is secondary local ref valid?
YES
Switch current clock ref to secondary local ref
NO
Scan all clock refs same type as secondary local ref
DONE
Found valid ref?
YES
Switch current clock ref to valid ref same type as secondary local ref
NO
Switch current clock ref to local oscillator

Figure 72: Clock selection mechanism with Multi-Blades system

Scenario #1 – MBL port as primary and secondary clock references on TB-MB (using system clocking mode on all system blades)

If primary clock reference fails on TB-MB then TB-MB switches to secondary clock reference and all TB640 clock configurations are modified to retrieve their clock reference from TB-MB except for the TB640 blade where secondary clock reference comes from. If secondary clock reference fails on TB-MB then TB-MB switches to first valid clock reference of the same type as the primary and secondary clock reference and all TB640 clock configurations are modified to retrieve their clock reference from the TB-MB except for the TB640 blade that clock reference comes from. If no valid clock reference of the same type as the primary and secondary clock reference is available on TB-MB then TB-MB switches to its local oscillator and all TB640 clock configurations are modified to retrieve their clock reference from the TB-MB without exception. As soon as clock reference of the same type as the primary and secondary clock reference becomes available, the TB-MB switches back the system clock reference to this source and all TB640 clock configurations are modified to retrieve their clock reference from TB-MB except for the TB640 blade that clock reference comes from. The TB-MB will not switch from secondary to primary clock reference when the current clock reference is secondary and the primary clock reference becomes available.

Scenario #2 – BITS port as primary and secondary clock references on TB-MB (using system clocking mode on all system blades)

If primary clock reference fails on TB-MB then TB-MB switches to secondary clock reference. If the secondary clock reference fails on TB-MB then TB-MB switches to first valid clock reference of the same type as the primary and secondary clock reference. If no valid clock reference of the same type as the primary and secondary clock reference is available on TB-MB then TB-MB switches to its local oscillator. As soon as clock reference of the same type as the primary and secondary clock reference becomes available, the TB-MB switches back the system clock reference to this source. The TB-MB will not switch from secondary to primary clock reference when the current clock reference is secondary and the primary clock reference becomes available.

Scenario #3 – BITS port as primary and MBL port as secondary clock references on TB-MB (using system clocking mode)

If the primary clock reference fails on TB-MB then TB-MB switches to first valid clock reference of the same type as the primary clock reference. If no clock reference of the same type as the primary clock reference is available on TB-MB then TB-MB switches to secondary clock reference and all TB640 clock configurations are modified to retrieve their clock reference from TB-MB except for the TB640 blade where secondary clock reference comes from. If secondary clock reference fails on TB-MB then TB-MB switches to first valid clock reference of the same type as the secondary clock reference and all TB640 clock configurations are modified to retrieve their clock reference from the TB-MB except for the TB640 blade that clock reference comes from. If no valid clock reference of the same type as the secondary clock reference is available on TB-MB then TB-MB switches to its local oscillator and all TB640 clock configurations are modified to retrieve their clock reference from the TB-MB without exception. As soon as clock reference of the same type as the primary or secondary clock reference becomes available, the TB-MB switches back the system clock reference to this source and all TB640 clock configurations are modified to retrieve their clock reference from TB-MB except for the TB640 blade that clock reference comes from. The TB-MB will not switch from secondary to primary clock reference when the current clock reference is secondary and the primary clock reference becomes available.

13.3 Ports management

Allocation of MBL port is required before being able to allocate MBL port resources and being able to connect MBL port resources to trunk channels. Configuration of MBL ports is done when allocating the ports. It is possible to retrieve the currently allocated ports and the current port configurations when ever application needs it. The port can be freed as soon as there is no more port resources currently allocated for this port.

13.4 Resources management

Allocation of MBL port resource is required before being able to connect MBL port stream/timeslot to another resource of the system. It is possible to retrieve the currently allocated MBL port resources of a previously allocated MBL port when ever application needs it. The MBL port resource can be freed as soon as there is no more connection using the MBL port resource.

Refer to section 6.1.5 for more details about MBL port resources.

13.5 States and statistics information

It is possible to retrieve the current link states and statistics information of MBL port when ever application needs it. It is not required to allocate MBL port to retrieve the current link states and statistics information of MBL port. The states information contains the current link alarms, the currently active link and remote link information to detect adapter interconnections. The statistic counters reset after every read request.

13.6 Alarms and indications

MBL port status changes are notified to host application that has previously allocated the MBL port. MBL port status change could be a link alarms or a link change indications. The link alarm is sent to host application when the error condition raises and falls.

13.7 Redundancy facility

The TB640 port has the capability to switch between link A and B. When link switchover occurs on link A, the link A becomes the active link and link B becomes the inactive link. When link switchover occurs on link B, the link B becomes the active link and link A becomes the inactive link. The link switchover operation could be done manually or automatically.

Manual link switchover:

The link switchover request can be sent to TB640 or TB-MB but link switchover always occurs on the TB640 side. This mean that the link switchover request message can be sent to TB-MB adapter to remotely switch TB640 active link. A single link switchover request is needed to switch all the links of all ports of all the TB640 blades of a system. Just send a single request to any blade of the system to switch active link of all TB640 ports. This single request allows complete link switchover using single request message.

Automatic link switchover:

The automatic link switchover behaves like the manual link switchover except that no request message is required to trig the operation. The automatic link switchover occurs automatically when a currently active link is getting major link alarms (clock error, frame error or communication frame error). It is possible to propagate the link switchover that occurs on a TB640 blade to all the TB640 blades of the system and make all TB640 blades switching their active link automatically.

13.8 Port LED status

The first LED of TB640 port gives link A status information. The second LED of TB640 port gives link B status information. The first LED of TB-MB port gives link A status information. The second LED of TB-MB port gives clock source selection information. See following description:

Link status information (first and second LEDs of TB640 and first LED of TB-MB):

Green LED:	There is no link alarm. Link is up.
Yellow LED:	There is at least a minor link alarm. Link is down.
Red LED:	There is at least a major link alarms. Link is down.
Blinking LED:	Currently active port (see section 13.7 `Redundancy facility`)

Clock source selection information (second LED of TB-MB):

Blank LED:	This port is not a valid master clock source.
Green LED:	This port is a valid master clock source but not currently driving clock source.
Blinking green LED:	This port is currently driving a valid clock source.

14 SNMP

14.1 Overview

The SNMP (Simple Network Management Protocol) was created to manage devices attached to the network. SNMP is based on the manager /agent model consisting of a manager, an agent, a database of management information, managed objects and the network protocol.

The manager and agent use Management Information Base (MIB) and a relatively small set of commands to exchange information. The MIB is organized in a tree structure. Each node in tree is assigned a unique dot separated sequence of integers called OID(Object Identifier).

SNMP uses five basic messages (GET, GET-NEXT,GET-RESPONSE,SET and TRAP) to communicate between the Management Station and the agent. The GET and GET-NEXT messages are used by the Management station to request information for a specific variable. The GET-RESPONSE message is sent by the agent in response to GET/GET-NEXT request. The GET-RESPONSE may contain the values of the all variables in the GET/GET-NEXT requests or an error if the agent fails to process at least one variable in the request message.
The SET message, in the other hand, is used by Management station to request a change be made to the value of a specific variable. The TRAP message allows the agent to spontaneously inform the manager of an important event.

14.2 SNMP Limitation

In this section we present the limitation of our current implementation of the SNMP protocol. The only version supported by our implementation is the SNMP version 1. The communities *public*, *telcobridges* can be used to query (GET, GET-NEXT) variables and the community *private* is used to request change (SET) of a specific variable

14.3 SNMP Messages

The messages GET and GET-NEXT are fully supported. The message SET is only used to change the trunks' loopback status. In other words, even if some variables are declared with read-write access, all requests to change the values of these variables cause an error to be returned to the Management station. The only exception is the trunk's loopback variable.
Finally, the message TRAP is not supported and it will not be generated by the agent.

14.4 Supported MIBs

14.4.1 RFC 1213 MIB II

Here is a list of the limitation or the unsupported feature in this MIB:

- `The ipRouteTable is not implemented`
- `The EGP group is not implemented`
- `The atTable is not implemented.`
- `TCP,UDP and ICMP statistics are the sum of statistics from the two CPUs`
- `Only the udpTable and TcpConnTable of CPU 0 will be shown`

14.4.2 RFC 2959 Real-Time Transport Protocol Management Information Base

`The following tables are supported:`

- `rtpSessionTable`
- `rtpSenderTable`
- `rtpRcvrTable`

The agents returns the value 9999 for the unsupported values

14.4.3 RFC 2495 DS1, E1, DS2 and E2 Interfaces

The trunks will be shown in the interface table independently of their configurations/allocations. In addition, the agent returns the 9999 for unsupported fields

The following tables are supported:

- Dsx1ConfigTable
- Dsx1CurrentTable
- Dsx1IntervalTable
- Dsx1TotalTable

It should be noted here that the trunk’s loopback setting is supported in dsx1ConfigTable

14.4.4 RFC 2496 - Definitions of Managed Object for the DS3/E3 Interface Type

The DS3 line interfaces will be shown in the interface table independently of their configurations/allocations. In addition, the agent returns the 9999 for unsupported fields

The following tables are supported:

- Dsx3ConfigTable
- Dsx3CurrentTable
- Dsx3IntervalTable
- Dsx3TotalTable

14.4.5 Telcobridges Private MIB (TB-MIB)

The private MIB provides information on the software and hardware components installed on the adapter.
The TB-MIB file is provided with the package under the directory ***tb/tb640/doc/mibs.***
Please refer to TB-MIB file for more information about a specific object.

The following paragraph provides roadmap of TB-MIB files.

1. **Direcotry group**: contains the module table which have an entry for each supported module
2. **Common Group** : contains the following objects
   a. tbHwCommon:
      i. Serial Number, Slot ID, Shelf ID and adapter's part number
      ii. tbHwVersinTable :
      iii. tbHwTempTable(Device Temperature)
   b. tbSwCommon
      i. tbSwFeatureTable
      ii. tbVersionTable
3. **Specific Group** : contains product's specific objects
4. **Experimental Group** : used for experimental purpose

At the moment of writing this document, the **Specific** and **Experimental** groups are empty

14.5 Browsing MIBs

The tools used during testing are the command lines tools provided by net-snmp. These tools can downloaded freely from http://net-snmp.sourceforge.net. In order to see the objects' names when using these tools, the environnement variable MIBDIRS should be set to the location where the MIB files were installed.

All the tools accept the following options:

-v	: version. In our case only version one is supported
-m	: modules list. Use ALL to load all modules
-c	: community's name. Use public for read and private for write

Please refer to the tools for more information about theirs usage.

14.5.1 Get a single object

The command *snmpget* can be used to get one or more objects in the same request. In this example we send a request command to read system's description and system up time:

>snmpget –v 1 –c public –m ALL 192.168.100.100 sysDescr.0 sysUpTime.0
Your can replace the object's name by theirs OIDs

14.5.2 Get the next object

The command *snmpgetnext* can be used to get the next lexicographically object in the tree following the specified object. In this example we send a request command to read the next object after the system's description.

```
>snmpgetnext  –v 1 –c public –m ALL 192.168.100.100 sysObjectID.0
```

```
SNMPv2-MIB::sysObjectID.0 = OID: TB-MIB::Specific.1.252
```

14.5.3 Get a specific table

The command *snmptable* can be used to get a table and display it in a tabular form. The following example reads the features tables from the Telcobridges private MIB.

```
>snmptable  –v 1 –c public –m ALL 192.168.100.100 tbFeaturesTable
```

14.5.4 Get a tree branch

In the following example, the command *snmpwalk* is used to display a branch. When using this command, the tables are displayed by columns rather than by lines.

```
>snmptable  –v 1 –c public –m ALL 192.168.100.100 tbFeaturesTable
```

15 REVISION HISTORY

Changes in revision 9000-00002-1X

- ✓ New API message (TB640_MSG_ID_CAS_CMD_ACCEPT_INCOMING_CALL) was added in CAS call scenarios for R1 variants (except Taiwan MDR1) in non-DDI mode only. This change affects sections 8.3.5.1.1 and 8.3.5.1.3.
- ✓ R1 variants (except Taiwan MDR1) in DDI mode now disconnect and free tone detector and generator resource right before sending the message TB640_MSG_ID_CAS_NOTIF_CONNECT_INDICATION or TB640_MSG_ID_CAS_NOTIF_CALL_PRESENT_INDICATION. This change affects sections 8.3.5.1.2 and 8.3.5.1.4.

Changes in revision 9000-00002-1Y

- ✓ Added the description of pure-TDM resource (section 6.2.2.4.1.2) in VP Group1 to support call progress tone detection feature.
- ✓ Modified Figure 43 and added Figure 44 to better distinguish between functions of a pure-TDM resource and functions of a TDM resource part of a TDM/Stream group
- ✓ Added a section about call progress tone detection (section 6.2.2.5.2)
- ✓ Added a warning about Payload type value configuration (section 6.2.2.5.5)
- ✓ Added a warning about adaptative jitter-buffer and modem/fax passthrough (section 6.2.2.5.7)

Changes in revision 9000-00002-1Z

- ✓ Updated Table 12 and Table 13 about Japan INS isdn variant

Changes in revision 9000-00002-2A

- ✓ Updated sections 6.2.2.5.5 and 6.2.2.5.6 on how to configure VP Group1 payload types and to decipher SIP SDP information.

Changes in revision 9000-00002-2B

- ✓ Fixed typos and bad indexes after review of section 6.2.2.5.6.

Changes in revision 9000-00002-2C

- ✓ Changed section 5 to include information about SONET and SDH line interfaces/services for the TB640-STM1 blade.
- ✓ New sub-section 5.3 describing SONET and SDH

Changes in revision 9000-00002-2D

- ✓ Changed Table 10 and Table 11 to use the enum TBX_MEDIA_TYPE instead of TBX_STREAM_PACKET_TYPE since the later is now obsolete (although still supported to ease porting older applications to newer releases)
- ✓ Change 'SIP to VPGrp1' code samples from section 6.2.2.5.6 to use TBX_MEDIA_TYPE instead of TBX_STREAM_PACKET_TYPE

Changes in revision 9000-00002-2E

- ✓ Added STM-1 clock configuration information in DS3 clock configuration section.
- ✓ Renamed section “DS3 clock configuration” to “DS3 and STM-1 clock configuration”

Changes in revision 9000-00002-2F

- ✓ Modified Table 8 to remove G.726 5msec

Changes in revision 9000-00002-2G

- ✓ Added a warning about invalid T.38 Fax resource pre-allocation methodology in sections 6.2.1.5, 6.2.1.7 and 6.2.2.5.9

Changes in revision 9000-00002-2H

- ✓ Removed references to obsoleted defines in section 6.2.2.5.5 and 6.2.2.5.6 when converting from SIP SDP to VpGrp1 parameters.

End of the document