TsLink3®
API Overview

TABLE OF CONTENTS

1. General Description

TeleSoft provides a Universal API which can be used with all TsLink3 stacks. The rich TsLink3 Universal API coupled with the straightforward structure of TsLink3 protocol stacks enables you to easily follow the API message flow through the code to determine where to make modifications required for your application. The API can be used in three different modes:

Unrestricted mode: For complex applications that combine signaling with data protocols or specialized hardware -- applications that benefit from a very flexible, message-based interface which allows extensions to the command set and the use of a large set of messages and a wide array of potential parameters. Can go directly to L2 and L1, primarily for test purposes.

Function-Call Mode: A simple interface for simple applications such as signaling-only. A set of example functions to be used directly for simple speech applications or used as templates for application-specific enhancements.

AT Mode: For serial control stream applications, such as TAs. Maps standard 'AT' commands to the Unrestricted API.

Many projects do not require full understanding of the details of the Unrestricted API, and can use the higher-level Function-Call or API Mode to accelerate portation.

All of the API modes allow copy-free data transfer by use of "buffer holes" allowed for at creation of outgoing data blocks. No copying of buffers is needed while a single buffer management mechanism is maintained.

Early in the integration process, the TeleSoft engineer providing your technical support will work with you to determine the API features appropriate for your application. The majority of signaling-only applications require a very small subset of the Unrestricted API messages and parameters, allowing most options to be ignored. Code examples provided with the stack (including examples listed in Section 2 of this document) show how easy it is to interface to the TsLink3 Universal API and quickly begin commanding TsLink3 to setup and teardown calls.

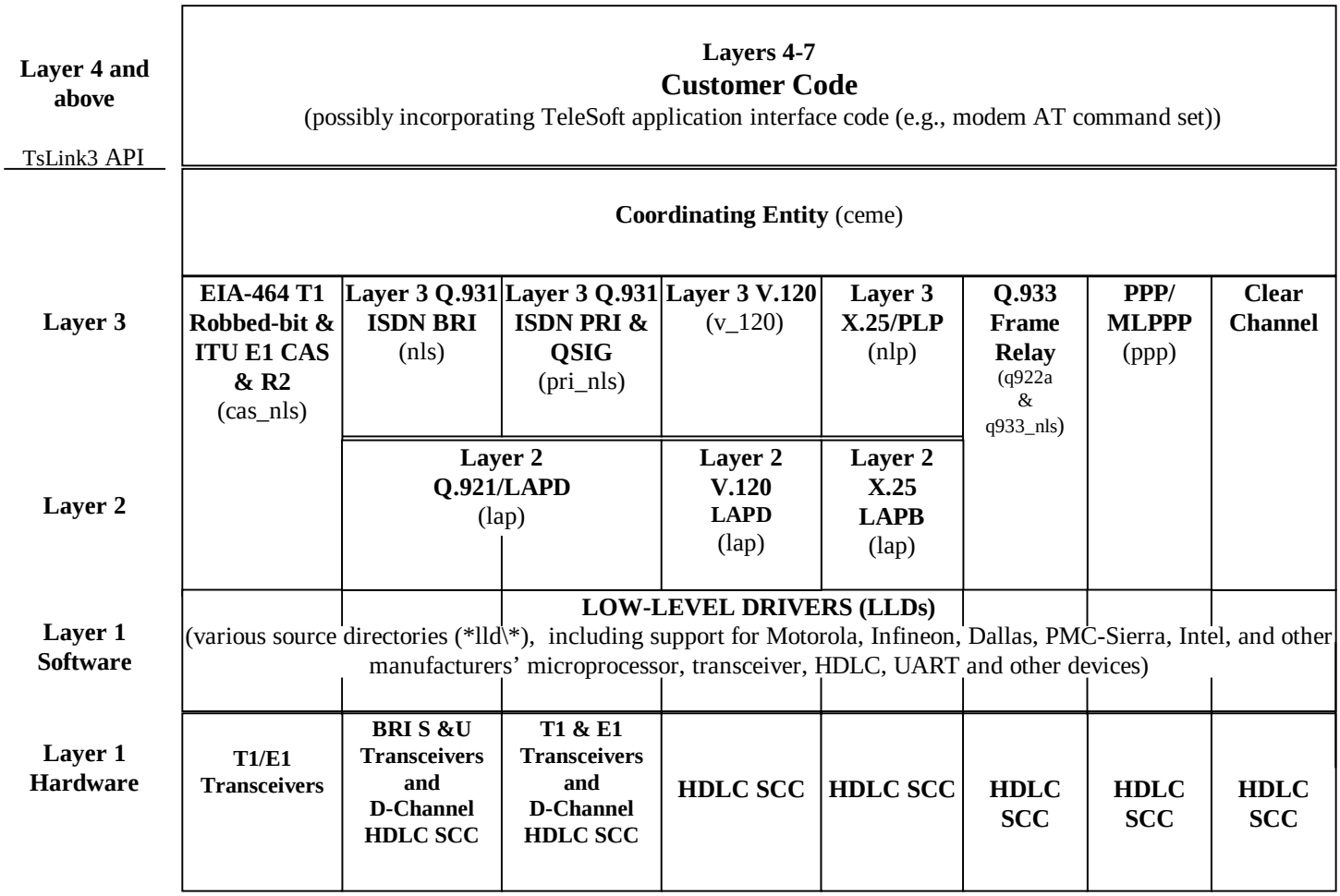
1.1 Relationship of the API Modes to Other Layer Interfaces

There are several software levels at which TsLink3 customers can interface their code to the TsLink3 stack. Figure 1 shows the TsLink3 software layer block diagram illustrating these inter-layer interfaces. In general, since each of the interfaces is well-defined and has its own message set, the TsLink3 customer can interface to the TsLink3 code at the top of Layer 1, Layer 2, or Layer 3, or interface to the "bottom of Layer 4" of the OSI 7-Layer Interconnection Model by making use of the rich flexibility of the Unrestricted API.

Most commonly, customers interface their to code to TsLink3 at the "bottom of Layer 4", and can use the provided Function-call API mode. The Inter-layer boundaries, able to be controlled by the APIs, are shown in Figure 1 below. The Unrestricted mode is referred to as the "TsLink3 API" because it is the application interface point common to all of the modes.

Referring to Figure 1, the CE is a "thin" software entity that is conceptually part of Layer 3 and provides call control sequencing/coordination between the call setup signaling protocol entities (Q.931, T1 Robbed-bit Signaling (RBS), E1 Channel Associated Signaling (CAS), etc.) and (optional) bearer channel user data protocol entities (PPP, X.25, etc.) delivered in a TsLink3 load.

Note: The software block diagram in Figure 1 and the entities contained therein are explained more fully in the TsLink3 "General Overview" document.



TsLink3 Software Block Diagram
Figure 1

Notes: The main source code directory in which entity's source code is found is indicated by the "src" subdirectory name in parentheses (e.g., (ceme)).

1.2 Overview of the TsLink3 API

The TsLink3 API, under Layer 4 control and monitoring, is a message-based system used for:

- Setting up and tearing down connections on telecom lines.
- Monitoring call progress and other status of the lines and connections.
- (Optionally) Sending and receiving user data on bearer channels allocated during connection setup (not applicable for signaling-only applications).
- (Optionally) Performing certain specialized functions which are application specific and not needed for most TsLink3 customers and applications.

The API consists of a set of messages, and parameters within messages, to accomplish the above functions. This document provides a high-level overview of the API. Detailed descriptions of each message type and parameter are provided in the "TsLink3 Interface Reference Guide," section 2 "Coordinating Entity Interface."

Universal API --Uniform Across Different Signaling Protocols
TsLink3 presents a uniform API to customer Layer 4 software across the range of supported protocols. The customer Layer 4 software exchanges simple protocol-independent messages with the TsLink3 software to setup a connection, optionally transfer user data, and eventually teardown the connection independent of the protocol type used in lower layers (e.g., ISDN, T1 RBS, E1 CAS). The uniform API facilitates the customer's development of a single Layer 4 code module that uses the same simple API messages to setup and teardown calls on different telecom lines that may employ different signaling protocols.

Higher-level Modes created with an Abstraction Layer
The TsLink3 API abstracts (simplifies) the complex, underlying lower layer protocols which hides unneeded complexity from higher layers so that the customer application interface is straightforward and is easier to maintain. During the lifetime of the connection, fewer and simpler messages related to a connection pass to the customer application, across the TsLink3 API, than pass across the lower layer TsLink3 interfaces.

The TsLink3 API is hardware independent and works with any message passing mechanism. It is easy to redirect the flow of messages at the TsLink3 API to any customer Layer 4 software message source/sink because all message interactions between software layer entities in the TsLink3 system are funneled through a few "send a message" and "receive a message" interface functions. Both the "send a message" and "receive a message" functions are relatively short C functions which are concerned only with the mechanics of passing messages, not with how to process messages once messages are received. For example, when the protocol-independent "receive" function receives a message, it immediately calls a protocol-specific "process a message" function to parse and process the message. Simple changes to these "send" and "receive" message functions/files are usually all that is required for most clients to interface their application to the TsLink3 API.

1.3 Ease of Adapting to a Variety of Message Passing Mechanisms

At the TsLink3 API (see Figure 1), the L4->CE and CE->L4 message passing mechanism has been implemented in several ways across many different product types by TeleSoft and its customers. Message passing mechanisms have included:

- Multitasking system message passing via message queues (i.e., customer Layer 4 is (at least in part) a set of tasks running on the same microprocessor under the same multitasking OS as the TsLink3 layer entity tasks and interrupt handlers).
- Bidirectional mailboxes in dual-port RAM (i.e., customer Layer 4 is running on a different processor than the TsLink3 layer 1 through 3 entities, and the two processors communicate via dual-port RAM message mailboxes and bidirectional inter-processor interrupts).
- Direct function calls (i.e., both L4 and CE run on the same processor and the "send a message function" is directly coupled to the "process a message" function and does not need to call the "receive a message" function.
- Message FIFOs.
- Asynchronous or synchronous serial channel. Our 'AT' mapping API is available for ease of use in this configuration.
- Ethernet

The messages are defined independently of the method of passing them so it is easy to install almost any physical passing mechanism by changing the appropriate "send" and "receive" functions.

1.4 Most Applications Use Only a Small Number of the API Messages

The TsLink3 API privides a powerful and flexible interface that supports the full range of TsLink3 protocols including ISDN, Robbed-bit/CAS, ML-PPP, Frame Relay,and X.25. Most customers use only a small subset of the many message types and parameters supported by TsLink3. The following table lists the message types most commonly used for various applications:

1.4.1 Layer 4 (application interface)-to-CE Commonly Used Messages

Message Type (primcode)	Function
N_CONNECT_REQUEST	Layer 4 initiates a call/connection.
N_CONNECT_RESPONSE	Layer 4 answers a call/connection offered earlier by the CE via an N_CONNECT_INDICATION.
N_DISCONNECT_REQUEST	Layer 4 hangs-up a call/connection.
N_DATA_REQUEST	Layer 4 sends a user data block on an existing connection.

1.4.2 CE-to-Layer 4 (application interface) Commonly Used Messages

Message Type (primcode)	Function
N_CONNECT_INDICATION	Layer 4 is offered an incoming call/connection.
N_CONNECT_CONFIRMATION	The far end has answered a call/connection requested previously by Layer 4 via an N_CONNECT_REQUEST.
N_DISCONNECT_INDICATION	The far end of a call/connection has hung up/disconnected.
N_DISCONNECT_CONFIRMATION	The far end has finished hanging up/disconnecting a call/connection previously disconnected by Layer 4 via an N_DISCONNECT_REQUEST.
N_DATA_INDICATION	Layer 4 is passed a user data block received from the far end on an existing connection.
N_STATUS_INDICATION	Layer 4 is passed status related to a particular call/connection or to a particular "physical interface" (e.g., T1/E1 span, ISDN telephone line, etc.). Only a few of the total set of TsLink3 status subcodes are used by most applications/cutomers.

Other messages in the set are used only for certain situations (e.g., X.25) and are not needed by most customers. Many of the messages are for informational purposes only and these optional messages may be ignored by customer Layer 4 software. In addition to the detailed descriptions of the messages in this document, example Layer 4 code segments supplied with most TsLink3 customer loads illustrate which L4<->CE<->L3 messages are needed for particular situations.
It is generally a quick convergence process for the TsLink3 customer to determine which relatively few messages need to be sent by customer Layer 4 software and which relatively few messages need to be handled by customer Layer 4 software received from the CE to support the customer's application.

1.5 Common L4<->CE and CE<->L3 Message Set

All communication among the Layer 3, CE, and Layer 4 software entities is by a common TsLink3 "l4cel3" message set where the same message set is shared across both sides of the CE and the CE simply passes along messages between Layer 4 and Layer 3 (Q.931 NLS_*, X.25 NLP, etc., entities). All messages in this set use a common 32-byte long "l4_ce_l3_msg" message structure and set of message types defined in file "src\include\l4cel3.h". A small number of "l4cel3" messages require more room for associated parameters than will fit in the basic "l4cel3" struct. The "l4cel3" struct for these "extra large" messages contains the length and address of an associated "Indirect Parameter Block" (IPB) memory buffer containing the parameters that would not fit in the base "l4cel3" message.

The CE does not usually originate or modify "l4cel3" messages as it performs its call control sequencing and routing functions. (Exceptions are certain situations where the CE sends a copy of a

connect request to more than one Layer 3 entity over time and certain error conditions where the CE detects that a call should be torn down.)

As a message passes from the “L4/CE” interface to one of the “CE/L3” interfaces, the mnemonic for the message type changes from the “N_” prefix to the appropriate Layer 3 message type prefix (e.g., “NLS_” for signaling and “NLP_” for data). However, as the name changes, the numerical “primcode” for the message does NOT change. For example, the primcode remains 0x01 and the “l4cel3” struct contents are identical for both the N_CONNECT_RESPONSE and the NLS_CONNECT_RESPONSE.

The following is a C language representation of the Layer 4/CE/NLP/NLS common message structure defined in file "include\l4cel3.h". Additional members may be added to this structure without affecting the existing software, where specific application needs exist:

```
structure l4_ce_l3_msg
{
   unsigned long reserved;                  /* reserved for oper sys use */
   unsigned long home_exch;                 /* reserved for oper sys use */
   unsigned char primcode;                  /* primitive command code */
   unsigned char receipt;                   /* primitive receipt code */
   unsigned char d_attrib;                  /* data attributes */
   unsigned char connid;                    /* CONNection ID */
   unsigned short datalen;                  /* length of data buf */
   unsigned short refnum;                   /* refnum of data buf */
   unsigned char cause[2];                  /* cause of DISC,RESET,STATUS */
   unsigned short lci_chantype;             /* Chantype or Logical Channel Number */
   unsigned char *dataptr;                  /* ptr to prim-specific data buf */
} ;
```

2. Example of Interface Functions

The following C code functions illustrate the process of formatting, sending and handling messages sent in both directions across the TsLink3 API.

2.1 Example of an outgoing speech call connection request

The following is an example of a routine which formats and sends an N_CONN_RQ message primitive across the TsLink3 API to setup an ISDN call.

```
int
sample_speech_conn_rq(unsigned char interface, unsigned char *dialed_num, unsigned char pchan)
{
   unsigned char handler_id;

   handler_id = l4_do_speech_conn_rq(interface, dialed_num, pchan);

   if(handler_id != UNUSED_CID) /* Success */
   {
      /* Store handler_id in application table for later use */
      return(0); /* Success */
   }
   else
      return(1); /* Failure */
}
```

2.2 Example of an outgoing disconnection request

The following is an example of a routine which formats and sends a message across the API to disconnect a call.

```
void
sample_disc_rq(unsigned char handler_id)
{
   struct  l4_ce_l3_msg  l4_ce_struct;

   l4_do_disc_rq(handler_id);
   /* Remove handler_id from application table */
   return();
}
```

2.3 Example of an outgoing data transfer request

The following is an example of a routine which formats and sends a primitive to transmit user data on an established connection. Note that this is a direct copy of a supplied Function-call API routine rather than an abstracted call of a Function-call API routine, as in the above sections. This has been done to show the use of "buffer holes" in data transfers.

```
int
l4_do_data_rq(unsigned char connid, unsigned char *outgoing_data, unsigned short outgoing_length)
{
   struct l4_ce_l3_msg l4_ce_struct;
   unsigned short refnum;
   unsigned char *dataptr;
   struct connparms *conn_ptr;
   int index;
   int ret_value;

   /* Get buffer for data block */
   ret_value = getbuf(HDRSSIZE + outgoing_length, &dataptr, &refnum);
   if(ret_value) /* Buffer too big to allocate */
      return(-2);

   /*
    *  Copy data into buffer, allowing for "hole" for L2/L3 protocol data use.
    *  Note that this extra copy could be removed if the data (and length) passed in already allowed for the "hole" and
    *  the routine passed in the refnum for a preobtained data buffer along with the outgoing_data pointer.
    */
   for(index = 0;index < outgoing_length;index++)
      *(dataptr + HDRSSIZE + index) = *(outgoing_data + index);

   l4_ce_struct.primcode = N_DATA_RQ;
   l4_ce_struct.connid     = connid;
   l4_ce_struct.dataptr     = dataptr;
   l4_ce_struct.refnum     = refnum;
   l4_ce_struct.datalen     = HDRSSIZE + outgoing_length;
   l4_ce_struct.receipt      = NOACK;  /* Primitive ready to be processed */

   l4_ceproc(&l4_ce_struct, LOCPROC); /* Call top of stack to process */

   if(l4_ce_struct.receipt == CMDACK)   /* Primitive processed without error */
      return(0);     /* Return successful */
    return(-1);   /* ERROR */
}
```

2.4 Example of incoming message processing

The following is an example of a routine which can be called from the TsLink3 routine "ce_l4_snd()" to process messages sent "up" from the stack.

```
void
handle_ce_l4_prim(struct l4_ce_l3_msg *ce_l4_msg_ptr)
{
   struct connparms *conn_ptr;
   int index;

   /* May want to pass along more optional parameters – or process the code within this routine */
   switch(ce_l4_msg_ptr->primcode)
   {
      case N_CONN_IN:   /* INCOMING CALL INDICATION */
         conn_ptr = (struct connparms ) ce_l4_msg_ptr->dataptr;  /* init temp ptr to IPB parms */
         process_conn_in(ce_l4_msg_ptr->connid, conn_ptr->voice_data, conn_ptr->svctype);
         break;
      case N_CONN_CF:  /* OTHER END HAS ANSWERED AN EARLIER N_CONN_RQ WE SENT */
         process_conn_cf(ce_l4_msg_ptr->connid);  /* process the call confirmation */
         break;
      case N_DISC_IN:  /* OTHER END HAS DISCONNECTED */
         process_disc_in(ce_l4_msg_ptr->connid);  /* process the call disconnection */
         break;
      case N_DATA_IN:  /* user data has been received (not applicable for signaling-only apps) */
         /* Note, no "hole" present in incoming data – header areas have been removed as processed */
         process_incoming_data(ce_l4_msg_ptr->connid, ce_l4_msg_ptr->dataptr, ce_l4_msg_ptr->datalen);
         break;
      default:  /* to handle any messages which are not needed and can be ignored */
         break;
   }
   ce_l4_msg_ptr->receipt = CMDACK;   /* acknowledge the message before returning */
   if(ce_l4_msg_ptr->datalen)  /* if datalen > 0, processed IBP/data buffer must be freed */
      freebuf(ce_l4_msg_ptr->refnum, 0x1234); /* second parameter may be set to unique value for tracing */
}
```