
**NSIF-030-1999**

**(SIF-DN-9712-105R4)**

**NSIF APPROVED DOCUMENT**

SIF PROJECT: CIT/OS Platform Definition

---

TITLE: Low Level TL1 API

---

SOURCE: Lucent Technologies Inc.

CONTACT: Jerome P. Moisand

Tel.: +1-978-960-4637
Fax: +1-978-960-6329
E-mail: jmoisand@lucent.com

---

DATE: November 10th, 1998

---

DISTRIBUTION: NSIF

---

ABSTRACT: This is an NSIF working document that proposes an operating-system independent low-level API for TL1 applications, shielding the application from the underlying communication network and protocol stack.

---

This Document has received the approval of the Network Services and Integration Forum (NSIF)

February 18, 1999

# 1. Introduction

## 1.1 Purpose

The purpose of this document is to define a specification of an operating-system independent low-level API for TL1 applications, insulating the application from the underlying communication network.

The API will be named "Basic TL1 API", or in short "BTL1 API".

## 1.2 Scope

The programming interface specified in this document is intended to be used while developing software on management platforms having to communicate through TL1 messages with other equipment like Network Elements (NEs). Such management platforms will typically be:

a) Craft Terminals (CIT)
b) Operation System using TL1 to manage other systems (e.g. EMS managing NEs)
c) Q-Adapters (e.g. CMISE to TL1)
d) Gateway Network Element (e.g. TL1/X.25 to TL1/OSI or TL1/TCP to TL1/OSI)

APIs used within Network Elements are beyond the scope of this document, although it is an objective that the API described in this document may be used as well for such platforms.

## 1.3 Acronyms

| | |
|---|---|
| API | Application Programming Interface |
| BTL1 API | Basic TL1 API |
| CIT | Craft Interface Terminal |
| CMISE | Configuration Management Information Service Element |
| DLL | Dynamic Linking Library |
| EMS | Element Manager System |
| GNE | Gateway Network Element |
| IP | Internet Protocol |
| IPC | Inter-Process Communication |
| LDAP | Lightweight Directory Access Protocol |
| NE | Network Element |
| OS | Operation System |
| OSI | Open Systems Interconnection |
| QA | Q-Adapter |
| QOS | Quality Of Service |
| RTOS | Real Time Operating System |
| SIF | SONET Interoperability Forum |
| SONET | Synchronous Optical NETwork |
| TCP | Transport Control Protocol |
| TID | Target IDentifier |
| TL1 | Transaction Language 1 |
| XAP | X/OPEN ACSE/Presentation API |

## 1.4 References

The following documents are referenced in this document :

[GR-253] Bellcore GR-253-CORE (1997) : Generic Requirements : SONET Transport Systems : Common Generic Criteria

| | |
|---|---|
| [DYN-LIB] | SIF-017-1998 : Dynamic Library Naming Rules For Use With CIT/OS Applications |
| [SOFT-ARCH] | SIF-CI-9706-045 : SIF OS/CIT Software Platform Architecture Specification |

## *1.5 Revision History*

| ISSUE | DATE | SUMMARY OF CHANGES |
|---|---|---|
| First Draft | 09-Dec-1997 | Very early draft |
| Second Draft | 09-Feb-1998 | More detailed specification |
| Third Issue | 06-Oct-1998 | Full specification |
| Fourth Issue | 10-Nov-1998 | Comments from Iselin SIF meeting incorporated |

# 2. Objectives

Before defining the API, the following objectives have been agreed upon by the CIT/OS Platform Definition SIF working group :

1. Low level API (no assumption about TL1 message syntax)
2. Simple API, easy to understand and use
3. Sophisticated enough for any use of TL1 (small or large number of connections, small or large amounts of data -flow control-, outgoing/incoming calls) on targeted systems
4. Unifying API, hides underlying communication network stack (OSI, X.25, TCP, TR-303, V.24, etc.) and TID mapping to underlying addressing information (OSI address, X.25 address, IP address, etc.)
5. Allows several communication network stacks on a given computer
6. Consistency of external definition with a well-known style like X/OPEN XAP or TCP sockets
7. Can be easily mapped on SIF reduced profiles defined for XAP/LDAP_NARSE
8. Reduce interoperability problems by isolating the TL1 mapping on a given underlying communication network stack and promoting reuse of such a mapping library
9. API independent from operating system peculiarities ; mappings on UNIX, Windows NT, or real-time operating systems shall be feasible
10. Implementation of the API as a shared library (UNIX) or a DLL (Windows NT) shall be feasible
11. Multithreading support shall be included
12. Compromise to be found between portability and implementation/environment-specific extensions

# 3. Programming interface overview

This section gives a high-level description of the functions defined by the BTL1 API.

A more precise description of each function will be found in section 4.

## *3.1 Programming model and terminology*

### 3.1.1 BTL1 API, BTL1 library

This document makes use of the term *BTL1 API* to define the API itself (the interface as provided to the application programmer).

This document makes use of the term *BTL1 library* to designate a possible implementation of a library providing the BTL1 API service to application programmers (sometimes referred as users of the API).

### 3.1.2 TL1 connections

When two systems communicate using TL1 messages through a given communication network, both are typically using one or several "connections" in order to carry TL1 messages.

Such a "connection" is provided thanks to a connection-oriented service provided by the communication network. The most common examples are:
a) an X.25 Virtual Circuit
b) an OSI association
c) a TCP connection

This document makes use of the term *TL1 connection* to designate in a generic way a connection established with a remote system in order to carry TL1 messages.

At a high level, the whole purpose of the API defined in this document is to:
a) manage TL1 connections (connection establishment, connection closure)
b) send and receive TL1 messages via such a TL1 connection

### 3.1.3 TL1 connections and TL1 gateways

In order to cope with differences between parts of the communication network, gateway systems are sometimes used in order to relay TL1 traffic.

One typical example (see GR-253) is to have:
a) Operation System (OS) makes use of X.25 to carry TL1 traffic to/from a Gateway Network Element (GNE)
b) the GNE makes use of OSI associations to carry TL1 traffic from/to the targeted Network Element (NE)

In such a case, there are at least two TL1 connections, one achieved as an X.25 virtual circuit, the other achieved as an OSI association. But relationships might be more complex, for example having only one X.25 virtual circuit established with the GNE, and several OSI associations established between the GNE and several NEs.

### 3.1.4 TL1 communication provider

As explained before, several protocol stacks may be used (depending on the communication network) in order to carry TL1 traffic.

In addition, on a given system, one or several implementations of such protocol stacks may be available.

A given implementation of a library providing the API described in this document may:
a) be specific to a given implementation of a given protocol stack, or
b) work on top of several protocol stacks

Notably in order to cover the latter case (which is important when building a TL1 gateway on a GNE for example), the API defined in this document makes use of a quite common technique in similar APIs (e.g. TLI, XAP, etc) :
a) in order to create a connection endpoint, a "provider" parameter is passed to the corresponding function call
b) that parameter allows to identify which underlying protocol stack is to be used
c) the exact format of that parameter is left open to implementors, although it will usually be an ASCII string

Such a parameter will be said to identify a given *TL1 communication provider*.

### 3.1.5 TL1 connection endpoints

Following OSI terminology, the local view (and the corresponding context of operation) of a given TL1 connection on a given system will be named *TL1 connection endpoint*.

More practically speaking, a TL1 connection endpoint will be materialized by a "handle", a mere pointer to an opaque data structure managed by the BTL1 library itself, allowing it to store and update contextual pieces of information about that TL1 connection.

It is noteworthy that a TL1 connection endpoint may exist before a TL1 connection is fully established. That allows the API user to perform locally some operation on the endpoint before actually try to communicate with the remote system. In a similar way, the TL1 connection might be broken, but the connection endpoint still exists until the API user explicitly deletes it.

A TL1 connection endpoint cannot be used for handling more than one TL1 connection.

### 3.1.6 TL1 connection endpoints and multithreading

The TL1 connection endpoint is the basis for multithreading, a BTL1 library having the minimum constraint of allowing simultaneous operations (through several threads) on distinct TL1 connection endpoints.

But a BTL1 library may or may not support more than one thread at a time to use simultaneously a given connection endpoint (for example, one implementation might allow one thread receiving data and several threads sending data ; another one might not allow that kind of parallelism on a given endpoint).

The exact level of multithreading shall be documented with the BTL1 library, but in order to develop fully portable applications, one should not rely on more that the minimum constraint described before.

### 3.1.7 Attributes and implementation-specific extensions

A given implementation might want to add various extensions, either implementation-specific or specific to a given TL1 communication provider.

In order to keep a good compromise between portability and implementation-specific extensions, the following mechanism is introduced: set or get implementation-specific *attributes* for a given TL1 connection endpoint.

Possible examples (for information only) are:

- ⇨ set some QOS characteristics or some other characteristics to be used when requesting a connection, or when listening/accepting incoming connections
- ⇨ specify the remote address (using communication provider specific address format like an OSI address or an X.25 address) and tell the communication provider to ignore the remote TID when requesting a connection
- ⇨ parameterize some security/authentication/encryption mechanism before requesting a connection or listening/accepting incoming connections
- ⇨ give some acceptance criteria before listening/accepting for incoming connections

- ⇨ give some exit function pointers for extending/modifying some processing (memory allocation, flow control policy, directory lookup, etc)
- ⇨ get some statistics about the connection (e.g. data volume)
- ⇨ get some communication network-specific information about the caller when a TL1 connection is established (e.g. caller's address)
- ⇨ retrieve implementation-specific details about the last error
- ⇨ etc.

## *3.2 Programming interface overview*

The following sections describe in an informal way the programming interfaces, grouped by main functionality.

A more formal, more precise and more detailed description can be found in section 4. In case of conflict between these "informal" sections and section 4, the latter prevails.

### 3.2.1 TL1 connection endpoint

int btl1_open (*hdl, provider, provider_ctx, oflags)
- ⇨ create a new TL1 connection endpoint: "hdl" (a pointer to an opaque type)
- ⇨ bind to the relevant TL1 communication provider

void btl1_close (hdl)
- ⇨ if TL1 connection established, disconnect it
- ⇨ delete the TL1 connection endpoint (free any related resource)

### 3.2.2 TL1 connection management

int btl1_connect (hdl, tid, cflags, timeout)
- ⇨ try to establish a TL1 connection to a remote system identified by its TID (the assumption is that the TL1 communication provider can use some kind of directory service, hence translate that TID into a lower level address usable for the underlying communication network)

int btl1_accept (hdl, cflags, timeout)
- ⇨ wait for an incoming TL1 connection, implicitly accept it

See btl1_close() for disconnecting an established connection.

### 3.2.3 Exchanging TL1 messages with the remote system

int btl1_send (hdl, data_ptr, data_len, timeout)
- ⇨ send a TL1 message defined by (data_ptr, data_len)

int btl1_recv (hdl, data_ptr, data_maxlen, timeout)
- ⇨ receive a TL1 message and return its length

### 3.2.4 Error messages

char* btl1_error (error)

⇨ translate an error number into an ASCII error message

### 3.2.5 Implementation-specific extensions

int btl1_set_attr (hdl, attr_id, attr_ptr, attr_len)
⇨ set a given attribute identified by attr_id to a value defined by (attr_ptr, attr_len)
⇨ depending on the semantics of the attribute, various behaviors & error cases might occur

int btl1_get_attr (hdl, attr_id, attr_ptr, attr_maxlen)
⇨ retrieve the value of a given attribute identified by attr_id
⇨ depending on the semantics of the attribute, various behaviors & error cases might occur

# 4. Programmer reference pages

This section presents the manual pages for the BTL1 API. These pages define the functions which make up the BTL1 API, providing the detailed specifications of parameters and data structures.

The functions are described using ANSI-C prototyping. Implementations not using such prototyping are feasible, but not recommended.

*Basic TL1 API functions* **btl1_open()**

**NAME**

`btl1_open` - create a TL1 connection endpoint

**SYNOPSIS**

```
#include "btl1.h"

int btl1_open (
   btl1_hdl_t     *handle_ptr,
   void           *provider,
   void           *provider_ctx
   int            oflags )
```

**DESCRIPTION**

This function creates a TL1 connection endpoint using the TL1 communication provider identified by *provider*. The BTL1 API doesn't assign any specific interpretation to the format of the parameter (although it will usually be an ASCII string null-terminated). However, individual implementations may assign additional semantics to that parameter in order to implement conventions applicable to a particular operating system and/or underlying communication network environment.

The *oflags* argument is a bit mask used to control certain aspects of how the *btl1_open()* invocation is handled. Legal values for the *oflags* argument are formed by OR'ing together zero, or more of the flags described below :

*No flags currently defined.*

If the TL1 connection endpoint is successfully created, then the location pointed to by *handle_ptr* is set. The corresponding value shall be used in subsequent calls to the BTL1 API to identify the TL1 connection endpoint to be processed.

The *provider_ctx* parameter is reserved for future extensions (like the one proposed in Appendix D) and should be set as a null pointer unless the BTL1 library is extended to support such a concept. The semantics of that parameter depends on the TL1 communication provider identified by *provider*.

**RETURN VALUE**

On success, *btl1_open()* returns a null value. Otherwise, a negative value is returned.

**ERRORS**

| | |
|---|---|
| [BTL1_BAD_FLAGS] | The specified combination of flags is invalid. |
| [BTL1_RESOURCE_SHORTAGE] | The operation can't be completed due to a memory shortage. |
| [BTL1_UNKNOWN_PROVIDER] | The communication provider identified by *provider* can't be found. |
| [BTL1_LOST_PROVIDER] | The operation can't be completed due to a communication provider unexpected error. |
| [BTL1_UNSPECIFIED] | Error with unspecified semantics |

**NAME**

`btl1_close` - close a TL1 connection endpoint

**SYNOPSIS**

```
#include "btl1.h"

void btl1_close (
   btl1_hdl_t     handle )
```

**DESCRIPTION**

This function frees the resources allocated to support the TL1 connection endpoint identified by *handle*.

If *btl1_close()* is called with a TL1 connection endpoint associated with an active TL1 connection, then the TL1 connection is aborted before the resources are released.

**RETURN VALUE**

No error conditions are reported by this function (which should be robust enough to recover from any internal problem).

**NAME**

`btl1_connect` - initiate a TL1 connection

**SYNOPSIS**

```
#include "btl1.h"

int btl1_connect (
   btl1_hdl_t     handle,
   char           *called_tid,
   int            cflags,
   long           timeout )
```

**DESCRIPTION**

This function attempts to establish a TL1 connection with the remote system identified by the *called_tid* argument, making use of the TL1 connection endpoint identified by *handle*.

The called TID is an ASCII string null-terminated. It is up to the TL1 communication provider to make use of some directory service to translate that TID into an address significant for the underlying communication network (e.g. X.25 address or OSI address, etc).

The *cflags* argument is a bit mask used to control certain aspects of how the *btl1_connect()* invocation is handled. Legal values for the *cflags* argument are formed by OR'ing together zero, or more of the flags described below :

*No flags currently defined.*

The *timeout* parameter is given in milliseconds (although a communication provider might have a less accurate clock). A positive value gives a maximum delay for the operation to complete. A value equal to *BTL1_WAIT_FOREVER* states that there is no time limit. Other values are reserved for future use.

If the TL1 connection is successfully established, then data exchanges may be performed in both directions using *btl1_send ()* and *btl1_recv()*.

**RETURN VALUE**

On success, *btl1_connect()* returns a null value. Otherwise, a negative value is returned.

**ERRORS**

| | |
|---|---|
| [BTL1_BAD_FLAGS] | The specified combination of flags is invalid. |
| [BTL1_BAD_HANDLE] | The handle argument doesn't correspond to a valid BTL1 handle. |
| [BTL1_CONNECTED] | There is a TL1 connection currently established with that handle, it can't be re-used for another connection. |
| [BTL1_LOST_PROVIDER] | The operation can't be completed due to a communication provider unexpected error. |
| [BTL1_RESOURCE_SHORTAGE] | The operation can't be completed due to a memory shortage. |
| [BTL1_PEER_ADDR_UNKNOWN] | The remote system or the communication network rejected the call due to an addressing problem (e.g. wrong address) |

| | |
|---|---|
| [BTL1_PEER_P_UNREACHABLE] | The remote system or the communication network rejected the call for permanent (or at least non transient) reasons (e.g. peer TL1 agent is disabled) |
| [BTL1_PEER_T_UNREACHABLE] | The remote system or the communication network rejected the call for transient reasons (e.g. resource shortage) |
| [BTL1_PEER_REFUSED] | The remote system can be reached, but rejected the call anyway. |
| [BTL1_PROTOCOL_ERROR] | Unexpected protocol error. |
| [BTL1_RESPONDER_ONLY] | This API implementation can't handle outgoing calls. |
| [BTL1_TID_UNKNOWN] | The called TID is not known by the underlying communication network (e.g. by its directory service) |
| [BTL1_TIMEOUT] | The operation didn't complete due to a timeout. |
| [BTL1_UNSPECIFIED] | Error with unspecified semantics |

**NAME**

`btl1_accept` - wait for an incoming connection

**SYNOPSIS**

```
#include "btl1.h"

int btl1_accept (
   btl1_hdl_t     handle,
   int            cflags,
   long           timeout )
```

**DESCRIPTION**

This function listens for an incoming TL1 connection, and accepts it when it comes, making use of the TL1 connection endpoint identified by *handle*.

The *cflags* argument is a bit mask used to control certain aspects of how the *btl1_accept()* invocation is handled. Legal values for the *cflags* argument are formed by OR'ing together zero, or more of the flags described below :

*No flags currently defined.*

The *timeout* parameter is given in milliseconds (although a TL1 communication provider might have a less accurate clock). A positive value gives a maximum delay for the operation to complete. A value equal to *BTL1_WAIT_FOREVER* states that there is no time limit. Other values are reserved for future use.

If a TL1 connection is successfully accepted, then data exchanges may be performed in both directions using *btl1_send ()* and *btl1_recv()*.

*btl1_accept()* allows the API user to listen and process one incoming connection. In order to listen for multiple incoming connections, the user needs to use either multiple threads -each creating a TL1 connection endpoint then calling *btl1_accept()*-, or to use the extension described in appendix D.

**RETURN VALUE**

On success, *btl1_accept()* returns a null value. Otherwise, a negative value is returned.

**ERRORS**

| | |
|---|---|
| [BTL1_BAD_FLAGS] | The specified combination of flags is invalid. |
| [BTL1_BAD_HANDLE] | The handle argument doesn't correspond to a valid BTL1 handle. |
| [BTL1_CONNECTED] | There is a TL1 connection currently established with that handle, it can't be re-used for another connection. |
| [BTL1_INITIATOR_ONLY] | This API implementation can't handle incoming calls. |
| [BTL1_RESOURCE_SHORTAGE] | The operation can't be completed due to a memory shortage. |
| [BTL1_LOST_PROVIDER] | The operation can't be completed due to a communication provider unexpected error. |
| [BTL1_PROTOCOL_ERROR] | Unexpected protocol error. |

| | |
|---|---|
| [BTL1_TIMEOUT] | The operation didn't complete due to a timeout. |
| [BTL1_UNSPECIFIED] | Error with unspecified semantics |

*Basic TL1 API functions* **btl1_send()**

**NAME**

`btl1_send` - send a TL1 message over the TL1 connection

**SYNOPSIS**

```
#include "btl1.h"

int btl1_send (
   btl1_hdl_t     handle,
   char           *data_ptr,
   int            data_len,
   long           timeout )
```

**DESCRIPTION**

This function is used to send a TL1 message to the peer. *handle* identifies the TL1 connection endpoint for which the TL1 message is to be sent.

The TL1 message is extracted starting from *data_ptr* location, up to *data_len* octets. If a null ('\0') character is located at the end of the message, it shall not accounted in the message length.

The *timeout* parameter is given in milliseconds (although a communication provider might have a less accurate clock). A positive value gives a maximum delay for the operation to complete. A null value might be used for trying to send without any blocking condition. A value equal to *BTL1_WAIT_FOREVER* states that there is no time limit. Other values are reserved for future use.

A blocking condition might be exercised by the communication due to a flow control condition. When the *btl1_send()* function returns successfully, that means that the communication provider copied, internally queued and scheduled the transmission of the TL1 message. It gives no guarantee that the remote system received it or will receive it. But unless a fatal communication error occurs (e.g. BTL1_LOST_CONNECTION), the TL1 message will be reliably transmitted as soon as possible.

Once the function call has completed, the API user keeps the ownership of the data, and has therefore the responsibility to free the corresponding buffer space if it was dynamically allocated.

**RETURN VALUE**

On success, *btl1_send()* returns a null value. Otherwise, a negative value is returned.

**ERRORS**

| | |
|---|---|
| [BTL1_BAD_HANDLE] | The handle argument doesn't correspond to a valid BTL1 handle. |
| [BTL1_RESOURCE_SHORTAGE] | The operation can't be completed due to a memory shortage. |
| [BTL1_INVALID_PARAMETER] | *data_maxlen* is negative or null or too long, or *data_ptr* is null. |
| [BTL1_NO_CONNECTION] | There is no TL1 connection currently established with that handle. |
| [BTL1_LOST_CONNECTION] | The connection with the TL1 peer has been broken. |

| | |
|---|---|
| [BTL1_LOST_PROVIDER] | The operation can't be completed due to a communication provider unexpected error. |
| [BTL1_PROTOCOL_ERROR] | Unexpected protocol error. |
| [BTL1_TIMEOUT] | The operation didn't complete due to a timeout, probably due to a flow control condition |
| [BTL1_UNSPECIFIED] | Error with unspecified semantics |

**NAME**

`btl1_recv` - receive a TL1 message over the TL1 connection

**SYNOPSIS**

```
#include "btl1.h"

int btl1_recv (
   btl1_hdl_t     handle,
   char           *data_ptr,
   int            data_maxlen,
   long           timeout )
```

**DESCRIPTION**

This function is used to receive a TL1 message from the peer. *handle* identifies the TL1 connection endpoint for which the TL1 message is to be received.

The TL1 message is stored starting from *data_ptr* location, up to *data_maxlen* (minus one) octets. A null ('\0') character is automatically appended at the end of the message (but not accounted in the returned length).

If the received TL1 message is longer than *data_maxlen* (minus one) octets, then the message is truncated to *data_maxlen* octets (without a null character appended), and a return code of [BTL1_TRUNCATED] is given back. The next call to *btl1_recv()* will give the next part of the TL1 message.

The *timeout* parameter is given in milliseconds (although a communication provider might have a less accurate clock). A positive value gives a maximum delay for the operation to complete. A null value might be used for simply checking if some TL1 message is available. A value equal to *BTL1_WAIT_FOREVER* states that there is no time limit. Other values are reserved for future use.

If the *btl1_recv()* function is not called, then the TL1 communication provider is not allowed to drop received data, but it should exercise some flow-control conditions on the sender starting from some threshold.

It is the API user responsibility to provide buffer space for the incoming data. Once the function call has completed, the API user keeps the ownership of the data, and has therefore the responsibility to free the corresponding buffer space if it was dynamically allocated.

**RETURN VALUE**

On success, *btl1_recv()* returns a positive value, the length of the received TL1 message. Otherwise, a negative value is returned.

**ERRORS**

| | |
|---|---|
| [BTL1_BAD_HANDLE] | The handle argument doesn't correspond to a valid BTL1 handle. |
| [BTL1_RESOURCE_SHORTAGE] | The operation can't be completed due to a memory shortage. |

| | |
|---|---|
| [BTL1_INVALID_PARAMETER] | *data_maxlen* is negative or null, or *data_ptr* is null. |
| [BTL1_LOST_CONNECTION] | The connection with the TL1 peer has been broken. |
| [BTL1_LOST_PROVIDER] | The operation can't be completed due to a communication provider unexpected error. |
| [BTL1_NO_CONNECTION] | There is no TL1 connection currently established with that handle. |
| [BTL1_PROTOCOL_ERROR] | Unexpected protocol error. |
| [BTL1_TIMEOUT] | The operation didn't complete due to a timeout. |
| [BTL1_TRUNCATED] | The received message didn't fit in *data_maxlen* octets and has been truncated. |
| [BTL1_UNSPECIFIED] | Error with unspecified semantics |

**NAME**

`btl1_set_attr` - change an attribute value

**SYNOPSIS**

```
#include "btl1.h"

int btl1_set_attr (
   btl1_hdl_t     handle,
   long           attr_id,
   void           *attr_ptr,
   int            attr_len )
```

**DESCRIPTION**

This function is used to set the value of a given attribute for the TL1 connection endpoint identified by *handle*.

The attribute value is made of a set of *attr_len* octets found starting from the *attr_ptr* location. There is no assumption about the structure of that value (might be an ASCII string, an integer value, a "C" structure, etc).

The exact semantics of the attribute value is defined by the attribute identifier *attr_id*. The exact semantics of setting it is attribute-dependent (it might be more than simply setting a value, but may actually trigger some processing).

Positive identifiers are reserved for implementation-specific purposes. Beware that making use of such a mechanism would be at the expense of the portability of the user application (introducing a dependency with a given implementation).

Negative identifiers are reserved for standardized values defined in this document, and recognized by all conformant implementations. No such identifier is defined at this time.

**RETURN VALUE**

On success, *btl1_set_attr()* returns a null value. Otherwise, a negative value is returned.

**ERRORS**

| | |
|---|---|
| [BTL1_BAD_HANDLE] | The handle argument doesn't correspond to a valid BTL1 handle. |
| [BTL1_RESOURCE_SHORTAGE] | The operation can't be completed due to a memory shortage. |
| [BTL1_INVALID_IDENTIFIER] | The attribute identifier is not known by the implementation. |
| [BTL1_INVALID_VALUE] | The attribute value for the semantics defined by *btl1_set_attr()* is invalid. |
| [BTL1_LOST_CONNECTION] | The connection with the TL1 peer has been broken. |
| [BTL1_READ_ONLY] | The attribute identifier is legal, but the corresponding value can't be changed |
| [BTL1_LOST_PROVIDER] | The operation can't be completed due to a communication provider unexpected error. |
| [BTL1_SPECIFIC_???] | Specific errors may be defined in an implementation-specific |

| | |
|---|---|
| | way. Their value shall be less than -1000. |
| [BTL1_UNSPECIFIED] | Error with unspecified semantics |

*Basic TL1 API functions* **btl1_get_attr()**

**NAME**

`btl1_get_attr` - change an attribute value

**SYNOPSIS**

```
#include "btl1.h"

int btl1_get_attr (
   btl1_hdl_t     handle,
   long           attr_id,
   void           *attr_ptr,
   int            attr_maxlen )
```

**DESCRIPTION**

This function is used to get the value of a given attribute for the TL1 connection endpoint identified by *handle*.

The attribute value is made of a set of up to *attr_maxlen* octets stored starting from the *attr_ptr* location. There is no assumption about the structure of that value (might be an ASCII string, an integer value, a "C" structure, etc). The actual length of the attribute value is returned by the function (in case of success).

The exact semantics of the attribute value is defined by the attribute identifier *attr_id*. The exact semantics of getting it is attribute-dependent (it might be more than simply getting a value and might actually trigger some processing).

Positive identifiers are reserved for implementation-specific purposes. Beware that making use of such a mechanism would be made at the expense of the portability of the user application (introducing a dependency with a given implementation).

Negative identifiers are reserved for standardized values defined in this document, and recognized by all conformant implementations. No such identifier is defined at this time.

**RETURN VALUE**

On success, *btl1_set_attr()* returns the length of the attribute value. Otherwise, a negative value is returned.

**ERRORS**

| | |
|---|---|
| [BTL1_BAD_HANDLE] | The handle argument doesn't correspond to a valid BTL1 handle. |
| [BTL1_RESOURCE_SHORTAGE] | The operation can't be completed due to a memory shortage. |
| [BTL1_INVALID_IDENTIFIER] | The attribute identifier is not known by the implementation. |
| [BTL1_INVALID_LENGTH] | The *attr_maxlen* value is either negative or not large enough for the current attribute value. The attribute value is not returned. |
| [BTL1_LOST_CONNECTION] | The connection with the TL1 peer has been broken. |
| [BTL1_LOST_PROVIDER] | The operation can't be completed due to a communication provider unexpected error. |

| | |
|---|---|
| [BTL1_WRITE_ONLY] | The attribute identifier is legal, but the corresponding value can't be read |
| [BTL1_SPECIFIC_???] | Specific errors may be defined in an implementation-specific way. Their value shall be less than -1000. |
| [BTL1_UNSPECIFIED] | Error with unspecified semantics |

**NAME**

`btl1_error` - return an error message

**SYNOPSIS**

```
#include "btl1.h"

char * btl1_error (
   int            error )
```

**DESCRIPTION**

This function returns a pointer to an error message that describes the error indicated by *error*. That error has been previously given back to the user, as a negative return code, following the use of a BTL1 API function with the TL1 connection endpoint identified by *handle*.

The error message is an ASCII, null-terminated, read-only, printable text string. The message itself will usually be in plain English, but the contents of such error messages is not standardized by this document.

The message pointer points to an internal, read-only, buffer area. If the caller wishes to retain the message text, before calling *btl1_error()* again, then the text should be copied to some private storage.

**RETURN VALUE**

Upon completion, a pointer to the appropriate error message is returned.

**ERRORS**

No error conditions are reported by this function. Special cases like an unknown error code will be processed returning a pointer to an appropriate error text message.

## 5. Include files

That sections details mandatory parts of the "btl1.h" include file. Implementation-specific content might be added. The include file shall contain at least the following:

- *typedef something btl1_hdl_t* // the handle might be a pointer or an integer
- *#define BTL1_MAX_DATA_LEN 4096* // maximum size of a TL1 message - GR-253 1997
- *#define BTL1_WAIT_FOREVER -1L* // "timeout" parameter special value
- *#define* of attribute ids (proprietary definitions) if any
- *#define* for error codes (see the table below)

| | |
|---|---|
| [BTL1_BAD_FLAGS] | -20 |
| [BTL1_BAD_HANDLE] | -21 |
| [BTL1_CONNECTED] | -30 |
| [BTL1_INITIATOR_ONLY] | -90 |
| [BTL1_INVALID_IDENTIFIER] | -91 |
| [BTL1_INVALID_LENGTH] | -92 |
| [BTL1_INVALID_PARAMETER] | -93 |
| [BTL1_INVALID_VALUE] | -94 |
| [BTL1_LOST_CONNECTION] | -120 |
| [BTL1_LOST_PROVIDER] | -121 |
| [BTL1_NO_CONNECTION] | -140 |
| [BTL1_RESOURCE_SHORTAGE] | -150 |
| [BTL1_PEER_ADDR_UNKNOWN] | -160 |
| [BTL1_PEER_P_UNREACHABLE] | -161 |
| [BTL1_PEER_T_UNREACHABLE] | -162 |
| [BTL1_PEER_REFUSED] | -163 |
| [BTL1_PROTOCOL_ERROR] | -164 |
| [BTL1_READ_ONLY] | -180 |
| [BTL1_RESPONDER_ONLY] | -181 |
| [BTL1_TID_UNKNOWN] | -200 |
| [BTL1_TIMEOUT] | -201 |
| [BTL1_TRUNCATED] | -202 |
| [BTL1_UNKNOWN_PROVIDER] | -210 |
| [BTL1_UNSPECIFIED] | -211 |
| [BTL1_WRITE_ONLY] | -230 |
| [BTL1_SPECIFIC_???] | less than -1000 |

## 6. Definition or export file

Definition or export files shall be defined following the naming rules defined in `[DYN-LIB].`

# 7. Requirements for conforming implementations

The following table describes the specific items that implementations claiming conformance to these requirements must support.

Optional requirements are tagged with an “O” mark ; mandatory requirements are tagged with an “M” mark.

Such requirements may be strengthened for a given platform type (e.g. in a document like [SOFT-ARCH]), only minimal requirements shared by every platform are defined here.

| No | Description | M/O | Reference |
|---|---|---|---|
| R1 | BTL1 "initiator" profile | O [1] | 3. 4. 5. 6. *btl1_accept* will always report an error |
| R2 | BTL1 "responder" profile | O [1] | 3. 4. 5. 6. *btl1_connect* will always report an error |
| R3 | BTL1 "initiator and responder" profile | O [1] | 3. 4. 5. 6. no restriction on *btl1_connect* or *btl1_accept* |

Notes :

[1] R1, R2 and R3 are mutually exclusive. Support for one of them is mandatory.

# 8. Future extensions

This section is not part of the specification. It only gives a few hints for possible future directions.

- ⇨ C++ and/or JAVA native mapping
- ⇨ IDL mapping

Note that the BTL1 API as defined in this specification may be used in applications not written in "C", making use of "C" calls encapsulation (e.g. usual technique in C++ or JAVA).

# Appendix A : XAP/LDAP TL1 communication provider

That section is for information only.

It gives a high-level view on how a mapping of the BTL1 API could be achieved on top of the XAP and LDAP_NARSE APIs. Implementors may make other choices.

btl1_open :
- ⇨ ap_open with basic memory handler
- ⇨ store XAP descriptor in TL1 connection endpoint descriptor (structure pointed by the handle)
- ⇨ set env. attributes AP_LIB_SEL (AP_LIB_VER1) & AP_BIND_PADDR (use wildcard NSAP and GR-253 Tsel/Ssel/Psel)
- ⇨ reset AP_NDELAY flag in AP_FLAGS (blocking mode)

btl1_close :
- ⇨ more or less direct mapping on ap_close
- ⇨ might be cleaner to send A_ABORT_REQ before if TL1 connection established

btl1_connect:
- ⇨ translate the TID to an OSI address thanks to LDAP_NARSE (using a Distinguished Name "cn=TID, cn=TID, npx=."), set AP_REM_PADDR accordingly
- ⇨ set env. attributes AP_ROLE_ALLOWED (AP_INITIATOR), then ap_bind
- ⇨ map TL1 connections on XAP associations (one to one) following GR-253 procedure
- ⇨ set AP_CNTX_NAME, AP_PCDL accordingly to GR-253 (see Editor's note below)
- ⇨ use ap_snd for issuing A_ASSOC_REQ
- ⇨ use ap_rcv to wait for A_ASSOC_CNF, or (P)ABORT_IND
- ⇨ add timeout processing

btl1_accept:
- ⇨ set env. attributes AP_ROLE_ALLOWED (AP_RESPONDER) & AP_QLEN (1), then ap_bind
- ⇨ mapping on ap_rcv, expect only A_ASSOC_IND
- ⇨ reply with A_ASSOC_RSP
- ⇨ map TL1 connections on XAP associations (one to one) following GR-253 procedure
- ⇨ add timeout processing

btl1_send :
- ⇨ mapping on ap_snd with P_DATA_REQ
- ⇨ if flow control blocked, wait until unblocked with timeout processing
- ⇨ map AP_HANGUP error code to BTL1_LOST_CONNECTION

btl1_recv :
- ⇨ mapping on ap_rcv, expect mainly P_DATA_IND, but abort/release may occur
- ⇨ if P_DATA_IND, copy XAP cdata (full SDU, handle AP_MORE) into TL1 user data space, free XAP cdata & vbuf
- ⇨ map (P)ABORT_IND on BTL1_LOST_CONNECTION return code
- ⇨ if RELEASE_IND, reply with RELEASE_RSP, then BTL1_LOST_CONNECTION return code
- ⇨ add timeout processing while waiting for data

# Appendix B : other TL1 communication providers

This appendix is for information only.

The information given here is imprecise and shall not be considered as a reference, but simply as preliminary ideas about possible various mappings of the BTL1 API.

**B.1 Mapping on data link (TR-303, V.24)**

The "connection" is mapped on the availability of the data link:
a) data link set up for TR-303/LAP-D
b) carrier signal for V.24

For V.24, an "End Of Line" character will be used for delimiting the TL1 message.

No more than one TL1 connection may be established on a given data link.

**B.2 Mapping on TCP**

One TL1 connection => one underlying TCP connection.

Some directory service has to be used for mapping the TID to an IP address (TCP port being "well-known"). DNS might be used for that purpose, the TID being the first naming component of an IP host name (e.g. "tttt.ooo.cc") where "ttt" stands for the TID, "ooo" stands for an organization, "cc" stands for "com" or "org" or a country like "fr". An alternative is to use the LDAP API, following principles described in LDAP_NARSE (possibly with a mapping on the true LDAP protocol).

Data needs to be "packetized", adding some length in front of the TL1 message or preferably some "end of line" character at the end of the message. On receiver side, read until a full message is received.

No keep-alive mechanism should be activated at the TCP level, since it is common practice to have TL1 messages polling the peer on a regular basis.

SIF plans to define a precise mapping of TL1 over TCP (or over TELNET), but details were not available when this document has been written.

**B.3 Mapping on X.25**

One TL1 connection => one underlying X.25 virtual circuit (SVC or PVC). Note that a GNE might act as a complex multiplexer between TL1/X25 connections and TL1/OSI connections, but that would occur at the application level.

Some directory service has to be used for mapping the TID to an X.25 address (e.g. local table or X.500/LDAP).

X.25 incoming calls dispatch is unclear: a special X.25 Protocol Id allows to distinguish TL1 traffic from something else (like CLNP/X25) ? Or X.25 sub-address ? GR-253 says nothing about that.

See GR-253 for more mapping information.

# Appendix C : rationale for some choices

That section is for information only.

It gives some clues explaining many choices made when defining the BTL1 API.

*No separated open/bind like in XAP/XTI*
What would be the added value ? Note that there is no calling address concept with TL1.

*Then why separate open from connect/accept*
Because that gives the opportunity to tune various things using btl1_set_attr() and btl1_get_attr().

*No explicit acceptance of incoming call (e.g. OSI connect_ind/connect_rsp or XTI listen/accept*
Far more simple. More conformant to TCP style. You can always abort the call later on. Negotiation/Acceptance criteria data might be set before through btl1_set_attr().

*No "qlen", no switch between fd with incoming call and fd accepting the call like in sockets/XTI*
The user might simply have N threads waiting on a btl1_accept() call. Far more simple for the implementor and more deterministic (no ugly race condition).

*No fd, but "handle"*
That's the style of modern APIs (see LDAP API for example, or any object-oriented stuff). That allows a far more simple (and more efficient) implementation easily multi-threaded. And that will ease a C++ or a Java mapping.

*No orderly release*
Would make some sense only with an OSI-based TL1 communication provider (can’t be achieved with X.25, semantics of graceful close too limited for TCP -no peer user acknowledgement-). And anyway, there is a TL1 logoff message, so no need to duplicate functionality.

*Only synchronous primitives*
First, far more simple (and more portable) paradigm. Next, multi-threading is widely available now, let's use it. Failed attempts to define portable asynchronous interfaces without multi-threading is a long story …

*Timeouts everywhere*
Attempt to simplify the life of implementors (and users) not trying to build some interrupt/signal system (hence a operating system dependency) for killing a blocking call.

*Proprietary extension mechanism*
Stable external definition even if proprietary extensions => ok for stable DLL/SharedLib definition files. Tradeoff allowing to define a portable interface for people doing simple (and portable) things, but keeping the door opened to introduce implementation-dependent subtleties (at the expense of losing portability) for special cases.

*Input flow control*

Handled in a simple & intuitive way : don't read if you want to block. Only drawback : can't survey disconnect without reading … well, TCP/sockets programmers never felt the need for such a feature.

*Output flow control*

Provider blocks you when queue full. Simple & allows multiple implementation choices.

*No "IDU" concept*

What would be the point with TL1 messages <= 4096 octets (cf. maximum size of TL1 messages as defined by [GR-253] ?

*Data sent/received remains in user space*

Far more simple & robust. Might imply memory copies, but well, TL1 data should not involve very large volumes for individual messages. Also simplifies the API.

*Null-terminated string subtleties*

Attempt to ease the user's life.

*Practical examples of the use of that strange "provider" parameter*

It will usually be an ASCII string identifying a device driver name, an IPC name (e.g. pipe), a DLL name, etc. Style used in TLI/XAP/LDAP and many others APIs.

One interesting example : a GNE with TL1/X25 on one side, TL1/OSI on the other side. That implies *two* independent communication stacks, but the GNE has to use both and to differentiate them thanks to the *provider* parameter.

Instead of being an ASCII string, that parameter might be a preprocessor definition (#define) hiding more complex information (e.g. a table of pointers for functions allowing to interface with a given communication provider).

A more sophisticated use is also described in Appendix D.

*No explicit parameter for the error code like in XAP*

Seems so simple to use a return code as a return code ! Combining that with the UNIX technique of negative error code and optional positive value giving pieces of information (e.g. a length) simplifies the interfaces.

*No return code to " tl1_close"*

What kind of error processing can you do after closing ? try to close again ? A good policy is to be sure that the API provider will clean up as much as possible, even if some troubles are encountered.

*Why have these apparently not so useful "flags" parameters*

Because it's a proven technique for expanding an API definition in the future while staying upward-compatible.

*Strange non-continuous numbering of error codes*

A very easy little riddle...

# Appendix D : extensions for extended synchronization

That section is for information only.

It describes additional functions & mechanisms that might help build a better service for implementations on operating systems having strong constraints about the number of threads or the inter-process communication channels.

The described extension is upward-compatible, which means that applications not using these extensions shall work without problems even if an implementation of the BTL1 API actually includes these extensions.

**D.1 Introduction**

The BTL1 API described in the main part of this document relies on an extensive use of multithreading when an application needs to handle several simultaneous connections (at least one thread per TL1 connection where incoming data can be expected at any time). It also possibly makes a more hidden assumption about IPC channels between a BTL1 library and the underlying software (at least one channel per TL1 connection endpoint).

Although perfectly reasonable on many operating systems (e.g. UNIX, Windows NT, RTOS with a small number of TL1 connections to handle), such assumptions might not be reasonable in some uncommon cases (e.g. RTOS with a large number of connections to handle, old UNIX flavors without multithreading support).

One should note that on an RTOS, the usual way to emulate a thread behavior is to use one task for that purpose since addressing space is usually shared by tasks. But the number of tasks are usually pre-configured to a relatively small value on an RTOS.

It is then necessary to use a more complex model where the application may have an event-driven design, typically waiting on inputs from the various connection endpoints, then process such inputs, then go back wait for the next incoming event.

**D.2 A possible extension**

One possible answer to the issue is to use a paradigm like the UNIX poll/select, where the application explicitly specifies the list of connection endpoints on which an incoming event has to be waited for.

But that paradigm is well-known for:

- be very CPU-consuming when the number of connection endpoints is large (since the list of endpoints has to be parsed each time the application comes back to wait for new inputs)
- be somewhat complex to implement not knowing in advance the exact list of connection endpoints that will be submitted to the poll/select (implies internal queue management for each connection endpoint and no easy way to rely on inherent properties of underlying IPCs)
- raise some multithreading & implementation issues on RTOS where a global link-editing has to occur and where the addressing space is fully shared (hence precluding use of internal global variables per instance of library loaded in memory for a given process).

For all these reasons, the poll/select style doesn’t seem a proper answer.

**D.3 Another possible extension**

The main idea is to create *provider contexts* (for a given TL1 communication provider) that may keep track of a set of communication channels with the underlying communication stack. Such a mechanism then easily allows to wait for incoming events on any connection endpoint that has been established based on that provider context.

Two additional functions are needed for creating/deleting such provider contexts:

int btl1_create_provider_ctx (void *provider, void **provider_ctx, int oflags)
- ⇨ create a new context for a given TL1 communication provider identified by "provider"
- ⇨ the new context is returned in "provider_ctx" and can be given to *btl1_open* as its "provider_ctx" parameter
- ⇨ oflags reserved for future use, must be zero

void btl1_delete_provider_ctx (void *provider_ctx)
- ⇨ can only be called if no TL1 connection endpoint is current established with that provider context
- ⇨ delete the provider context (free any related resource)

One additional function is needed to synchronize on incoming events, for any TL1 connection endpoint that has been created through *btl1_open* and the use of that provider context.

int btl1_wait_event (void *provider_ctx, btl1_hdl_t *handle, long timeout)
- ⇨ wait for incoming events ; returns immediately if some event is available
- ⇨ if the return code is positive, then "handle" gives the TL1 connection endpoint on which some event is available and the relevant BTL1 function has to be called
- ⇨ if the return code is positive, then it identifies the type of incoming event that is available, allowing the application to call the relevant BTL1 function. The following list may be defined:

1. BTL1_EVENT_ACCEPT (incoming connection, call *btl1_accept*)
2. BTL1_EVENT_INPUT_DATA (incoming data, call *btl1_recv*)
3. BTL1_EVENT_ABORT (connection broken, call *btl1_close*)

Not all connection endpoints are intended (or ready if one intends to set attributes) for incoming connections. Therefore the API user shall call *btl1_accept* (with a null timeout) to enable a connection endpoint to be associated with an incoming call.

One might optionally add to that list BTL1_EVENT_OUTPUT_READY (outgoing flow control unblocked, call *btl1_send* when data has to be sent). This would allow a more flexible use of *btl1_send* (use it with an unblocking behavior -null timer- and synchronize on *btl1_wait_event* when an BTL1_TIMEOUT error code is returned indicating when the output flow is blocked).

One might also want to add the ability to set/get attributes on provider contexts in order to change the behavior of a given provider context or to retrieve information about its current status, hence to add functions like:
- ⇨ int btl1_set_ctx_attr (provider_ctx, attr_id, attr_ptr, attr_len)
- ⇨ int btl1_get_ctx_attr (provider_ctx, attr_id, attr_ptr, attr_maxlen)

When that provider context technique is used, it is recommended to BTL1 library implementors to:
- always support a null value for "provider_ctx" in the *btl1_open* function
- design some "fairness" algorithm in *blt1_wait_event* to not introduce some undesirable priority between the processing of the various connections.

- if the operating system allows it, provide an implementation of an BTL1 library that allows any regular BTL1 function to be called by a given thread, while another thread is calling *btl1_wait_event*. Beware that this raises some non-obvious technical issues notably when a connection endpoint is closed.

Since that extension introduces some significant complexity in the API definition as well as in its implementation, this extension is viewed as "experimental" and not a formal part of the BTL1 API specification.