COMMON-ISDN-API

Version 2.0

February 1994

Author:
COMMON-ISDN-API working group

Editor:
AVM GmbH
Voltastr. 5
D-13355 Berlin
Germany

First Edition (February 1994)

published by:
Telekom ROLAND
Eurolab
COMMON-ISDN-API working group
Postfach 91 00
D-55541 Bad Kreuznach
Germany

Contents

SPECIAL NOTICES

Reader's Guide

This document specifies **COMMON-ISDN-API Version 2.0**. Readers should be generally familiar with ISDN concepts.

Chapter 1 serves as an introduction into the general concepts of **COMMON-ISDN-API** as an application interface from a global point of view. Chapter 2 provides a detailed look at **COMMON-ISDN-API**'s position relative to the OSI layers and introduces the different supported protocol options. Chapter 3 describes the basic mechanisms that ensure operating system independence such as messages, message structures and the used message protocol. Chapter 4 describes the operations which are necessary to exchange messages between **COMMON-ISDN-API** and applications. Chapter 5 and 6 specify in detail the functionality and coding of each message and parameter. Chapter 7 defines the allowed actions in different states of a connection by introducing a presentation of state diagrams. Chapter 8 includes all operating system dependent **COMMON-ISDN-API** operations to exchange messages. It is divided into subchapters for each operating system supported by **COMMON-ISDN-API**. Annex A gives an intuitive understanding of how to connect, exchange data and disconnect, exemplified by arrow diagrams. Annex B is added for providing a coding scheme used by **COMMON-ISDN-API** to exchange fax G3 documents between **COMMON-ISDN-API** and applications. The following index lists every message, parameter and operation of **COMMON-ISDN-API.**

Disclaimer

Whilst every care has been taken in the preparation and publication of this document, errors in content, typographical or otherwise, may occur. If you have comments concerning its accuracy, please write to "Telekom ROLAND, Eurolab, COMMON-ISDN-API working group" at the address shown on the back of the title page.

The COMMON-ISDN-API working group makes no representations or warranties with respect to the contents or use of this manual, and specially disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, the COMMON-ISDN-API working group reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions and changes.

Trademarks

The following terms are trademarks of companies, but they are not explicitly shown in this text.

MS-DOS is a registered trademark of Microsoft Corporation.
NetWare is a registered trademark of Novell, Inc.
Novell is a registered trademark of Novell, Inc.
OS/2 is a trademark of International Business Machines Corporation.
UNIX is a registered trademark of UNIX Systems Laboratories Inc.
Windows is a trademark of Microsoft Corporation.

PREFACE

COMMON-ISDN-API (**CAPI**) is an application programming interface standard used to access ISDN equipment connected to basic rate interfaces (**BRI**) and primary rate interfaces (**PRI**). By adhering to the standard, applications can make use of well defined mechanism for communications over ISDN lines, without being forced to adjust to the idiosyncrasies of hardware vendor implementations. ISDN equipment vendors in turn will benefit from a wealth of applications, ready to run with their equipment.

COMMON-ISDN-API is now a well established standard. Potential cost savings were the driving force for **COMMON-ISDN-API** controller and application development. Commercial users in Germany are rapidly migrating to ISDN (**I**ntegrated **S**ervices **D**igital **N**etwork) as the principal vehicle for data exchange of a wide range of formats.

In 1989 manufacturers started to define an application interface which would be accepted in the growing ISDN market. To get an acceptable result, the focus of this standard was the possibility of running the national ISDN protocol, for an ETSI ISDN protocol standard was not available at this time. Work on this application interface was finished in 1990 by a CAPI working group consisting of application providers, ISDN equipment manufacturers, large customers / user groups and DBP Telekom. **COMMON-ISDN-API** Version 1.1 was a great step towards opening the national ISDN market in Germany. Meanwhile almost every German ISDN solution as well as an increasing count of international ones is based on **COMMON-ISDN-API** Version 1.1; there exists a well accepted conformance test laboratory at DBP Telekom.

To reflect on the current situation it can be stated that the international protocol specification is finished and almost every telecommunication provider offers BRI / PRI with protocols based on Q.931 / ETS 300 102. **COMMON-ISDN-API** will be additionally needed for the **DSS1** protocol. Experience in ISDN application interface design, knowledge of the market needs and a large installed base of **COMMON-ISDN-API** solutions (hardware controller and applications on top of different operating systems) result in the necessity of developing a new application interface, usable in international ISDNs.

COMMON-ISDN-API Version 2.0 includes more than 5 years of ISDN business implementation experience in an exploding market. It covers all benefits of CAPI Version 1.1 plus new aspects of ISDN (e.g. Facsimile Group 3 connectivity or video telephony). It is based on Q.931 / ETS 300 102 but not limited to these. It simplifies the development of ISDN applications through many defaults which need not to be programmed. It keeps applications free of ISDN protocol knowledge and thus makes many applications possible.

By using **COMMON-ISDN-API Version 2.0** the international market can exploit the available experience and realise a large growth.

1 INTRODUCTION

COMMON-ISDN-API enables applications to access ISDN adapter boards in a straightforward manner and allows unrestricted use of their functions through a standardised software interface.

Applications which use this interface will not be affected by future expansions or hardware changes. **COMMON-ISDN-API** makes the changes transparent to user application. Future expansions that retain compatibility with existing software base are possible.

COMMON-ISDN-API provides an abstraction of ISDN services that is independent from the underlying network and from the adapters used to connect to the network. It provides an easy-to-use interface for applications and offers a unique access to the different ISDN services like data, voice, fax, video, telephony, etc..

COMMON-ISDN-API provides a base for modular applications development in ISDN systems.

1.1 Scope

This document describes **COMMON-ISDN-API**, the application programming interface for ISDN. **COMMON-ISDN-API** is designed in a message-oriented, event driven way. **COMMON-ISDN-API** will be described in two parts: the main part defines each message used and its message parameter. This part is entirely operating system independent. The other part deals with operations needed to exchange these messages.

The specification of **COMMON-ISDN-API** as such is an application *interface*, however the *implementation* of **COMMON-ISDN-API** designates a kind of *instantiation*, which is actually seen by an application dealing with ISDN communications. The state diagrams shown in chapter 7 explain behaviour of **COMMON-ISDN-API** from a point of view which is set at interface level, but also take the implementation of **COMMON-ISDN-API** as an instantiation (for real states) into consideration.

1.2 Features

COMMON-ISDN-API includes a number of important features.

- Support for basic call features, such as call setup and clearing
- Support for several B channels for data and/or voice connections
- Support for several logical connections for data links within a physical connection
- Possibility of selecting different services and protocols during connection setup and incoming call
- Transparent interface for protocols above layer 3
- Support for one or more Basic Rate Interfaces (Basic Access) as well as Primary Rate Interfaces (Primary Access) on one or more ISDN adapters
- Support of multiple applications
- Operating-system independent messages
- Operating-system dependent exchange mechanism for optimum operating system integration
- Asynchronous event driven mechanism, resulting in high throughput
- Well defined mechanism for manufacturer specific expansions

2 OVERVIEW

COMMON-ISDN-API provides a standardised interface for any number of application programs (applications) to any number of ISDN drivers and ISDN controllers. Applications can be freely assigned to drivers and controllers.

- One application can use one controller
- One application can use more than one controller
- Several applications can share a single controller
- Several applications can share more than one controller

Applications can use different protocols at different protocol levels, **COMMON-ISDN-API** provides a selection mechanism in support of this. **COMMON-ISDN-API** also performs an abstraction from different protocol variants, creating a standardised network access. All connection related data such as connection state, display messages etc. is available to applications at any time.

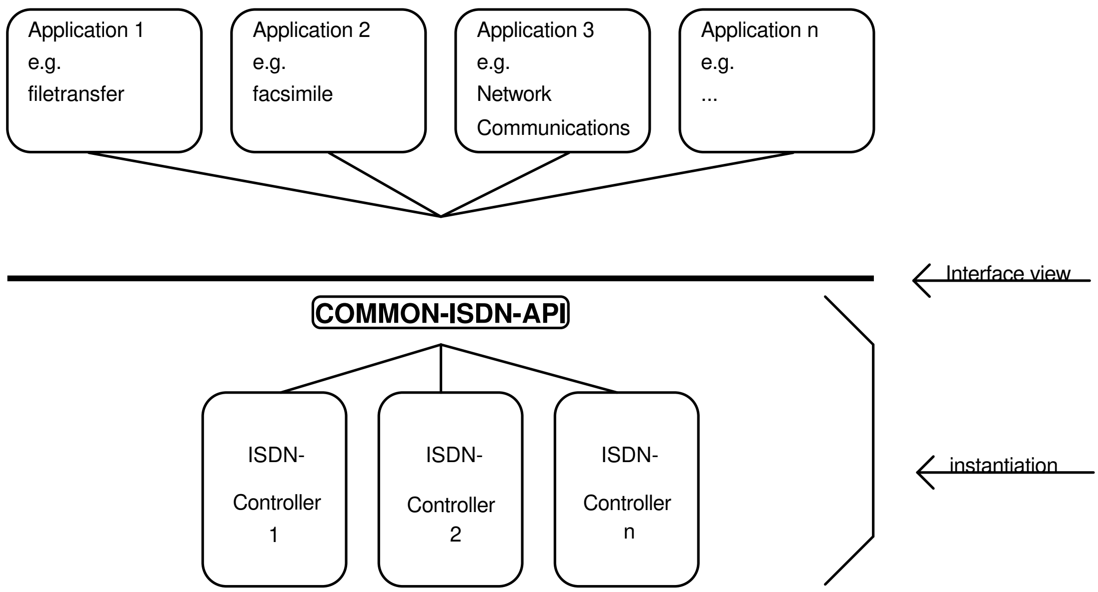


Figure 1: Position of **COMMON-ISDN-API**

COMMON-ISDN-API covers the whole signalling protocol as well as protocol layer 1 to 3 (physical and framing layer, data link layer and network layer) for data channels. The interface of **COMMON-ISDN-API** is located between layer 3 and layer 4 and provides the point of reference for applications and higher level protocols.

COMMON-ISDN-API offers many currently used protocols to applications without deep protocol knowledge. The default protocol is **ISO 7776** (X.75 SLP), i.e. framing protocol **HDLC**, data link protocol **ISO 7776** (**X.75** SLP), and a transparent network layer.

Other supported variants of framing layer are: **HDLC inverted**, **PCM** (bit transparent with byte framing) **64/56** kBit, **V.110** sync / async. **COMMON-ISDN-API** integrates the following data link and network layers: **LAPD** according to Q.921 for **X.25 D-channel** implementation, **PPP** (Point to Point protocol), **ISO 8208** (**X.25** DTE-DTE), **X.25 DCE**, **T.90NL** (with compatibility to **T.70NL**) and **T.30** (fax group 3).

Even if not all protocols can be fit completely within the OSI scheme, **COMMON-ISDN-API** will always support three layers. Each layer can be configured by applications. In case of illegal or meaningless combinations of protocol stack combinations (e.g. bit transparency 56 kBit and X.25 DCE) **COMMON-ISDN-API** will report this error.

The following chapter first presents the basic mechanism used for **COMMON-ISDN-API**. It is based on message queues provided for the exchange of commands and data. The operations on these message queues are described, the structure of exchanged messages is indicated. Afterwards the description of other functions for identification and the mechanism for manufacturer specific expansions will be provided.

3 MESSAGE OVERVIEW

The term *message* is a fundamental one to define **COMMON-ISDN-API**. An asynchronous mechanism, used to exchange information only defined by **COMMON-ISDN-API** (*messages*), achieves operating system independence..

3.1 General Message Protocol

Communication between application and **COMMON-ISDN-API** always uses the following general protocol:

A message is always followed by a corresponding response. Messages from an application going to **COMMON-ISDN-API** are called **REQUESTs**, the appropriate answer from **COMMON-ISDN-API** is called **CONFIRMATION**. On the other side messages originating from **COMMON-ISDN-API** are called **INDICATIONs**, the corresponding reactions of an application are called **RESPONSEs**. This also is reflected in the naming convention of messages: every message name ends with the appropriate suffix (_REQ, _CONF, _IND, _RESP).

Each message contains a message number. **COMMON-ISDN-API** will always return the number used in the REQUEST message in the corresponding CONFIRMATION. Applications may choose unique message numbers to identify message correlations before interpreting incoming messages. INDICATIONS from **COMMON-ISDN-API** will be numbered so that an application is guaranteed to get different message numbers for every incoming INDICATION.

An application is not allowed to send RESPONSE messages without receiving an INDICATION. **COMMON-ISDN-API** will ignore these illegal messages.

3.2 Type Definitions

Parameters are associated with every message exchanged. To describe the message and its parameters, only few basic types are used:

- byte — coded as one octet
- word — coded as two contiguous octets, least significant first
- dword — coded as two contiguous words, least significant first
- struct — coded as an array of octets, the first octet containing the length of following data. If the first octet has the value **255** (0xFF), it indicates an escape character for interpreting the following word as containing the length of the data. An empty struct will be coded as one single octet with value 0.

Every message will be described in terms of these basic types.

3.3 Message Structure

All messages exchanged between application and **COMMON-ISDN-API** consist of a fixed-length header and a parameter area of variable length, parameter followed by parameter. No padding occurs in the message or parameter area.

Message header	Parameter 1	Parameter 2	.....	Parameter n

Figure 2: Message Layout

In order to facilitate future extensions of this standard, messages containing additional parameters shall be treated as valid messages. **COMMON-ISDN-API** implementations and applications shall ignore all additional parameters.

The message header has the following layout:

Total length	ApplID	Command	Sub-command	Message number

Figure 3: Message Header Layout

Explanation of message header:

Message	**Type**	**Contents**
Total length	word	Total length of the message including the complete message header.
ApplID	word	Identification of the application. The application number is assigned to the application by **COMMON-ISDN-API** in the CAPI_REGISTER operation
Command	byte	Command
Subcommand	byte	Command extension
Message number	word	Message number as described above

3.4 Manufacturer Specific Expansion

Manufacturer specific expansions of **COMMON-ISDN-API** will be possible without altering the basic structure. They are identified by an appropriate command/subcommand field in the message.

3.5 Table of Messages

Messages are logically grouped into three kinds:

- messages concerning the signalling protocol of the ISDN (D-channel)
- messages concerning logical connections (B- or D-channel)
- administrative and other messages

The following table gives an overview of the defined messages and their functionality. The complete description of each message will be given in chapter 5.

Messages concerning signalling protocol:

Message	Description
CONNECT_REQ	initiates an outgoing physical connection
CONNECT_CONF	local confirmation of request
CONNECT_IND	indicates an incoming physical connection
CONNECT_RESP	response to indication
CONNECT_ACTIVE_IND	indicates the activation of a physical connection
CONNECT_ACTIVE_RESP	response to indication
DISCONNECT_REQ	initiates clearing of a physical connection
DISCONNECT_CONF	local confirmation of request
DISCONNECT_IND	indicates the clearing of a physical connection
DISCONNECT_RESP	response to indication
ALERT_REQ	initiates sending of ALERT, i.e. compatibility to call
ALERT_CONF	local confirmation of request
INFO_REQ	initiates sending of signalling information
INFO_CONF	local confirmation of request
INFO_IND	indicates selected signalling information
INFO_RESP	response to indication

Table 1: Messages concerning signalling protocol

Messages concerning logical connections:

Message	Description
CONNECT_B3_REQ	initiates an outgoing logical connection
CONNECT_B3_CONF	local confirmation of request
CONNECT_B3_IND	indicates an incoming logical connection
CONNECT_B3_RESP	response to indication
CONNECT_B3_ACTIVE_IND	indicates the activation of a logical connection
CONNECT_B3_ACTIVE_RESP	response to indication
CONNECT_B3_T90_ACTIVE_IND	indicates switching from T.70NL to T.90NL
CONNECT_B3_T90_ACTIVE_RESP	response to indication
DISCONNECT_B3_REQ	initiates clearing of a logical connection

DISCONNECT_B3_CONF	local confirmation of request
DISCONNECT_B3_IND	indicates the clearing of a logical connection
DISCONNECT_B3_RESP	response to indication
DATA_B3_REQ	initiates sending of data on a logical connection
DATA_B3_CONF	local confirmation of request
DATA_B3_IND	indicates incoming data on a logical connection
DATA_B3_RESP	response to indication
RESET_B3_REQ	initiates the reset of a logical connection
RESET_B3_CONF	local confirmation of request
RESET_B3_IND	indicates the reset of a logical connection
RESET_B3_RESP	response to indication

Table 2: Messages concerning logical connections

Administrative and other messages:

Message	Description
LISTEN_REQ	activates call indications
LISTEN_CONF	local confirmation of request
FACILITY_REQ	requests additional facilities (e.g. ext. equipment)
FACILITY_CONF	local confirmation of request
FACILITY_IND	indicates additional facilities (e.g. ext. equipment)
FACILITY_RESP	response to indication
SELECT_B_PROTOCOL_REQ	selects current protocol stack of a logical connection
SELECT_B_PROTOCOL_CONF	local confirmation of request
MANUFACTURER_REQ	manufacturer specific operation
MANUFACTURER_CONF	manufacturer specific operation
MANUFACTURER_IND	manufacturer specific operation
MANUFACTURER_RESP	manufacturer specific operation

Table 3: Administrative and other messages

4 EXCHANGE MECHANISM

4.1 Message Queues

Communication between an application program and **COMMON-ISDN-API** takes place via message queues. As shown in figure 4, there is exactly one message queue for **COMMON-ISDN-API** and for each registered application program. Messages are exchanged between the applications programs and **COMMON-ISDN-API** via these message queues. For data transfer the messages are used for control purposes only, and the data itself is transferred via a data area common to the application and **COMMON-ISDN-API**. The queues are organised first in - first out, so **COMMON-ISDN-API** will process messages in the order of their arrival.

An application issues commands to an ISDN driver or controller by placing an appropriate message in the **COMMON-ISDN-API** message queue. In the reverse direction, a message from an ISDN driver or controller is transferred to the message queue of the addressed application.

This method, used in higher-level protocols and modern operating systems, allows flexible access by several applications to different ISDN drivers and controllers. It also provides a powerful mechanism for processing events that arrive asynchronously, which is a paramount requirement for high speed data transfer.

The message queue structure is not specified. It is manufacturer-dependent and is transparent to the application program. The necessary access operations are defined by **COMMON-ISDN-API**.

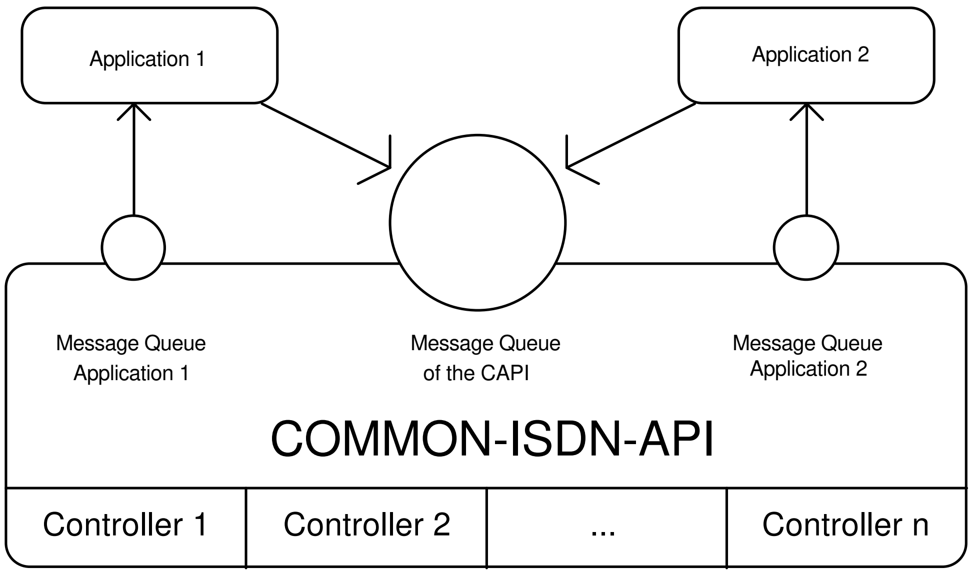


Figure 4: Message queues in **COMMON-ISDN-API**

4.2 Operations on Message Queues

The message queues described represent the link between an application and **COMMON-ISDN-API** with its connected ISDN drivers and controllers. Only four operations are required to use the message queues. The operations on the message queues are not restricted to a particular system specification. Their respective characteristics and implementation are operating system specific. At the same time, these operations form the complete interface which has to be matched to the particular operating system. The four operations are described below.

4.2.1 Registering an Application

Before an application can issue commands to **COMMON-ISDN-API** it must be registered at **COMMON-ISDN-API**. The CAPI_REGISTER function is used to do this. **COMMON-ISDN-API** uses this function to assign a unique application number (ApplID) to the application. The message queue for the application is set up at the same time.

4.2.2 Messages from Application to COMMON-ISDN-API

All messages from an application to **COMMON-ISDN-API** are put in the message queue of **COMMON-ISDN-API**. The operation CAPI_PUT_MESSAGE is provided for this purpose. When this operation is used, the application transfers the message. If **COMMON-ISDN-API** message queue cannot accept any more messages, the operation CAPI_PUT_MESSAGE returns an error.

4.2.3 Messages from COMMON-ISDN-API to Application

COMMON-ISDN-API manages a message queue for each application; **COMMON-ISDN-API** puts all messages to the application in this queue. The operation CAPI_GET_MESSAGE is provided for reading new messages from this queue. When this operation is used, it returns the received message. If the application message queue is empty, the operation CAPI_GET_MESSAGE returns an error. If an application does not retrieve these messages and message queue size was configured too small, this queue may overflow. In this case one or more messages from **COMMON-ISDN-API** are lost. The application is informed of this error on the next CAPI_GET_MESSAGE operation.

4.2.4 Releasing an Application

If a registered application wants to terminate **COMMON-ISDN-API** usage, the connection to **COMMON-ISDN-API** must be released. This can be done with the CAPI_RELEASE operation. Releasing the application releases the previously used message queue. An application has to disconnect all existing connections before issuing an CAPI_RELEASE, otherwise the behaviour of **COMMON-ISDN-API** is undefined. This is valid only for non-external equipment, external devices controlled by **COMMON-ISDN-API** (e.g. phone) may allow releasing from **COMMON-ISDN-API** without terminating existing calls.

4.2.5 Other Operations

Additional Operations are available to get information about manufacturer, software releases, configuration and serial numbers. Depending on the operating system there exists also a possibility to register a call-back function which will be activated if a new message is put in the application's message queue.

4.2.6 Manufacturer Specific Expansion

There also exists a manufacturer specific operation, e.g. to configure ISDN controller.

4.3 Table of Operations

Operation	Description
CAPI_REGISTER	Register an application
CAPI_RELEASE	Release an application
CAPI_PUT_MESSAGE	Transfer message to CAPI
CAPI_GET_MESSAGE	Get message from CAPI
CAPI_SET_SIGNAL	Register call-back function
CAPI_GET_MANUFACTURER	Get manufacturer identification
CAPI_GET_VERSION	Get CAPI version numbers
CAPI_GET_SERIAL_NUMBER	Get serial number
CAPI_GET_PROFILE	Get capabilities of CAPI implementation
CAPI_MANUFACTURER	Manufacturer specific function

Table 4: Operations defined in COMMON-ISDN-API

5 MESSAGE DESCRIPTIONS

The following section defines all **COMMON-ISDN-API** messages with their respective parameters. Parameters are explained more detailed in chapter 6.

Messages are sorted alphabetically irrespective of the extension, which defines the originator and direction of the message. The following order always will be used for basic names: REQUEST, CONFIRMATION, INDICATION, RESPONSE.

5.1 ALERT_REQ

Description

This message should be used by applications to indicate compatibility to an incoming call. It will send an ALERT to the network to prevent the call from expiring (No user responding). If an application is able to accept the call immediately it is not necessary to use this message; the application can issue immediately a CONNECT_RESP to **COMMON-ISDN-API**.

ALERT_REQ	Command	**0x01**
	Subcommand	**0x80**

Parameter	**Type**	**Comment**
PLCI	dword	Physical Link Connection Identifier
Additional info	struct	Additional info elements

Note

The parameter *Additional info* will be a coded as an empty structure if no additional information (e.g. user data) has to be transmitted.

5.2 ALERT_CONF

Description

This message confirms the reception of an ALERT_REQ.

ALERT_CONF	Command	**0x01**
	Subcommand	**0x81**

Parameter	Type	Comment
PLCI	dword	Physical Link Connection Identifier
Info	word	**0:** alert initiated **0x0003:** alert already sent by another application **0x2001:** message not supported in current state **0x2002:** illegal PLCI **0x2007:** illegal message parameter coding

Note

Info 0x0003 will be returned if another application already initiated the sending of an ALERT message to the network. In this case the parameter *Additional info* of the corresponding REQUEST has been ignored.

See also

Description of *broadcast mechanism* in LISTEN_REQ

5.3 CONNECT_REQ

Description

This message initiates the set-up of a physical connection. An application only has to offer the relevant parts of the parameters, i.e. *Controller*, *CIP Value, B protocol* and normally c*alled party number*. Every other structure can be empty (length of 0). In this case the default values as described in chapter 6 will be used.

CONNECT_REQ	Command	0x02
	Subcommand	0x80

Parameter	Type	Comment
Controller	dword	
CIP Value	word	Compatibility Information Profile
Called party number	struct	Called party number
Calling party number	struct	Calling party number
Called party subaddress	struct	Called party subaddress
Calling party subaddress	struct	Calling party subaddress
B protocol	struct	B protocol to be used
BC	struct	Bearer Capability
LLC	struct	Low Layer Compatibility
HLC	struct	High Layer Compatibility
Additional Info	struct	Additional information elements

Note

If an application offers *BC*, *LLC* and/or *HLC*, the parameter will be used without checking the resulting combination.

The absence (i.e. coding as an empty structure) of *B protocol* will result in the default protocol behaviour: ISO 7776 (X.75) and window size 7. This is a recommended selection to get overall connectivity with the benefits of HDLC error recovery. Note that ISO 7776 deals with a default maximum data length of 128 octets, whereas **COMMON-ISDN-API** is able to handle up to at least 2048 octets, depending on **CAPI_REGISTER** values of an application.

5.4 CONNECT_CONF

Description

This message confirms the initiation of a call set-up. This connection is assigned a *PLCI* which serves as an identifier in further processing. Errors are returned in the parameter *info*.

CONNECT_CONF	Command	**0x02**
	Subcommand	**0x81**

Parameter	Type	Comment
PLCI	dword	Physical Link Connection Identifier
Info	word	**0:** connect initiated **0x2002:** illegal controller **0x2003:** out of PLCI **0x2007:** illegal message parameter coding **0x3001:** B1 protocol not supported **0x3002:** B2 protocol not supported **0x3003:** B3 protocol not supported **0x3004:** B1 protocol parameter not supported **0x3005:** B2 protocol parameter not supported **0x3006:** B3 protocol parameter not supported **0x3007:** B protocol combination not supported **0x300A:** CIP Value unknown

Note

The connection is in the set-up phase at this point in time. Subsequent successful switching is indicated by the message **CONNECT_ACTIVE_IND**.

If an application has to identify the corresponding REQUEST to this message, it can use the message number mechanism described in chapter 3.

5.5 CONNECT_IND

Description

This message indicates an incoming call for a physical connection. For the incoming call a PLCI is assigned which is used to identify this connection in subsequent messages.

CONNECT_IND	Command	0x02
	Subcommand	0x82

Parameter	Type	Comment
PLCI	dword	Physical Link Connection Identifier
CIP Value	word	Compatibility Information Profile
Called party number	struct	Called party number
Calling party number	struct	Calling party number
Called party subaddress	struct	Called party subaddress
Calling party subaddress	struct	Calling party subaddress
BC	struct	Bearer compatibility
LLC	struct	Low Layer Compatibility
HLC	struct	High Layer Compatibility
Additional Info	struct	Additional information elements

Note

To activate the signalling of incoming calls, the message **LISTEN_REQ** must be sent to the controller.

Every information available from the network at this point will be signalled to the application. Empty structs will show the absence of this information.

5.6 CONNECT_RESP

Description

This message is used to accept or reject an incoming call on behalf of the application. The incoming call is identified via parameter *PLCI*. The parameter *reject* is used to accept, reject or ignore the call. In case of ignoring the call, other ISDN equipment connected on the same bus (basic access) will have the chance to accept this call, whereas the rejection of this incoming call will try to terminate the call on the entire bus. For primary access, these parameter values of parameter *Reject* will behave identically.

CONNECT_RESP	Command	**0x02**
	Subcommand	**0x83**

Parameter	**Type**	**Comment**
PLCI	dword	Physical Link Connection Identifier
Reject	word	**0:** accept call **1:** ignore call **2:** reject call, normal call clearing **3:** reject call, user busy **4:** reject call, requestet circuit/channel not available **5:** reject call, facility rejected **6:** reject call, channel unacceptable **7:** reject call, incompatible destination **8:** reject call, destination out of order
B protocol	struct	B protocol to be used
Connected number	struct	Connected number
Connected subaddress	struct	Connected subaddress
LLC	struct	Low Layer Compatibility
Additional Info	struct	Additional information elements

Note

The parameter *LLC* can optionally be used for LLC negotiation if supported by the network.

Any unknown *reject* value will be mapped to *normal call clearing*.

Any *reject* value other than *accept call* will cause a DISCONNECT_IND to be sent to the application.

The absence (i.e. coding as an empty structure) of *B protocol* will result in the default protocol behaviour: ISO 7776 (X.75) and window size 7. This is a recommended selection to get overall connectivity with the benefits of HDLC error recovery. Note that

ISO 7776 deals with a default maximum data length of 128 octets, whereas **COMMON-ISDN-API** is able to handle up to at least 2048 octets, depending on **CAPI_REGISTER** values of an application.

5.7 CONNECT_ACTIVE_IND

Description

This message indicates the physical connection of a B channel. The connection is identified by the parameter *PLCI*.

CONNECT_ACTIVE_IND	Command	0x03
	Subcommand	0x82

Parameter	Type	Comment
PLCI	dword	Physical Link Connection Identifier
Connected number	struct	Connected number
Connected subaddress	struct	Connected subaddress
LLC	struct	Low Layer Compatibility

Note

The parameter *connected number/subaddress* and *LLC* will be filled in completely if this information is provided by the network. The absence of network information will be indicated by empty structures.

5.8 CONNECT_ACTIVE_RESP

Description

With this message the application confirms the receipt of a **CONNECT_ACTIVE_IND**.

CONNECT_ACTIVE_RESP	Command	0x03
	Subcommand	0x83

Parameter	Type	Comment
PLCI	dword	Physical Link Connection Identifier

5.9 CONNECT_B3_ACTIVE_IND

Description

This message indicates the logical connection of a B channel. The connection is identified by the parameter *NCCI*. The parameter *NCPI* is used to transfer additional protocol dependent information.

CONNECT_B3_ACTIVE_IND	Command	**0x83**
	Subcommand	**0x82**

Parameter	**Type**	**Comment**
NCCI	dword	Network Control Connection Identifier
NCPI	struct	Network Control Protocol Information

Note

The meaning of the parameter *NCPI* depends on the protocol used.

After this message incoming data can be indicated to the application.

In case of protocol T.30 and outgoing calls, this message does not imply the successful training between both fax stations. This is to enable an application to send data to **COMMON-ISDN-API** without waiting for termination of training phase. If this training phase is not successful, corresponding indications will be given by **COMMON-ISDN-API** in the message DISCONNECT_B3_IND.

5.10 CONNECT_B3_ACTIVE_RESP

Description

With this message the application confirms the receipt of a **CONNECT_B3_ACTIVE_IND**.

CONNECT_B3_ACTIVE_RESP	Command	0x83
	Subcommand	0x83

Parameter	Type	Comment
NCCI	dword	Network Control Connection Identifier

5.11 CONNECT_B3_REQ

Description

This message initiates the set-up of a logical connection. The physical connection is identified by the parameter *PLCI*. Additional protocol dependent information can be transferred with the parameter *NCPI*.

CONNECT_B3_REQ	Command	0x82
	Subcommand	0x80

Parameter	Type	Comment
PLCI	dword	Physical Link Connection Identifier
NCPI	struct	Network Control Protocol Information

Note

The meaning of the parameter *NCPI* depends on the protocol used.

5.12 CONNECT_B3_CONF

Description

With this message the initiation of a logical connection set-up is confirmed. This connection is assigned a *NCCI*, which subsequently identifies this logical connection. Errors are supplied in the parameter *info*.

CONNECT_B3_CONF	Command	0x82
	Subcommand	0x81

Parameter	Type	Comment
NCCI	dword	Network Control Connection Identifier
Info	word	**0:** connect initiated **0x0001:** NCPI not supported by current protocol, NCPI ignored **0x2001:** message not supported in current state **0x2002:** illegal PLCI **0x2004:** out of NCCI **0x3008:** NCPI not supported

Note

The connection is in the set-up phase at this stage. The successful set-up will be indicated by the message **CONNECT_B3_ACTIVE_IND**.

If parameter *info* returns **0x0001**, the set-up of a logical connection is initiated, but parameter *NCPI* has been ignored. In that case the used layer 3 protocol does not support the usage of *NCPI* (e.g. the transparent mode of layer 3).

5.13 CONNECT_B3_IND

Description

This message indicates an incoming call for a logical connection. For this incoming call a *NCCI* is assigned, which subsequently identifies the call. Additional protocol dependent information will be transferred with parameter *NCPI* if available.

CONNECT_B3_IND	Command	0x82
	Subcommand	0x82

Parameter	Type	Comment
NCCI	dword	Network Control Connection Identifier
NCPI	struct	Network Control Protocol Information

Note

The meaning of the parameter *NCPI* depends on the protocol used.

The connection is in the set-up phase at this stage. The successful set-up will be indicated by the message **CONNECT_B3_ACTIVE_IND**.

5.14 CONNECT_B3_RESP

Description

With this message the application accepts or rejects an incoming logical call. The incoming call is identified via the parameter *NCCI*. The call can be accepted or rejected via the parameter *reject*. The parameter *NCPI* can be used to transfer additional protocol dependent information.

CONNECT_B3_RESP	Command	**0x82**
	Subcommand	**0x83**

Parameter	**Type**	**Comment**
NCCI	dword	Network Control Connection Identifier
Reject	word	**0:** accept call **2:** reject call, normal call clearing
NCPI	struct	Network Control Protocol Information

Note

The meaning of the parameter *NCPI* depends on the protocol used.

Any other value of *reject* will result in rejecting the call.

5.15 CONNECT_B3_T90_ACTIVE_IND

Description

This message indicates the switching from T.70 to T.90 within a logical connection of a B channel. The connection is identified by the parameter *NCCI*. The parameter *NCPI* is used to transfer additional T.90 information.

CONNECT_B3_T90_ACTIVE_IND	Command	**0x88**
	Subcommand	**0x82**

Parameter	**Type**	**Comment**
NCCI	dword	Network Control Connection Identifier
NCPI	struct	Network Control Protocol Information

Note

This message will only be generated if the selected protocol is T.90NL with compatibility to T.70NL according to T.90 Appendix II. In this case the initially used protocol is T.70. This message indicates the negotiation and switching to T.90.

5.16 CONNECT_B3_T90_ACTIVE_RESP

Description

With this message the application confirms the receipt of a **CONNECT_B3_T90_ACTIVE_IND**.

CONNECT_B3_T90_ACTIVE_RESP	Command	0x88
	Subcommand	0x83

Parameter	Type	Comment
NCCI	dword	Network Control Connection Identifier

5.17 DATA_B3_REQ

Description

This message sends data within the logical connection identified by the *NCCI*. Data to be sent is referenced via the parameter *data/data length*. The data is not part of the message, a 32-bit pointer is used to transfer the address of the data area. The application issues a unique identifier for this data in the parameter *data handle*. On subsequent confirmation by a **DATA_B3_CONF** this handle is used. It is possible to set additional information, such as more data, delivery confirmation etc. via parameter *flags*. The flags are not supported by all protocols.

DATA_B3_REQ	Command	**0x86**
	Subcommand	**0x80**

Parameter	**Type**	**Comment**
NCCI	dword	Network Control Connection Identifier
Data	dword	Pointer to the data to be sent
Data length	word	Size of data area to be sent
Data handle	word	Referenced in **DATA_B3_CONF**
Flags	word	**[0]:** qualifier bit **[1]:** more data bit **[2]:** delivery confirmation bit **[3]:** expedited data **[4] to [15]:** reserved

Note

The data transfer does not support assembly or re-assembly of data.

An application must not change or free the data area until the corresponding **DATA_B3_CONF** is received.

Flags are protocol dependent. If an application set reserved bits in parameter *Flags*, **COMMON-ISDN-API** will reject the **DATA_B3_REQ**. This is to allow future expansion of this parameter. If an application set bits in parameter *Flags*, which are not supported by the current protocol, **COMMON-ISDN-API** will accept the **DATA_B3_REQ** but will return this information in the corresponding **DATA_B3_CONF**.

5.18 DATA_B3_CONF

Description

This message confirms the acceptance of a data package to be sent. The logical connection is identified by the parameter *NCCI*. The parameter *data handle* supplies the identifier used by the application in the associated **DATA_B3_REQ** as reference to the transferred data area. After receiving this message, the application can reuse the referenced data area.

DATA_B3_CONF	Command	0x86
	Subcommand	0x81

Parameter	Type	Comment
NCCI	dword	Network Control Connection Identifier
Data handle	word	Identifies the data area of corresponding **DATA_B3_REQ**
Info	word	**0:** data transmission initiated **0x0002:** flags not supported by current protocol, flags ignored **0x2001:** message not supported in current state **0x2002:** illegal NCCI **0x2007:** illegal message parameter coding **0x300A:** flags not supported (reserved bits) **0x300C:** data length not supported by current protocol

Note

Every **DATA_B3_REQ** will result in a corresponding **DATA_B3_CONF** exept in the following case: after transmitting the message **DISCONNECT_B3_IND** to an application, **COMMON-ISDN-API** is not allowed to send any other message concerning this logical connection identified by the parameter NCCI. So in this case the application has to make sure that resources or buffer management will be reset correctly.

If an application sets the delivery confirmation bit in the corresponding DATA_B3_REQ and the selected protocol supports this mechanism it is guaranteed that this confirmation will be given to the application after the delivery of the sent packet is confirmed by the used protocol.

Seven unconfirmed DATA_B3_REQ messages will be supported.

5.19 DATA_B3_IND

Description

This message indicates incoming data within a logical connection. The logical connection is identified via the *NCCI*. The length of the incoming data area is indicated via the parameter *data length*. The incoming data area can be referenced by the parameter *data*. The data is not part of the message, a 32-bit pointer is used to transfer the address of the data area. **COMMON-ISDN-API** issues a handle to this data area via the parameter *data handle*. On subsequent confirmation by a **DATA_B3_RESP**, this handle must also be supplied by the application. Additional information - such as more data, delivery confirmation etc. - is supplied by parameter *flags*, if available.

DATA_B3_IND	Command	0x86
	Subcommand	0x82

Parameter	Type	Comment
NCCI	dword	Network Control Connection Identifier
Data	dword	Pointer to data received
Data length	word	Size of data area received
Data handle	word	handle to data area, referenced in **DATA_B3_RESP**
Flags	word	**[0]:** qualifier bit **[1]:** more-data bit **[2]:** delivery confirmation bit **[3]:** expedited data **[4 to 14]:** reserved **[15]:** framing error bit, data may be invalid (only with corresponding B2 protocol)

Note

The data transfer does not support re-assembly functions.

The data area which contains the data remains allocated until the corresponding **DATA_B3_RESP** is received. However, expedited data is only valid until the next **CAPI_GET_MESSAGE** is performed by the application.

In case of receiving **DATA_B3_IND** messages with reserved bits switched on in the flags parameter an application must ignore the data area but process the message, i.e. send a **DATA_B3_RESP** to **COMMON-ISDN-API**. This is to allow future expansion of the *flags* parameter.

5.20 DATA_B3_RESP

Description

With this message the application confirms acceptance of an incoming data package. The logical connection is identified by the parameter *NCCI*. The parameter *data handle* identifies the data handle used by **COMMON-ISDN-API** in the corresponding **DATA_B3_IND** as the reference to the transferred data area.

DATA_B3_RESP	Command	**0x86**
	Subcommand	**0x83**

Parameter	**Type**	**Comment**
NCCI	dword	Network Control Connection Identifier
Data handle	word	Data area reference in corresponding **DATA_B3_IND**

Note

This message frees the data buffer referenced by *Data handle* for reuse by **COMMON-ISDN-API**.

Data throughput depends on an application's rapid response to DATA_B3_IND messages. Failure to do so will trigger flow control on the line (for protocols supporting flow control such as ISO 7776(X.75) or ISO8208(X.25)) and may cause loss of incoming data for protocols without flow control mechanism.

5.21 DISCONNECT_B3_REQ

Description

This message initiates the clearing of a logical connection identified via the parameter *NCCI*. The parameter *NCPI* can be used to transfer additional protocol dependent information.

DISCONNECT_B3_REQ	Command	0x84
	Subcommand	0x80

Parameter	Type	Comment
NCCI	dword	Network Control Connection Identifier
NCPI	struct	Network Control Protocol Information

Note

The meaning of the parameter *NCPI* depends on the protocol used.

In case of fax group 3 (B protocol T.30) and speech (B1 protocol bit transparent, B2/B3 protocol transparent) data already given to transmission via **DATA_B3_REQ** will be sent before disconnecting the logical connection.

5.22 DISCONNECT_B3_CONF

Description

With this message the initiation of clearing a logical connection is confirmed. Any errors are coded in the parameter *info*.

DISCONNECT_B3_CONF	Command	0x84
	Subcommand	0x81

Parameter	Type	Comment
NCCI	dword	Network Control Connection Identifier
Info	word	**0:** disconnect initiated **0x0001:** NCPI not supported by current protocol, NCPI ignored **0x2001:** message not supported in current state **0x2002:** illegal NCCI **0x2007:** illegal message parameter coding **0x3008:** NCPI not supported

5.23 DISCONNECT_B3_IND

Description

This message indicates the clearing of a logical connection identified via the parameter *NCCI*. The parameter *Reason_B3* indicates if this clearing is caused by wrong protocol behaviour. The parameter *NCPI* is used to indicate additional protocol dependent information if available.

DISCONNECT_B3_IND	Command	**0x84**
	Subcommand	**0x82**

Parameter	**Type**	**Comment**
NCCI	dword	Network Control Connection Identifier
Reason_B3	word	**0:** clearing according to protocol **0x3301:** protocol error layer 1 **0x3302:** protocol error layer 2 **0x3303:** protocol error layer 3 protocol dependent values are described in chapter 6
NCPI	struct	Network Control Protocol Information

Note

The meaning of the *NCPI* parameter depends on the protocol used.

After this message no other message concerning this *NCCI* will be sent to the application. The application has to answer this message with **DISCONNECT_B3_RESP** to free the resources allocated to the *NCCI*.

5.24 DISCONNECT_B3_RESP

Description

With this message the application confirms the clearing of a logical connection.

DISCONNECT_B3_RESP	Command	0x84
	Subcommand	0x83

Parameter	Type	Comment
NCCI	dword	Network Control Connection Identifier

Note

With this message resources allocated to the *NCCI* are released.

If an application fails to send this message after receiving **DISCONNECT_B3_IND**, **COMMON-ISDN-API** will eventually reject subsequent **CONNECT_B3_REQ** with the info value **out of NCCI** (0x2004).

5.25 DISCONNECT_REQ

Description

This message initiates the clearing of a physical connection, identified by the parameter *PLCI*.

DISCONNECT_REQ	Command	0x04
	Subcommand	0x80

Parameter	Type	Comment
PLCI	dword	Physical Link Connection Identifier
Additional Info	struct	Additional information elements

Note

Existing logical connections will be cleared by **COMMON-ISDN-API** using the message DISCONNECT_B3_IND containing the cause **protocol error layer 1** (0x3301) before clearing the physical connection.

5.26 DISCONNECT_CONF

Description

This message confirms the initiation of clearing a physical connection. Any errors are coded in the parameter *info*.

DISCONNECT_CONF	Command	**0x04**
	Subcommand	**0x81**

Parameter	Type	Comment
PLCI	dword	Physical Link Connection Identifier
Info	word	**0:** disconnect initiated **0x2001:** message not supported in current state **0x2002:** illegal PLCI **0x2007:** illegal message parameter coding

5.27 DISCONNECT_IND

Description

This message indicates the clearing of the physical channel identified via the parameter *PLCI*. The parameter *reason* indicates the network delivered cause or if this clearing is caused by wrong protocol behaviour

DISCONNECT_IND	Command	**0x04**
	Subcommand	**0x82**

Parameter	**Type**	**Comment**
PLCI	dword	Physical Link Connection Identifier
Reason	word	**0:** no cause available **0x3301:** protocol error layer 1 **0x3302:** protocol error layer 2 **0x3303:** protocol error layer 3 **0x3304:** another application got that call **0x34xx**: disconnect cause from the network according to Q.931/ETS 300 102-1. In the field 'xx' the cause value received within a cause information element (octet 4) from the network is indicated.

Note

After this message no other message concerning this *PLCI* will be sent to the application. The application has to answer this message with **DISCONNECT_RESP** to free the resources allocated to the *PLCI*.

5.28 DISCONNECT_RESP

Description

With this message the application confirms the clearing of the physical channel.

DISCONNECT_RESP	Command	**0x04**
	Subcommand	**0x83**

Parameter	Type	Comment
PLCI	dword	Physical Link Connection Identifier

Note

With this message the *PLCI* is released.

If an application fails to send this message after receiving **DISCONNECT_IND** resources bound to this *PLCI* will not be freed. This may lead to **COMMON-ISDN-API** resource problems (indicated by info value **out of PLCI),** affecting other applications too.

5.29 FACILITY_REQ

Description

This message is used to handle optional facilities on the *controller* or facilities related on connections identified by *PLCI* or *NCCI*. The struct *facility request parameters* is defined for each facility. At the moment facilities Handset Support and DTMF are defined. Handset Support is used to support external ISDN equipment, DTMF (Dual Tone Multi Frequency) is used in the PSTN (Public Switched Telephone Network) to select and control several provided services (e.g. automatic answering service).

Handset Support as well as DTMF support are optional **COMMON-ISDN-API** features. In case **COMMON-ISDN-API** does not support these facilities, an appropriate information value is returned in the **FACILITY_CONF**.

DTMF can not be used with all B protocols. Normally it is used with 64 kBit/sec bit speech and T.30 audio.

FACILITY_REQ	Command	**0x80**
	Subcommand	**0x80**

Parameter	**Type**	**Comment**
Controller/PLCI/NCCI	dword	Depending on the facility selector
Facility selector	word	**0:** Handset Support **1:** DTMF **2 to n:** reserved
Facility request parameter	struct	Facility depending parameters

5.30 FACILITY_CONF

Description

This message confirms the acceptance of the **FACILITY_REQ**. The event is identified by *Controller/PLCI/NCCI*, depending on the facility. The struct *facility confirmation parameters* is defined for every facility. Any error is coded in the parameter *info*.

FACILITY_CONF	Command	**0x80**
	Subcommand	**0x81**

Parameter	Type	Comment
Controller/PLCI/NCCI	dword	Depending on the facility selector
Info	word	**0:** request accepted **0x2001:** message not supported in current state **0x2002:** incorrect Controller/PLCI/NCCI **0x2007:** illegal message parameter coding **0x300B:** facility not supported
Facility selector	word	**0:** Handset Support **1:** DTMF **2 to n:** reserved
Facility confirmation parameter	struct	Facility-depending parameters

5.31 FACILITY_IND

Description

This message is used to indicate a facility dependent event originating from a controller or connections identified via *controller/PLCI/NCCI*, depending on the facility. The struct *facility indication parameter* is defined for every facility.

FACILITY_IND	Command	0x80
	Subcommand	0x82

Parameter	Type	Comment
Controller/PLCI/NCCI	dword	Depending on the facility selector
Facility selector	word	**0:** Handset Support **1:** DTMF **2 to n:** reserved
Facility indication parameter	struct	Facility-depending parameters

Note

In case of facility selector **0** (Handset Support) this message may allocate a new PLCI (in case of off-hooking the handset) which has to be released afterwards by means of **DISCONNECT_IND / DISCONNECT_RESP**.

5.32 FACILITY_RESP

Description

With this message the application confirms receipt of a facility indication message. The struct *facility response parameters* is defined for each facility.

FACILITY_RESP	Command	0x80
	Subcommand	0x83

Parameter	Type	Comment
Controller/PLCI/NCCI	dword	Depending on the facility selector
Facility selector	word	**0:** Handset Support **1:** DTMF **2 to n:** reserved
Facility response parameters	struct	Facility-depending parameters

5.33 INFO_REQ

Description

This message permits sending of protocol information for a the physical connection, e.g. overlap sending.

INFO_REQ	Command	**0x08**
	Subcommand	**0x80**

Parameter	Type	Comment
Controller/PLCI	dword	See note
Called party number	struct	Called party number
Additional Info	struct	Additional information elements

Note

The first parameter identifies a physical connection (if a PLCI is given) or the addressed controller (if the PLCI field of parameter *Controller/PLCI* is zero). Depending on the parameter different messages will be sent to the network.

5.34 INFO_CONF

Description

This message confirms acceptance of **INFO_REQ**. If in the corresponding **INFO_REQ** a controller is given as an addressing parameter, this connection is assigned a *PLCI* which serves as an identifier in further processing. Any error is coded in the parameter *info*.

INFO_CONF	Command	0x08
	Subcommand	0x81

Parameter	Type	Comment
PLCI	dword	Physical Link Connection Identifier
Info	word	**0:** transmission of information initiated **0x2001:** message not supported in current state **0x2002:** illegal Controller/PLCI **0x2003:** out of PLCI **0x2007:** illegal message parameter coding

5.35 INFO_IND

Description

This message indicates an event for a physical connection as expressed by an information element (*info element*) whose coding is described by the parameter *info number*. The connection is identified via the parameter *controller/PLCI*.

INFO_IND	Command	0x08
	Subcommand	0x82

Parameter	Type	Comment
Controller/PLCI	dword	Physical Link Connection Identifier
Info number	word	Information element identifier
Info element	struct	Information element dependent structure

Note

An individual **INFO_IND** is displayed for each information element. To enable indication of events, the info mask parameter of the message **LISTEN_REQ** has to be used.

If the *PLCI* field in the address parameter is 0, the network has sent information not associated with a physical connection.

In case of getting information from the network which will lead to other **COMMON-ISDN-API** messages (e.g. receiving a RELEASE from the network which includes charging information) it is guaranteed that an application will get the **INFO_IND** first, followed by the corresponding **COMMON-ISDN-API** message.

5.36 INFO_RESP

Description

With this message the application confirms the receipt of an **INFO_IND**.

INFO_RESP	Command	**0x08**
	Subcommand	**0x83**

Parameter	Type	Comment
Controller/PLCI	dword	As in INFO_IND

5.37 LISTEN_REQ

Description

This message is used to activate signalling of incoming events from **COMMON-ISDN-API** to the application. *Info mask* is used to define which signalling protocol events are indicated to the application. These events are normally associated with physical connections. *CIP mask* defines selection criteria based upon *Bearer Capability* and *High Layer Compatibility*, thus indicating which incoming calls are signalled to an application.

More than one application may listen to the same *CIP Values*. *Every* application listening to a matching value will be informed about incoming calls. In case more than one application wants to accept the call, the first **CONNECT_RESP** received by **COMMON-ISDN-API** as a reaction to the **CONNECT_IND** will be accepted. Every other application will get the message **DISCONNECT_IND** with a Parameter *reason* which indicates this situation.

This scenario is similar to the situation where more than one set of compatible ISDN equipment on an ISDN line attempts to accept an incoming call.

LISTEN_REQ	Command	0x05
	Subcommand	0x80

Parameter	Type	Comment
Controller	dword	
Info mask	dword	Bit field, coding as follows: **[0]:** cause **[1]:** date/Time **[2]:** display **[3]:** user-user information **[4]:** call progression **[5]:** facility **[6]:** charging **[7 to 31]:** reserved
CIP Mask	dword	explained below
CIP Mask 2	dword	reserved for additional services
Calling party number	struct	Calling party number
Calling party subaddress	struct	Calling party subaddress

Explanation of *CIP Mask*:

Parameter	Type	Comment
CIP Mask	dword	Bit field, coding as follows: **[0]:** any match **[1]:** speech **[2]:** unrestricted digital information **[3]:** restricted digital information **[4]:** 3.1 kHz audio **[5]:** 7.0 kHz audio **[6]:** video **[7]:** packet mode **[8]:** 56 kBit/s rate adaptation **[9]:** unrestricted digital information with tones/announcements **[10..15]:** reserved **[16]:** telephony **[17]:** fax group 2/3 **[18]:** fax group 4 class 1 **[19]:** Teletex service (basic and mixed), fax group 4 class 2 **[20]:** Teletex service (basic and processable) **[21]:** Teletex service (basic) **[22]:** Videotex **[23]:** Telex **[24]:** message handling systems according X.400 **[25]:** OSI applications according X.200 **[26]:** 7 kHz Telephony **[27]:** Video Telephony F.721, first connection **[28]:** Video Telephony F.721, second connection **[29 to 31]:** reserved

Note

Clearing all bits in the *CIP mask* disables the signalling of incoming calls to the application.

Calling party number/subaddress are only used for external ISDN equipment (handsets), which might need the *own* (local) address to handle *outgoing* calls.

5.38 LISTEN_CONF

Description

This message confirms the acceptance of the LISTEN_REQ. Any errors are coded in the parameter *info*.

LISTEN_CONF	Command	**0x05**
	Subcommand	**0x81**

Parameter	Type	Comment
Controller	dword	
Info	word	**0:** listen is active **0x2002:** illegal controller **0x2005:** out of LISTEN-Resources **0x2007:** illegal message parameter coding

5.39 MANUFACTURER_REQ

Description

This message is used to transfer manufacturer specific information.

MANUFACTURER_REQ	Command	**0xFF**
	Subcommand	**0x80**

Parameter	**Type**	**Comment**
Controller	dword	
Manu ID	dword	Manufacturer specific ID (should be unique)
Manufacturer specific		Manufacturer specific data

Note

This message should not be used, for it is a non compatible message. Applications which use this message will only work with one manufacturer of ISDN equipment.

A manufacturer will choose *one* manufacturer specific ID for all of that **COMMON-ISDN-API** implementations. This manufacturer specific ID shall be unique. A shortcut or nickname based on the manufacturer's initials might be a good choice.

The behaviour of **COMMON-ISDN-API** is **not defined** after receiving any MANU-FACTURER_REQ.

5.40 MANUFACTURER_CONF

Description

This message confirms the reception of a **MANUFACTURER_REQ**.

MANUFACTURER_CONF	Command	0xFF
	Subcommand	0x81

Parameter	Type	Comment
Controller	dword	
Manu ID	dword	Manufacturer specific ID (should be unique)
Manufacturer specific		Manufacturer specific data

5.41 MANUFACTURER_IND

Description

This message is used to indicate manufacturer specific information to an application. **COMMON-ISDN-API** must not generate this message except it is requested by a **MANUFACTURER_REQ**.

MANUFACTURER_IND	Command	**0xFF**
	Subcommand	**0x82**

Parameter	**Type**	**Comment**
Controller	dword	
Manu ID	dword	Manufacturer specific ID (should be unique)
Manufacturer specific		Manufacturer specific data

Note

This message shall not be sent from **COMMON-ISDN-API** without initial application request from an application by means of **MANUFACTURER_REQ**.

5.42 MANUFACTURER_RESP

Description

With this message an application confirms receipt of a **MANUFACTURER_IND**.

MANUFACTURER_RESP	Command	**0xFF**
	Subcommand	**0x83**

Parameter	Type	Comment
Controller	dword	
Manu ID	dword	Manufacturer specific ID (should be unique)
Manufacturer specific		Manufacturer specific data

5.43 RESET_B3_REQ

Description

With this message the specified logical connection is reset. The logical connection is identified by the parameter *NCCI*.

RESET_B3_REQ	Command	0x87
	Subcommand	0x80

Parameter	Type	Comment
NCCI	dword	Network Control Connection Identifier
NCPI	struct	Network Control Protocol Information

Note

The meaning of the parameter *NCPI* depends on the protocol used.

The reaction to a **RESET_B3_REQ** depends on the selected layer 3 protocol. If ISO 8208, T.90, X.25 DCE or X.25 PLP in the D channel was selected, the reset procedure is performed in accordance with the protocol recommendations. In case of a transparent layer 3, a reset procedure in layer 2 is initiated.

If a reset procedure is not defined for the protocol a **RESET_B3_REQ** causes the controller to generate a **RESET_B3_CONF** with info value **reset procedure not supported by current protocol** (0x300D). No further action is taken.

After successfully initiating a reset on a logical connection, an application is not allowed to transmit data until the resulting RESET_B3_IND (or DISCONNECT_B3_IND) message is received.

Loss of data may occur during reset procedure!

5.44 RESET_B3_CONF

Description

With this message the controller confirms the initiation of resetting a logical connection.

RESET_B3_CONF	Command	0x87
	Subcommand	0x81

Parameter	Type	Comment
NCCI	dword	Network Control Connection Identifier
Info	word	**0:** reset initiated **0x0001:** NCPI not supported by current protocol, NCPI ignored **0x2001:** message not supported in current state **0x2002:** illegal NCCI **0x2007:** illegal message parameter coding **0x3008:** NCPI not supported **0x300D:** reset procedure not supported by current protocol

5.45 RESET_B3_IND

Description

With this message the resetting of a logical connection is indicated. The logical connection is identified by a *NCCI*.

RESET_B3_IND	Command	**0x87**
	Subcommand	**0x82**

Parameter	Type	Comment
NCCI	dword	Network Control Connection Identifier
NCPI	struct	Network Control Protocol Information

Note

The meaning of the parameter *NCPI* depends on the protocol used.

In case of transparent layer 3 the re-establishment of layer 2 is indicated.

This message may indicate a loss of data!

5.46 RESET_B3_RESP

Description

With this message the application confirms the resetting of a logical connection.

RESET_B3_RESP	Command	**0x87**
	Subcommand	**0x83**

Parameter	Type	Comment
NCCI	dword	Network Control Connection Identifier

5.47 SELECT_B_PROTOCOL_REQ

Description

This message allows an application to change the current protocol during the lifetime of a physical connection after receiving the message CONNECT_ACTIVE_IND. The support of this message is optional. If a particular **COMMON-ISDN-API** implementation does not support this switching the info parameter of the corresponding **SELECT_B_PROTOCOL_CONF** will be set to **message not supported in current state** (0x2001).

SELECT_B_PROTOCOL_REQ	Command	0x41
	Subcommand	0x80

Parameter	Type	Comment
PLCI	dword	Physical Link Connection Identifier
B protocol	struct	Protocol definition

5.48 SELECT_B_PROTOCOL_CONF

Description

This message confirms the execution of switching the protocol stack for a physical connection. Any error will be shown in *info*.

SELECT_B_PROTOCOL_CONF	Command	**0x41**
	Subcommand	**0x81**

Parameter	Type	Comment
PLCI	dword	Physical Link Connection Identifier
Info	word	**0:** protocol switch successful **0x2001:** message not supported in current state **0x2002:** illegal PLCI **0x2007:** illegal message parameter coding **0x3001:** B1 protocol not supported **0x3002:** B2 protocol not supported **0x3003:** B3 protocol not supported **0x3004:** B1 protocol parameter not supported **0x3005:** B2 protocol parameter not supported **0x3006:** B3 protocol parameter not supported **0x3007:** B protocol combination not supported

6 PARAMETER DESCRIPTIONS

This section describes the parameters used in **COMMON-ISDN-API** messages. Each parameter is listed with its type, possible values and reference to the messages in which the parameter appears.

Some parameter values are defined according to ETS 300 102-1 or Q.931. In that case there is no private **COMMON-ISDN-API** coding for these parameters. These parameters are coded as **COMMON-ISDN-API** structures starting with a length octet and the remainder of the parameter being coded as defined in ETS 300 102-1 / Q.931 from octet three onwards. References to the contents of a structure in this chapter always use index 0 to identify the first octet of information, i.e. the octet following the length octet.

Parameters may not be ommitted, instead an empty structure shall be used. An empty structure shall be coded as a single octet containing a value of 0.

Default values as described in the following section have to be implemented in **COMMON-ISDN-API**. They need not be valid for external ISDN equipment; in that case the external equipment defines the default values for its usage.

Parameters may again contain parameters which are refered to as 'sub parameters'.

Additional Info (struct)

The purpose of the parameter *additional info* is to exchange signalling protocol specific information of the network. Depending on the signalling protocol only relevant elements of this structure will be used (e.g. the B channel information will be ignored in the message DISCONNECT_REQ).

The parameter has the following structure:

struct	**B channel information**
struct	**Keypad facility (coded according to ETS 300 102-1 / Q.931)**
struct	**User user data (coded according to ETS 300 102-1 / Q.931)**
struct	**Facility data array, which is used to transfer additional parameters coded according to ETS 300 102-1 / Q.931 starting from octet 1. This field is used to transport one or more complete facility data information elements.**

This information element appears in:

ALERT_REQ
CONNECT_REQ
CONNECT_IND
CONNECT_RESP
DISCONNECT_REQ
INFO_REQ

B Channel Information (struct)

The purpose of the sub parameter *B channel information* is to choose between B channel data exchange, D channel data exchange or pure user-user data exchange. If this struct is empty the default value is assumed.

This sub parameter is coded as a structure, to give an easy way of extending its contents in future changes. At the moment, it is coded as a structure of two bytes length and has one element:

word Channel:
- **0 : use B channel (default value)**
- **1 : use D channel**
- **2 : use neither B channel or D channel**

This sub parameter appears in parameter:

Additional information

B Protocol (struct)

The purpose of the parameter *B protocol* is to select and configure the B channel protocols. There is a protocol identifier and configuration information for each layer. If this struct is empty the default value is assumed.

The parameter has the following structure:

word	**B1 protocol : Physical layer and framing**
word	**B2 protocol : Data link layer**
word	**B3 protocol : Network layer**
struct	**B1 configuration : Physical layer and framing parameter**
struct	**B2 configuration : Data link layer parameter**
struct	**B3 configuration : Network layer parameter**

This information element appears in:

CONNECT_REQ
CONNECT_RESP
SELECT_B_PROTOCOL_REQ

B1 Protocol (word)

The purpose of the sub parameter *B1 protocol* is to specify the physical layer and framing used for this connection.

The following values are defined:

0:	**64 kBit/s with HDLC framing. This is the default B1 protocol.**
1:	**64 kBit/s bit transparent operation with byte framing from the network**
2:	**V.110 asynchronous operation with start/stop byte framing**
3:	**V.110 synchronous operation with HDLC framing**
4:	**T.30 modem for fax group 3**
5:	**64 kBit/s inverted with HDLC framing.**
6:	**56 kBit/s bit transparent operation with byte framing from the network**

This sub parameter appears in parameter:

B protocol

B2 Protocol (word)

The purpose of the sub parameter *B2 protocol* is to specify the data link layer used for this connection.

The following values are defined:

0:	**ISO 7776 (X.75 SLP) This is the default B2 protocol.**
1:	**Transparent**
2:	**SDLC**
3:	**LAPD according Q.921 for D channel X.25**
4:	**T.30 for fax group 3**
5:	**Point to Point Protocol (PPP)**
6:	**Transparent (ignoring framing errors of B1 protocol)**

This sub parameter appears in parameter:

B protocol

B3 Protocol (word)

The purpose of the sub parameter *B3 protocol* is to specify the network layer used for this connection.

The following values are defined:

0:	**Transparent. This is the default B3 protocol**
1:	**T.90NL with compatibility to T.70NL according to T.90 Appendix II.**
2:	**ISO 8208 (X.25 DTE-DTE)**
3:	**X.25 DCE**
4:	**T.30 for fax group 3**

This sub parameter appears in parameter:

B protocol

The purpose of the sub parameter *B1 configuration* is to offer additional configuration information for the B1 protocol. The parameter has the following structure:

word	Rate	This parameter has different meaning and default values depending on the selected B1 protocol: • **B1 protocol 0:** not applicable • **B1 protocol 1:** not applicable • **B1 protocol 2:** the maximum bit rate, coded as unsigned integer value. Default: adaptive • **B1 protocol 3:** the maximum bit rate, coded as unsigned integer value. Default: 56 kBit • **B1 protocol 4:** the maximum bit rate, coded as unsigned integer value. Default: adaptive • **B1 protocol 5:** not applicable • **B1 protocol 6:** not applicable
word	Bits per character/ Transmit Level	This parameter has different meaning and default values depending on the selected B1 protocol: • **B1 protocol 0:** not applicable • **B1 protocol 1:** not applicable • **B1 protocol 2:** bits per character, coded as unsigned integer value. Default: 8 • **B1 protocol 3:** not applicable • **B1 protocol 4:** the level is coded as signed integer specifying dB's. If this parameter or its value is not supported by the ISDN controller, it is ignored. • **B1 protocol 5:** not applicable • **B1 protocol 6:** not applicable
word	parity	This parameter has different meaning and default values depending on the selected B1 protocol: • **B1 protocol 0:** not applicable • **B1 protocol 1:** not applicable • **B1 protocol 2:** Parity: 0: none, 1: odd, 2: even. Default: no parity • **B1 protocol 3:** not applicable • **B1 protocol 4:** not applicable • **B1 protocol 5:** not applicable • **B1 protocol 6:** not applicable
word	stop bits	This parameter has different meaning and default values depending on the selected B1 protocol: • **B1 protocol 0:** not applicable • **B1 protocol 1:** not applicable • **B1 protocol 2:** stop bits: 0: 1 stop bit, 1: 2 stop bit. Default: 1 stop bit • **B1 protocol 3:** not applicable • **B1 protocol 4:** not applicable • **B1 protocol 5:** not applicable • **B1 protocol 6:** not applicable

This sub parameter appears in parameter:

B protocol

B2 Configuration (struct)

The purpose of the sub parameter *B2 configuration* is to offer additional configuration information for B2 protocol. It is only used for B2 protocols 0, 2 and 3. The parameter has the following structure:

byte	Address A	This parameter has different meaning and default values depending on the selected B2 protocol: • **B2 protocol 0:** link Address A, default is 0x03 • **B2 protocol 2:** link Address, default is 0xC1 • **B2 protocol 3:** bit 0: ´0´ - automatic TEI assignment procedure shall be used. ´1´ - the TEI value shall be used as fixed TEI. In this case Bit 7 - Bit 1: TEI value
byte	Address B	This parameter has different meaning and default values depending on the selected B2 protocol: • **B2 protocol 0:** link Address B, default is 0x01 • **B2 protocol 2:** not applicable • **B2 protocol 3:** not applicable
byte	Modulo Mode	Mode of operation: • 8 - normal operation (Default) • 128 - extended operation
byte	Window Size	Window size, default is 7.
struct	XID	This parameter has different meaning and default values depending on the selected B2 protocol: • **B2 protocol 0:** not applicable • **B2 protocol 2:** this is the content of the XID response which is sent when a XID command is received. • **B2 protocol 3:** not applicable

This sub parameter appears in parameter:

B protocol

B3 Configuration (struct)

The purpose of the sub parameter *B3 configuration* is to offer additional configuration information for B3 protocol. Different structures of this parameter are defined, depending on the B3 protocol:

For B3 protocols 0 (transparent) this parameter does not apply (coded as an empty structure).

For B3 protocols 1, 2 and 3 (T.90NL, ISO8208, X.25 DCE))the following structure is defined:

word	LIC	Lowest incoming channel, default is 0
word	HIC	Highest incoming channel, default is 0
word	LTC	Lowest two-way channel, default is 1
word	HTC	Highest two-way channel, default is 1
word	LOC	Lowest outgoing channel, default is 0
word	HOC	Highest outgoing channel, default is 0
word	Modulo Mode	Mode of operation: • 8 - normal operation (default) • 128 - extended operation
word	Window Size	Used to configure non-standard defaults for the transmit window size, default is 2

For B3 protocol 4 (Fax G3) the following structure is used:

word	resolution	0: standard 1: high
word	format	0: SFF (Default, description in Annex B) 1: Plain FAX Format (modified Huffman coding) 2: PCX 3: DCX 4: TIFF 5: ASCII 6: Extended ANSI 7: Binary-File transfer
struct	station id	ID of the calling station. Coded in ASCII
struct	head line	Headline sent on each fax page. Coded in ASCII

This sub parameter appears in parameter:

B protocol

BC (struct)

The purpose of the parameter *Bearer Capability (BC)* information element is to indicate a requested CCITT Recommendation 1.231 bearer service to be provided by the network. It contains only information which may be used by the network. The information element is coded according to ETS 300 102-1 / Q.931.

This information element appears in:

CONNECT_IND
CONNECT_REQ

Called Party Number (struct)

The purpose of the parameter *called party number* information element is to identify the called party of a call. The information element is coded according to ETS 300 102-1 / Q.931.

Byte 0	Type of number and numbering plan identification (byte 3 of the *called party number* information element, see ETS 300 102). At the calling side the value supplied by the application will be transmitted over the network, **0x80** is the suggested default value. At the called side the value received from the network will be passed to the application.
Bytes 1..n	Number digits of the *called party number* information element.

This information element appears in:

CONNECT_IND
CONNECT_REQ

Called Party Subaddress (struct)

The purpose of the parameter c*alled party subaddress* is to identify the subaddress of the called party of a call. The information element is coded according to ETS 300 102-1 / Q.931.

Byte 0	Type of subaddress At the calling side the value supplied by application will be transmitted over the network, **0x80** is the suggested default value (NSAP according X.213). In this case, the first subaddress information octet should have the value **0x50**. At the called side, the value received from the network will be passed to the application.
Bytes 1..n	Contents of the called party subaddress information element.

This information element appears in:

CONNECT_REQ
CONNECT_IND

Calling Party Number (struct)

The purpose of the parameter c*alling party number* information element is to identify the origin of a call. The information element is coded according to ETS 300 102-1 / Q.931.

Byte 0	Type of number and numbering plan identification (byte 3 of the *calling party number* information element, see ETS 300 102). At the calling side the value supplied by the application will be transmitted over the network, **0x00** is the suggested default value. At the called interface the value received from the network will be passed to the application. The extension bit will always be cleared.
Byte 1	Presentation and screening indicator (byte 3a of the c*alling party number* information element). This byte may be used to allow or suppress the presentation of the caller's number in an incoming call. At the originating interface the value supplied by the application will be transmitted over the network, **0x80** is the suggested default value. With this default value the presentation of the callers number is allowed. **0xA0** will suppress the presentation of the calling number, if the network supports this mechanism. At the called interface the value received from the network will be passed to the application. If this byte was not transmitted from the network, the controller inserts the valid default value **0x80** (user provided, not screened).
Bytes 2..n	Number digits of the *calling party number* information element.

This information element appears in:

CONNECT_REQ
CONNECT_IND
LISTEN_REQ

Calling Party Subaddress (struct)

The purpose of the parameter c*alling party subaddress* information element is to identify a subaddress associated with the origin of a call. The information element is coded according to ETS 300 102-1 / Q.931.

Byte 0	Type of subaddress At the calling side the value supplied by application will be transmitted over the network, **0x80** is the suggested default value (NSAP according X.213). In this case, the first subaddress information octet should have the value **0x50**. At the called side, the value received from the network will be passed to the application.
Bytes 1..n	Contents of the calling party subaddress information element.

This information element appears in:

CONNECT_IND
CONNECT_REQ
LISTEN_REQ

CIP Value (word)

The purpose of parameter *CIP Value* is to identify a complete profile of compatibility information (*Bearer Capability*, *Low Layer Compatibility* and *High Layer Compatibility*). With this parameter standard applications are not required to do complex coding and decoding of the above mentioned information elements.

Some of the *CIP* values only define a *Bearer Capability* (*CIP* 1 to 9) and some values define a combination of *Bearer Capability* and *High Layer Compatibility* (*CIP* 16 to 28). A *Low Layer Compatibility* information element is not defined with the *CIP*. The *Low Layer Compatibility* information element has to be provided by the application if necessary.

The following *CIP* values are defined:

CIP value	Service	Relation to BC/HLC
0		no predefined profile
1	**Speech**	Bearer capability: coding standard: CCITT information transfer capability: speech transfer mode: circuit mode information transfer rate: 64 kBit/s user information layer 1 protocol: G.711 Coding of BC: <0x04, 0x03, 0x80, 0x90, 0xA3> or <0x04, 0x03, 0x80, 0x90, 0xA2>(see note)
2	**unrestricted digital information**	Bearer capability: coding standard: CCITT information transfer capability: unrestricted digital information transfer mode: circuit mode information transfer rate: 64 kBit/s Coding of BC: <0x04, 0x02, 0x88, 0x90>
3	**restricted digital information**	Bearer capability: coding standard: CCITT information transfer capability: restricted digital information transfer mode: circuit mode information transfer rate: 64 kBit/s Coding of BC: <0x04, 0x02, 0x89, 0x90>

4	**3.1 kHz audio**	Bearer capability: coding standard: CCITT information transfer capability: 3.1 kHz audio transfer mode: circuit mode information transfer rate: 64 kBit/s user information layer 1 protocol: G.711 Coding of BC: <0x04, 0x03, 0x90, 0x90, 0xA3> or <0x04, 0x03, 0x80, 0x90, 0xA2>(see note)
5	**7 kHz audio**	Bearer capability: coding standard: CCITT information transfer capability: unrestricted digital information with tones/announcements (this codepoint was formally labelled '7 kHz audio') transfer mode: circuit mode information transfer rate: 64 kBit/s Coding of BC: <0x04, 0x02, 0x91, 0x90>
6	**video**	Bearer capability: coding standard: CCITT information transfer capability: video transfer mode: circuit mode information transfer rate: 64 kBit/s Coding of BC: <0x04, 0x02, 0x98, 0x90>
7	**packet mode**	Bearer capability: coding standard: CCITT information transfer capability: unrestricted digital information transfer mode: packet mode information transfer rate: packet mode layer 2 protocol: X.25 layer 2 layer 3 protocol: X.25 layer 3 Coding of BC: <0x04, 0x04, 0x88, 0xC0, 0xC6, 0xE6>
8	**56 kBit/s rate adaptation**	Bearer capability: coding standard: CCITT information transfer capability: unrestricted digital information transfer mode: circuit mode layer 1 protocol: CCITT standardised rate adaptation V.110/X.30 information transfer rate: packet mode rate: 56 kBit/s Coding of BC: <0x04, 0x04, 0x88, 0x90, 0x21, 0x8F>
9	**unrestricted digital information with tones/announcements**	Bearer capability: coding standard: CCITT information transfer capability: unrestricted digital information with tones/announcements (this codepoint was formally labelled '7 kHz audio') transfer mode: circuit mode information transfer rate: 64 kBit/s layer 1 protocol: H.221, H.242 Coding of BC: <0x05, 0x02, 0x91, 0x90, 0xA5>
10..15	**reserved**	
16	**Telephony**	Bearer Capability according to CIP 1. High Layer Compatibility: coding standard: CCITT interpretation: First characteristics identification is to be used Presentation: High layer protocol profile High layer characteristics identification: Telephony Coding of HLC: <0x7D, 0x02, 0x91, 0x81>

17	**Facsimile Group 2/3**	Bearer Capability according to CIP 4. High Layer Compatibility: coding standard: CCITT interpretation: First characteristics identification is to be used Presentation: High layer protocol profile High layer characteristics identification: Facsimile Group 2/3 Coding of HLC: <0x7D, 0x02, 0x91, 0x84>
18	**Facsimile Group 4 Class 1**	Bearer Capability according to CIP 2. High Layer Compatibility: coding standard: CCITT interpretation: First characteristics identification is to be used Presentation: High layer protocol profile High layer characteristics identification: Facsimile Group 4 Class 1 Coding of HLC: <0x7D, 0x02, 0x91, 0xA1>
19	**Teletex service basic and mixed mode and facsimile service Group 4 Classes II and III**	Bearer Capability according to CIP 2. High Layer Compatibility: coding standard: CCITT interpretation: First characteristics identification is to be used Presentation: High layer protocol profile High layer characteristics identification. Teletex service and facsimile service Group 4 Coding of HLC: <0x7D, 0x02, 0x91, 0xA4>
20	**Teletex service basic and processable mode**	Bearer Capability according to CIP 2. High Layer Compatibility: coding standard: CCITT interpretation: First characteristics identification is to be used Presentation: High layer protocol profile High layer characteristics identification. Teletex service basic and processable mode Coding of HLC: <0x7D, 0x02, 0x91, 0xA8>
21	**Teletex service basic mode**	Bearer Capability according to CIP 2. High Layer Compatibility: coding standard: CCITT interpretation: First characteristics identification is to be used Presentation: High layer protocol profile High layer characteristics identification. Teletex service basic mode Coding of HLC: <0x7D, 0x02, 0x91, 0xB1>
22	**International inter working for Videotex**	Bearer Capability according to CIP 2. High Layer Compatibility: coding standard: CCITT interpretation: First characteristics identification is to be used Presentation: High layer protocol profile High layer characteristics identification. International inter working for Videotex Coding of HLC: <0x7D, 0x02, 0x91, 0xB2>
23	**Telex**	Bearer Capability according to CIP 2. High Layer Compatibility: coding standard: CCITT interpretation: First characteristics identification is to be used Presentation: High layer protocol profile High layer characteristics identification: Telex Coding of HLC: <0x7D, 0x02, 0x91, 0xB5>

24	**Message Handling Systems according to X.400**	Bearer Capability according to CIP 2. High Layer Compatibility: coding standard: CCITT interpretation: First characteristics identification is to be used Presentation: High layer protocol profile High layer characteristics identification: Message Handling Systems according X.400 Coding of HLC: <0x7D, 0x02, 0x91, 0xB8>
25	**OSI application according to X.200**	Bearer Capability according to CIP 2. High Layer Compatibility: coding standard: CCITT interpretation: First characteristics identification is to be used Presentation: High layer protocol profile High layer characteristics identification: OSI application according X.200 Coding of HLC: <0x7D, 0x02, 0x91, 0xC1>
26	**7 kHz Telephony**	Bearer Capability according to CIP 9. High Layer Compatibility: coding standard: CCITT interpretation: First characteristics identification is to be used Presentation: High layer protocol profile High layer characteristics identification: Telephony Coding of HLC: <0x7D, 0x02, 0x91, 0x81>
27	**Video Telephony, first connection**	Bearer Capability according to CIP 9. High Layer Compatibility: coding standard: CCITT interpretation: First characteristics identification is to be used Presentation: High layer protocol profile High layer characteristics identification: Video telephony (Rec. F.721) Extended high layer characteristics identification: Capability set of initial channel of H.221 Coding of HLC: <0x7D, 0x03, 0x91, 0xE0, 0x01>
28	**Video Telephony, second connection**	Bearer Capability according to CIP 2 High Layer Compatibility: coding standard: CCITT interpretation: First characteristics identification is to be used Presentation: High layer protocol profile High layer characteristics identification: Video telephony (Rec F.721) Extended high layer characteristics identification: Capability set of subsequent channel of H.221 Coding of HLC: <0x7D, 0x03, 0x91, 0xE0, 0x02>

Note

This coding applies to ISDN with a default of A-Law coding for speech/audio. For ISDN with a default of µ Law coding the corresponding values will be used.

This information element appears in:

CONNECT_REQ
CONNECT_IND

CIP mask (dword)

The purpose of the parameter *CIP mask* is to select basic classes of incoming calls. The bit position within this mask identifies the related CIP value. When an incoming call is received, **COMMON-ISDN-API** tries to match this incoming call to the defined CIP values (more than one value may match). A CONNECT_IND message is sent to the application when the bit position within the *CIP mask* of any matching CIP value is set to ´1´. The CIP value in the CONNECT_IND message is set to the highest matching CIP value.

The following rules are defined to find matching CIPs:

1. CIP values which define a Bearer Capability only (CIP 1 to CIP 9) will generate a match with any incoming call which includes a Bearer Capability with the same information. Additional information included in the Bearer Capability information element will be ignored. The match is generated regardless of any Low Layer Compatibility or High Layer Compatibility received.

2. CIP values which define a Bearer Capability and a High Layer Compatibility (CIP 16 to CIP 28) will generate a match with any incoming call which includes a Bearer Capability and a High Layer Compatibility with the same identical information. The match is generated regardless of any Low Layer Compatibility received.

Bit 0 in the *CIP mask* has a special meaning. When no other matching bit is set in the *CIP mask* but the Bit 0, a CONNECT_IND is sent to the application with a CIP value of 0. In this case the application has to evaluate the parameters Bearer Capability, Low Layer Compatibility and High Layer Compatibility to decide whether it is compatible to the call or not.

Examples:

Service	Bits to be set in the CIP mask	
Telephony Application	1	For calls within ISDN from equipment which does not send High Layer Compatibility info.
	4	For calls from the analogue network.
	16	For call within ISDN equipment which sends High Layer Compatibility info.
Fax Group 2/3 Application	4	For calls from the analogue network.
	17	For calls within ISDN.
Non standard 64 kBit/s data applications	2	No checking of High Layer Compatibility information is provided. The application should verify that no High Layer Compatibility information is received.
Non standard 56 kBit/s data applications	8	No checking of High Layer Compatibility information is provided. The application has to verify that no High Layer Compatibility information is received.
Fax Group 4 application	2	For calls from equipment which does not send High Layer Compatibility information. The application has to verify that no High Layer Compatibility information is received.
	18	For call from equipment which sends High Layer Compatibility information.

This information element appears in:

LISTEN_REQ

Connected Number (struct)

The purpose of the parameter c*onnected number* information element is to indicate which number is connected to a call. The information element is coded according to ETS 300 097.

Byte 0	Type of number and numbering plan identification (byte 3 of the connected number information element, see ETS 300 097). In the direction application to **COMMON-ISDN-API**, the value supplied by the application will be transmitted over the network, **0x00** is the suggested default value. In the direction **COMMON-ISDN-API** to application, the value received from the network will be passed to the application. The extension bit will always be cleared.
Byte 1	Presentation and screening indicator (byte 3a of the connected number information element). In the direction application to **COMMON-ISDN-API**, the value supplied by the application will be transmitted over the network, **0x80** is the suggested default value. In the direction **COMMON-ISDN-API** to application, the value received from the network will be passed to the application. If this byte was not transmitted over the network, the controller provides the value **0x80** (user provided, not screened).
Bytes 2..n	Number digits of the connected number information element.

This information element appears in:

CONNECT_ACTIVE_IND
CONNECT_RESP

Connected Subaddress (struct)

The purpose of the parameter *connected subaddress* information element is to identify the subaddress of the connected user of a call. The information element is coded according to ETS 300 097.

Byte 0	Type of subaddress At the calling side the value supplied by application will be transmitted over the network, **0x80** is the suggested default value (NSAP according X.213). In this case, the first subaddress information octet should have the value **0x50**. At the called side, the value received from the network will be passed to the application.
Bytes 1..n	Contents of the connected subaddress information element.

This information element appears in:

CONNECT_ACTIVE_IND
CONNECT_RESP

Controller (dword)

The purpose of the parameter *controller* is to address a hardware unit, that gives access to an ISDN at the application's disposal. A *controller* supports none, one or several physical and logical connections. The parameter *controller* is a dword (to be compatible in size with PLCI and NCCI) with the range from 1 to 127 (0 reserved). Bit 7 additionally contains the information, if the message is used for internal (0) or external (1) equipment. Controllers are numbered sequentially and can be designed to handle external equipment additional to internal functionality or exclusively provide access to external equipment. External equipment e.g. is a handset.

Definition of external equipment behaviour, e.g. B channel handling, is not covered by **COMMON-ISDN-API**.

Format for *controller*:

0	0	0	Ext./Int.	Controller
31 … 16		8	7	6 … 0

This information element appears in:

CONNECT_REQ
FACILITY_REQ
FACILITY_CONF
FACILITY_IND
FACILITY_RESP
LISTEN_REQ
LISTEN_CONF
MANUFACTURER_REQ
MANUFACTURER_CONF
MANUFACTURER_IND
MANUFACTURER_RESP

Data (dword)

The purpose of the parameter *data* is to exchange a 32 bit pointer to the data area containing the information.

This information element appears in:

DATA_B3_REQ
DATA_B3_IND

Data Length (word)

The purpose of the parameter *data length* is to specify the length of the data.

This information element appears in:

DATA_B3_REQ
DATA_B3_IND

Data Handle (word)

The purpose of the parameter *data handle* is to identify the data area in data exchange messages.

This information element appears in:

DATA_B3_REQ
DATA_B3_CONF
DATA_B3_IND
DATA_B3_RESP

Facility Selector (word)

The purpose of the parameter *facility selector* is to identify the requested **COMMON-ISDN-API** facility.

The defined values are:

0	**Handset (external ISDN equipment) support**
1	**DTMF (Dual Tone Multi Frequency)**

This information element appears in:

FACILITY_REQ
FACILITY_CONF
FACILITY_IND
FACILITY_RESP

Facility Request Parameter (struct)

The purpose of the parameter *facility request parameter* is to offer additional information concerning the message FACILITY_REQ.

This parameter is coded depending on *facility selector* as a structure with following elements:

Facility selector:

0 **Parameter does not apply (coded as empty structure)**

1 **DTMF (Dual Tone Multi Frequency):**

Function	word	1: Start DTMF listen on B channel data 2. Stop DTMF listen 3: Send DTMF digits 4 to n: Reserved
Tone-Duration	word	Time in ms for one digit, default is 40 ms
Gap-Duration	word	Time in ms between the digits, default is 40 ms
DTMF-Digits	struct	Characters to be sent, coded as IA5-char. '0' to '9', '*', '#', 'A', 'B', 'C' or 'D', each character generates a unique DTMF- Tone.

Sending of DTMF characters will interrupt the transmission of **DATA_B3_REQ**. After DTMF generation, the data transmission will be resumed

This information element appears in:

FACILITY_REQ

Facility Confirmation Parameter (struct)

The purpose of the parameter *facility confirmation parameter* is to offer additional information concerning the message FACILITY_CONF.

This parameter is coded depending on *facility selector* as a structure with following elements:

Facility selector:

0 Parameter does not apply (coded as structure with a length of 0)

1 DTMF (Dual Tone Multi Frequency):

DTMF information	word	0: sending of DTMF info successfully initiated 1: incorrect DTMF digit 2: unknown DTMF request

This information element appears in:

FACILITY_CONF

Facility Indication Parameter (struct)

The purpose of the parameter *facility indication parameter* is to offer additional information concerning the message FACILITY_IND.

This parameter is coded depending on *facility selector* as a structure with following elements:

Facility selector:

0 Handset Support:

handset digits	byte array	Received characters, coded as IA5-char. '0' to '9', '*', '#', 'A', 'B', 'C' or 'D'; or '+': Handset off-hook '-': Handset on-hook

Facility selector:

1 DTMF (Dual Tone Multi Frequency):

DTMF digits	byte array	Received characters, coded as IA5-char. '0' to '9', '*', '#', 'A', 'B', 'C' or 'D'

This information element appears in:

FACILITY_IND

Facility Response Parameter (struct)

The purpose of the parameter *facility respond parameter* is to offer additional information concerning the message FACILITY_RESP.

This parameter is coded depending on *facility selector* as a structure with following elements:

Facility selector:

0	**Parameter does not apply (coded as structure with a length of 0)**
1	**Parameter does not apply (coded as structure with a length of 0)**

This information element appears in:

FACILITY_RESP

Flags (word)

The purpose of the parameter *flags* is to exchange additional protocol dependent information about the data.

Bit 0	qualifier bit
Bit 1	more data bit
Bit 2	delivery confirmation bit
Bit 3	expedited data bit
Bit 15	framing error bit, data may be invalid (only with corresponding B2 protocol)

This information element appears in:

DATA_B3_REQ
DATA_B3_IND

HLC (struct)

The purpose of the parameter *High Layer Compatibility (HLC)* information element is to provide a means which should be used by the remote user for compatibility checking. The information element is coded according to ETS 300 102-1 / Q.931.

This information element appears in:

CONNECT_IND
CONNECT_REQ

The purpose of the parameter *info* is to provide error information to the application. For each error which can be detected by the controller a unique code is defined, independing from the context of the error.

COMMON-ISDN-API shall not generate other information values as defined below. In case of future extension of possible information values however an application should interpret any information value except class **0x00xx** as an indication that the corresponding request was rejected from **COMMON-ISDN-API**. Class **0x00xx** indicates the successful handling of the corresponding request and returns additional information.

class 0x00xx: informative values (corresponding message was processed)

Value	Reason
0	request accepted
0x0001	NCPI not supported by current protocol, NCPI ignored
0x0002	flags not supported by current protocol, flags ignored
0x0003	alert already sent by another application

class 0x10xx: error information concerning CAPI_REGISTER

Value	Reason
0x1001	too many applications
0x1002	logical block size too small, must be at least 128 bytes
0x1003	buffer exceeds 64 kByte
0x1004	message buffer size too small, must be at least 1024 bytes
0x1005	max. number of logical connections not supported
0x1006	reserved
0x1007	the message could not be accepted because of an internal busy condition
0x1008	OS Resource error (e.g. no memory)
0x1009	**COMMON-ISDN-API** not installed
0x100A	Controller does not support external equipment
0x100B	Controller does only support external equipment

class 0x11xx: error information concerning message exchange functions

Value	Reason
0x1101	illegal application number
0x1102	illegal command or subcommand or message length less than 12 octets
0x1103	the message could not be accepted because of a queue full condition. The error code does not imply that **COMMON-ISDN-API** cannot receive messages directed to another controller, PLCI or NCCI.
0x1104	queue is empty
0x1105	queue overflow, a message was lost. This indicates a configuration error. The only recovery from this error is to perform a CAPI_RELEASE.
0x1106	unknown notification parameter
0x1107	the message could not be accepted because of an internal busy condition
0x1108	OS Resource error (e.g. no memory)
0x1109	**COMMON-ISDN-API** not installed
0x110A	Controller does not support external equipment
0x110B	Controller does only support external equipment

class 0x20xx: error information concerning resource / coding problems

Value	Reason
0x2001	message not supported in current state
0x2002	illegal Controller/PLCI/NCCI
0x2003	out of PLCI
0x2004	out of NCCI
0x2005	out of LISTEN
0x2006	out of FAX resources (protocol T.30)
0x2007	illegal message parameter coding

class 0x30xx: error information concerning requested services

Value	Reason
0x3001	B1 protocol not supported
0x3002	B2 protocol not supported
0x3003	B3 protocol not supported
0x3004	B1 protocol parameter not supported
0x3005	B2 protocol parameter not supported
0x3006	B3 protocol parameter not supported
0x3007	B protocol combination not supported
0x3008	NCPI not supported
0x3009	CIP Value unknown
0x300A	flags not supported (reserved bits)
0x300B	facility not supported
0x300C	data length not supported by current protocol
0x300D	reset procedure not supported by current protocol

This information element appears in:

CONNECT_B3_CONF
CONNECT_CONF
INFO_CONF
DATA_B3_CONF
DISCONNECT_B3_CONF
DISCONNECT_CONF
LISTEN_CONF
RESET_B3_CONF
SELECT_B_PROTOCOL_CONF

Info Element (struct)

The purpose of the parameter *info element* depends on the value of the parameter info number.

If the info number specifies an information element, the *info element* contains that information element with the coding as defined in ETS 300 102-1 / Q.931.

If the info number specifies a charging information *info element* contains a dword indicating the sum of charges accumulated by the network up to this moment.

If the info number specifies a message type the *info element* is an empty **COMMON-ISDN-API** struct.

This information element appears in:

INFO_IND

Info Mask (dword)

The parameter *info mask* specifies which type of information for a physical connection or controller will be provided by **COMMON-ISDN-API**. The selected information will be indicated within the message INFO_IND to the application. A given *info mask* (set in LISTEN_REQ) is valid until it is superseded by another LISTEN_REQ and applies to all information concerning the corresponding application. The *info mask* is coded as a bit field. A bit set to 1 means that corresponding INFO_IND messages will be generated, a bit set to 0 means the specified information will be suppressed. In the default *info mask* all bits are set to 0. If an application wants to change this value it has to send a LISTEN_REQ message even if it does not want to be informed about incoming calls.

Bit 0	Cause; cause information given by the net during disconnection. The parameter info element of the corresponding INFO_IND message is a **COMMON-ISDN-API** struct which contains the cause information element defined in ETS 300 102-1 and Q.931 (both 4.5.12).
Bit 1	Date/time; date/time information indicated by the net. The parameter info element of the corresponding INFO_IND message contains the date/time information element defined in ETS 300 102-1 and Q.931 (both 4.6.1).
Bit 2	Display; display information to be displayed to the user. The parameter info element of the corresponding INFO_IND message contains the display information element defined in ETS 300 102-1 and Q.931 (both 4.5.15).
Bit 3	User-user; user-user information that is transparently carried by the net. The parameter info element of the corresponding INFO_IND message contains the user-user information element defined in ETS 300 102-1 and Q.931 (both 4.5.29).
Bit 4	Call progression; information referring to the progress of the call. There are five different INFO_IND messages that correspond to this information type, each with a unique info number. The first indication contains the information element progress indicator as defined in ETS 300 102-1 and Q.931. The other four messages indicate the occurrence of the network events SETUP ACKNOWLEDGE, CALL PROCEEDING, ALERTING and PROGRESS. In these cases the parameter info number indicates the corresponding message type and the info element is an empty **COMMON-ISDN-API** struct.
Bit 5	Facility; facility information to indicate the invocation and operation of supplementary services. The parameter info element of the corresponding INFO_IND message contains the facility information element defined in ETS 300 102-1 and Q.931 (both 4.6.2).
Bit 6	Charging information; connection oriented charging information provided by the net. There are two different INFO_IND messages with unique info number values that correspond to this information type. The first one shows the sum of charging units indicated by the net up to this moment, the second the sum of charges in the national currency indicated by the net up to this moment. In both cases the parameter info element is coded as a **COMMON-ISDN-API** struct containing a dword. It is highly recommended to provide only one of this two types of charging information to the user and to transform one type to the other. However, in some networks this might be impossible due to the information provided from the net. In these cases it is not defined, if the current charges are represented by only one or both or the sum of this indicated charges.
Bits 7-31	Reserved, must be set to 0

This information element appears in:

LISTEN_REQ

Info Number (word)

The purpose of the parameter *info number* specifies the coding of the parameter *info element* and the type of information which is carried by this INFO_IND message. The high byte is structured as a bit field and indicates which type of information is held in the low byte.

Bit 15	If this bit set to 1 the low byte contains a message type, if it is set to 0 the low byte represents an information element type.
Bits 14	If this bit is set to 1 the low byte indicates supplementary information not covered by network events or information elements. In this case bit 15 must be set to 0.
Bits 13-8	Reserved, set to 0.

If bit 15 is set, the low byte containing the message type is coded according to ETS 300 102-1 / Q.931. In this case the INFO_IND message indicates the occurrence of a network event according to the specified message and the parameter *info element* is an empty **COMMON-ISDN-API** struct.

If bits 14 and 15 are cleared, the low byte represents an information element type coding according to ETS 300 102-1 / Q.931. The parameter *info element* contains the content of the information element.

If bit 14 is set, the low byte represents supplementary information. The defined values are

0	**sum of charges in charging units.** In this case the parameter *info element* contains the content of the information element.
1	**sum of charges in national currency.** In this case the parameter *info element* contains the content of the information element.

formation element appears in:

INFO_IND

LLC (struct)

The purpose of the parameter *Low Layer Compatibility (LLC)* information element is to provide a means which should be used for compatibility checking by an addressed entity (e.g. a remote user or an inter working unit or a high layer function network node addressed by the calling user). The *Low Layer Compatibility* information element is transferred transparently by ISDN between the call originating entity (e.g. the calling user) and the addressed entity. If *Low Layer Compatibility* negotiation is allowed by the network, the *Low Layer Compatibility* information element is also passed transparently from the addressed entity to the originating entity. The information element is coded according to ETS 300 102-1 / Q.931.

This information element appears in:

CONNECT_ACTIVE_IND
CONNECT_IND
CONNECT_REQ
CONNECT_RESP

Manu ID (dword)

The purpose of the parameter *Manu ID* is to exchange a dword inside MANUFACTURER-Messages which identifies the manufacturer. Every manufacturer offering MANUFACTURER-Messages should choose a unique value (e.g. shortcut of company name).

This information element appears in:

MANUFACTURER_REQ
MANUFACTURER_RESP
MANUFACTURER_CONF
MANUFACTURER_IND

Manufacturer Specific

The purpose of the parameter *manufacturer Specific* is to exchange manufacturer specific information.

This information element appears in:

MANUFACTURER_REQ
MANUFACTURER_RESP
MANUFACTURER_CONF
MANUFACTURER_IND

NCCI (dword)

The purpose of the parameter *NCCI* is to identify a logical connection. The *NCCI* is given by **COMMON-ISDN-API** during creation of the logical connection. Depending on the layer 3 protocol selection (e.g. ISO 8208), it is possible to have multiple *NCCIs* based on one PLCI. The *NCCI* is a dword with a range from 1 to 65535 (0 reserved), coded as described below, and includes additionally the corresponding PLCI and controller.

Format for *NCCI*:

NCCI	PLCI	Ext./Int.	Controller
31 … 16	8	7	6 … 0

This information element appears in:

CONNECT_B3_ACTIVE_IND
CONNECT_B3_ACTIVE_RESP
CONNECT_B3_CONF
CONNECT_B3_IND
CONNECT_B3_RESP
DATA_B3_CONF
DATA_B3_IND
DATA_B3_REQ
DATA_B3_RESP
DISCONNECT_B3_CONF
DISCONNECT_B3_IND
DISCONNECT_B3_REQ
DISCONNECT_B3_RESP
FACILITY_REQ
FACILITY_CONF

FACILITY_IND
FACILITY_RESP
RESET_B3_CONF
RESET_B3_IND
RESET_B3_REQ
RESET_B3_RESP

NCPI (struct)

The purpose of the parameter *NCPI* is to provide additional protocol specific information.

For the layer 3 protocols ISO 8208 and X.25 the parameter data of structure *NCPI* are coded as follows:

Byte 0	Bit field **[0]:** Enable the usage of the delivery confirmation procedure in call set-up and data packets (D-Bit). **[1..7]:** Reserved.
Byte 1	Logical channel group number of the permanent virtual circuit (PVC) to be used. In the case of virtual calls (VC) this number must be set to zero.
Byte 2	Logical channel number of the permanent virtual circuit (PVC) to be used. In the case of virtual calls (VC) this number must be set to zero.
Bytes 3..n	Bytes following the packet type identifier field in the X.25 PLP packets.

For layer 3 protocol T.30 (fax group 3) the parameter data of structure *NCPI* are valid only for DISCONNECT_B3_IND and coded as follows (in every other message the structure is empty):

word	Rate	actual used bit rate, coded as unsigned integer value
word	resolution	0: standard 1: high
word	format	0: SFF (Default, description in Annex A) 1: Plain FAX Format (modified Huffman coding) 2: PCX 3: DCX 4: TIFF 5: ASCII 6: Extended ANSI 7: Binary-File transfer
word	pages	number of pages, coded as unsigned integer value
struct	receive id	id of remote side

This information element appears in:

CONNECT_B3_ACTIVE_IND
CONNECT_B3_T90_ACTIVE_IND
CONNECT_B3_IND
CONNECT_B3_REQ
CONNECT_B3_RESP
DISCONNECT_B3_IND
DISCONNECT_B3_REQ
RESET_B3_REQ
RESET_B3_RESP

PLCI (dword)

The purpose of the parameter *PLCI* is to describe a physical connection between two endpoints. The *PLCI* is given by **COMMON-ISDN-API** during creation of the physical connection. The *PLCI* is a dword with the range from 1 to 255 (0 reserved), coded as described below, and additionally includes the controller.

Format for *PLCI*:

0	0	PLCI	Ext./Int.	Controller
31	16	8	7	6 0

This information element appears in:

CONNECT_ACTIVE_IND
CONNECT_ACTIVE_RESP
CONNECT_B3_REQ
CONNECT_CONF
CONNECT_IND
CONNECT_RESP
DISCONNECT_REQ
DISCONNECT_CONF
DISCONNECT_IND
DISCONNECT_RESP
FACILITY_REQ
FACILITY_CONF
FACILITY_IND
FACILITY_RESP
INFO_REQ
INFO_CONF
INFO_IND
INFO_RESP
SELECT_B_PROTOCOL_REQ
SELECT_B_PROTOCOL_CONF

Reason (word)

The purpose of the parameter *reason* is to provide error information to the application regarding the clearing of a physical connection . The defined values are:

0	**normal clearing, no cause available**
0x3301	**protocol error layer 1**
0x3302	**protocol error layer 2**
0x3303	**protocol error layer 3**
0x3304	**another application got that call (see LISTEN_REQ)**
0x34xx	**disconnect cause from the network according to ETS 300 102-1 / Q.931. In the field 'xx' the cause value received within a cause information element (octet 4) from the network is indicated.**

This information element appears in:

DISCONNECT_IND

Reason_B3 (word)

The purpose of the parameter *reason* is to provide error information to the application regarding the clearing of a logical connection . The defined values are:

protocol independent:

0	normal clearing, no cause available
0x3301	protocol error layer 1 (broken line or B channel removed by signalling protocol)

0x3302	protocol error layer 2
0x3303	protocol error layer 3

T.30 specific reasons:

0x3311	connecting not successful (remote station is no fax G3 machine)
0x3312	connecting not successful (training error)
0x3313	disconnected before transfer (remote station does not support transfer mode, e.g. resolution)
0x3314	disconnected during transfer (remote abort)
0x3315	disconnected during transfer (remote procedure error (e.g. unsuccessful repetition of T.30 commands)
0x3316	disconnected during transfer (local tx data underrun)
0x3317	disconnected during transfer (local rx data overflow)
0x3318	disconnected during transfer (local abort)
0x3319	illegal parameter coding (e.g. SFF coding error)

Reject (word)

The purpose of the parameter *reject* is to define the action of **COMMON-ISDN-API** for incoming calls.

The defined values are

0	**Accept the call**
1	**Ignore the call**
2	**reject call, normal call clearing**
3	**reject call, user busy**
4	**reject call, requested circuit/channel not available**
5	**reject call, facility rejected**
6	**reject call, channel unacceptable**
7	**reject call, incompatible destination**
8	**reject call, destination out of order**

This information element appears in:

CONNECT_B3_RESP
CONNECT_RESP

7 STATE DIAGRAM

7.1 User's Guide

To explain the message exchange between CAPI and application, a graphic description is mandated. In the absence of an international standard for the description of a message exchange between two local entities, a new way of presentation was created. The state machines on the following pages are described in the form of a state diagram covering application and controller. This state diagram is a monitor view of an idealised interface. In reality the CAPI is not only an interface definition, it is also a concrete instantiation.

The state diagram on the following pages is split into three separate state machines:

1. LISTEN state machine

2. PLCI state machine (physical connections)

3. NCCI state machine (logical connections)

On every physical connection, identified by a PLCI, several logical layer 3 links could exist, identified by a NCCI. Therefore a splitting into PLCI and NCCI state machine is necessary. A description of "n" physical links with "m" logical links at one time in one state machine is impossible. Therefore only one PLCI or one NCCI at a time is considered in the state machine.

COMMON-ISDN-API messages LISTEN_REQ and LISTEN_CONF are described in a separate state machine, because the availability of a successful LISTEN setting exceeds the lifetime of logical and/or physical connections.

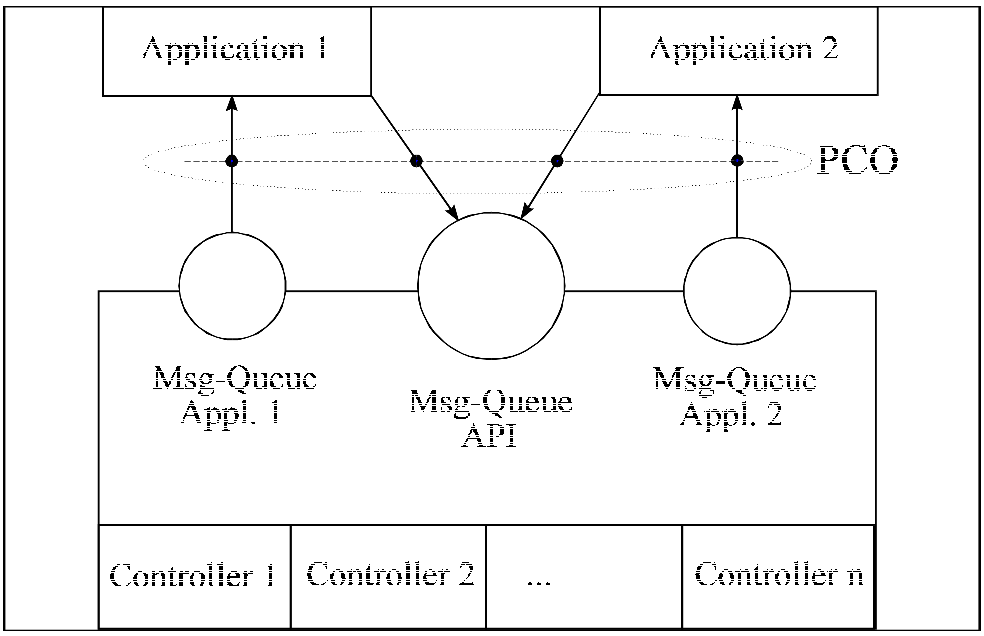


Figure 5: Position of PCO (Point of Control and Observation)

7.2 Explanation

The state diagrams define a faultless exchange of messages. The point of control and observation (PCO) for the message exchange description is on the level of the CAPI operations. For real implementations it is not allowed that an asynchronous exchange of messages results in an error condition.

The state diagrams define the flow of the messages on the PCO without consideration of their possible asynchronicity in real implementations.

Confirmations and responses, which do not evoke a state transition, are not shown in this state diagrams.

In "ANY-State" it is allowed that an expected confirmation on a request or an expected response appears.

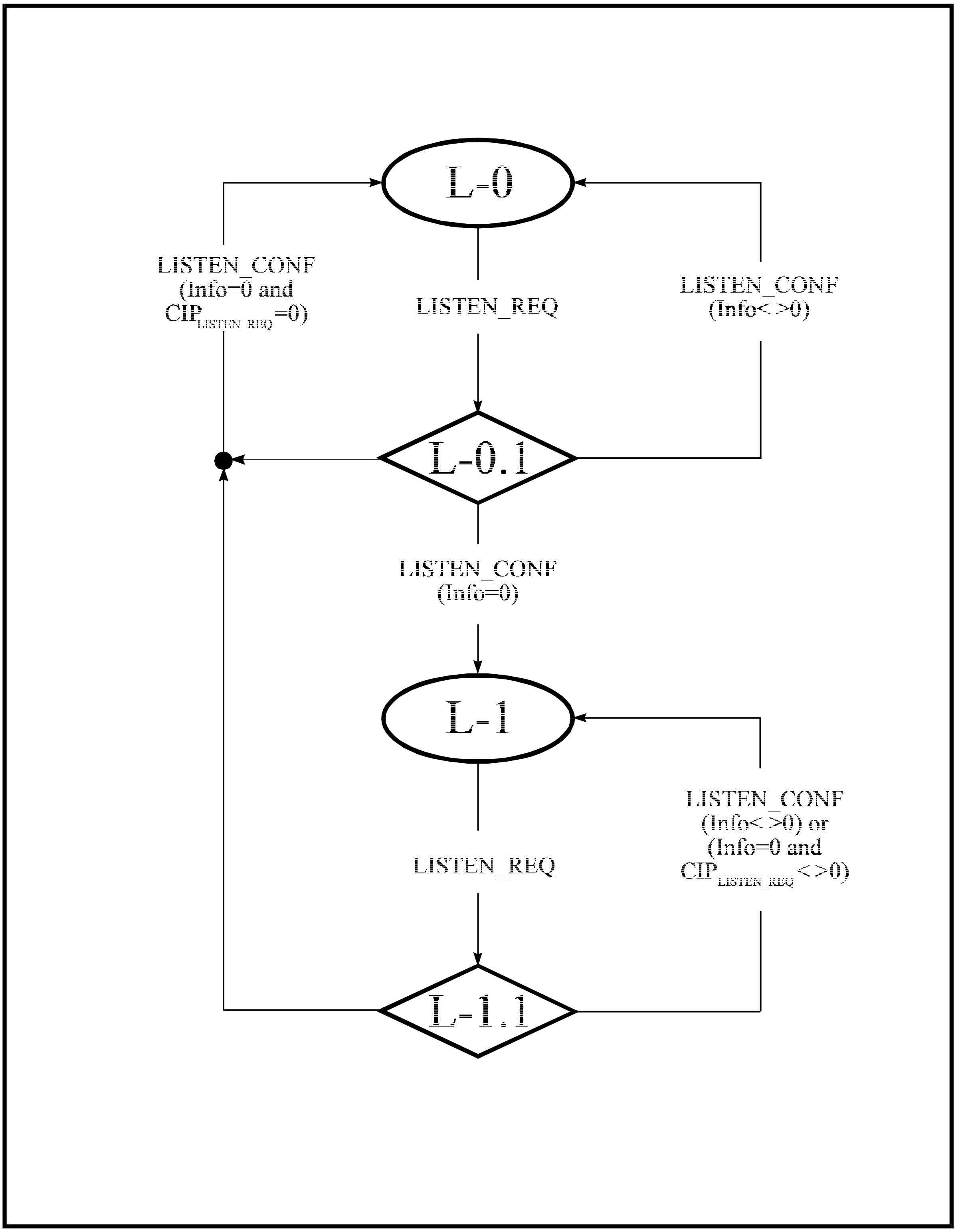
The messages MANUFACTURER_REQ, MANUFACTURER_CONF, MANUFACTURER_IND and MANUFACTURER_RESP could result in incompatibility. They are not described in the state diagrams.

Requests with an invalid PLCI or an invalid NCCI are wrong messages and therefore are not described in the state diagrams.

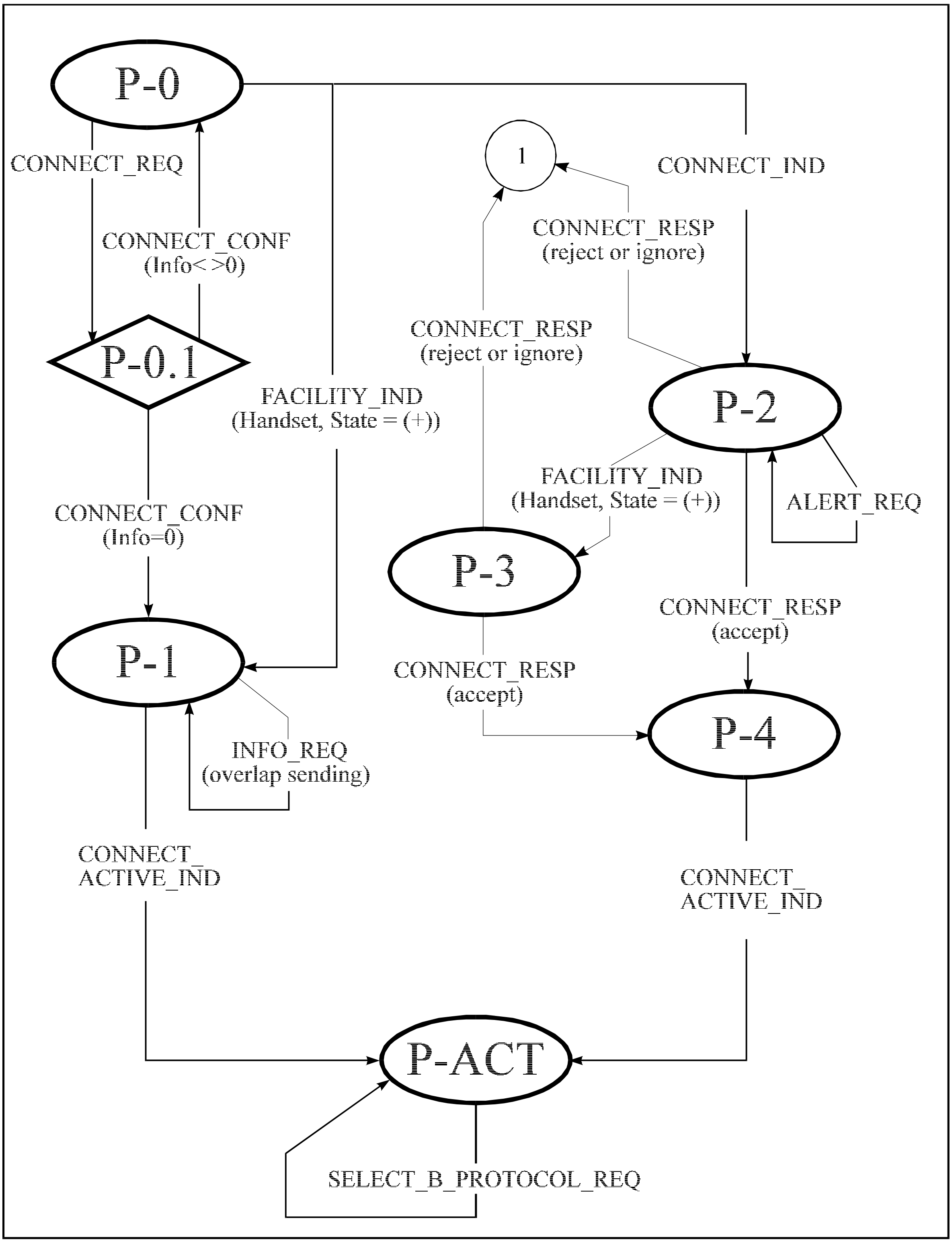
INFO_REQ and INFO_IND are network specific elements which can appear at any time. The use of INFO_REQ especially for "overlap sending" is described in the PLCI-state machine 1/2.

FACILITY_REQ, FACILITY_CONF, FACILITY_IND and FACILITY_RESP are facility specific messages which can appear at any time. Therefore they can occur in every state of the LISTEN- , PLCI- and NCCI- state machine. Especially the FACILITY_IND concerning "Handset Support" is described in the PLCI-state machine 1/2. The flow of the messages for the Handset Support depends on the real handset interface (e.g. AEI, i.e. Additional Equipment Interface) or manufacturer specific codecs. So it is possible, that only a part of the described flow of the messages for the Handset Support is used. But it is not allowed to use the FACILITY messages for the Handset Support in another way, as described in the message definition and the state machines.

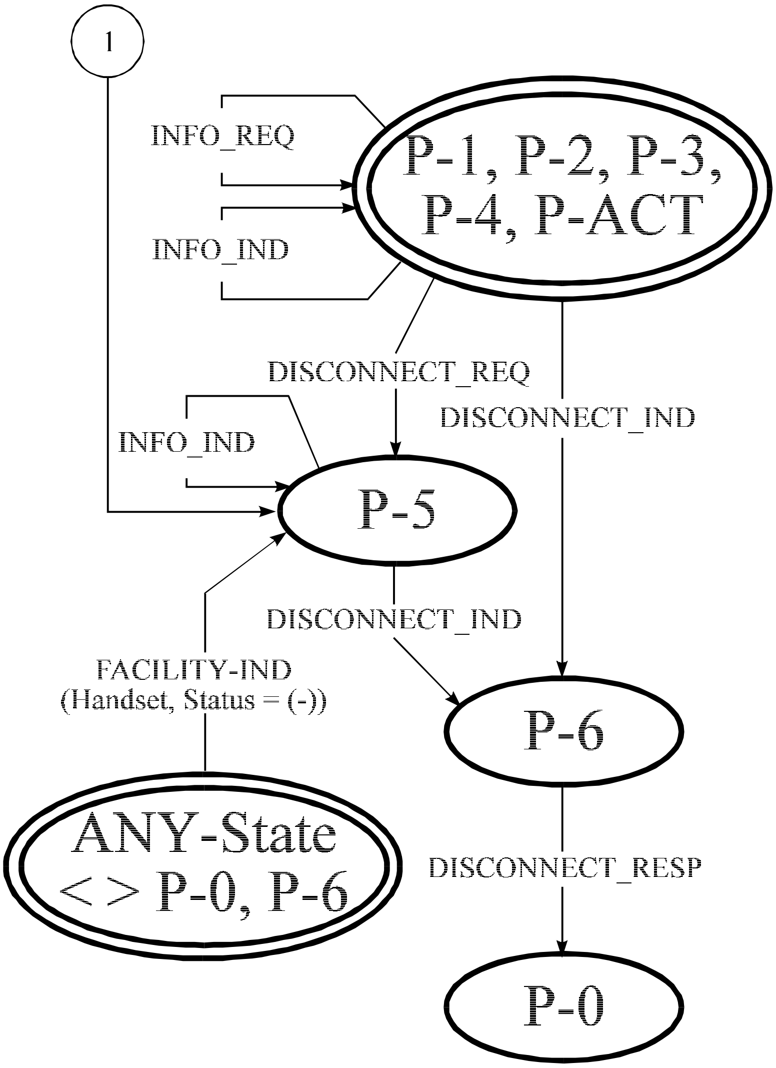
LISTEN - state machine



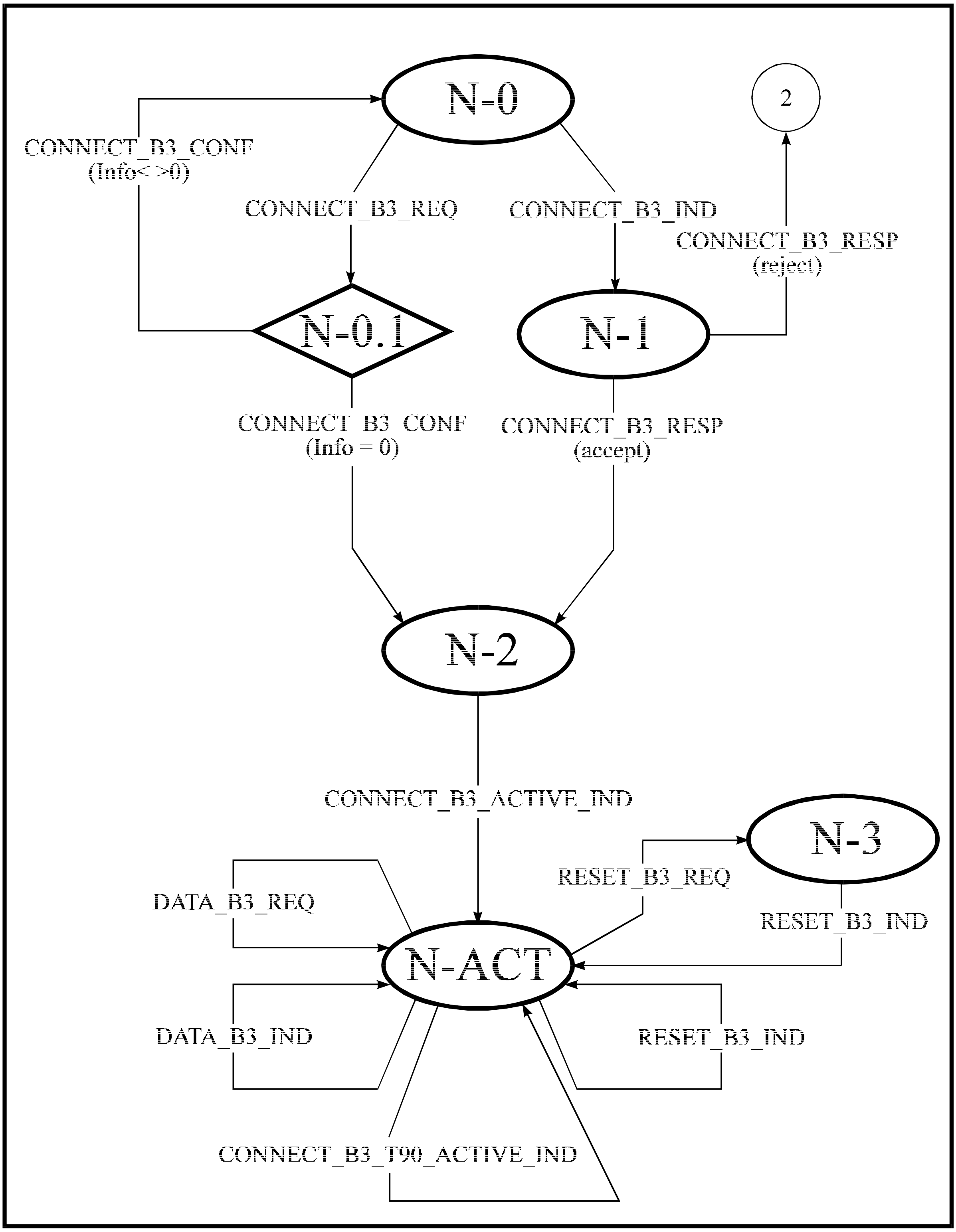
PLCI - state machine 1/2



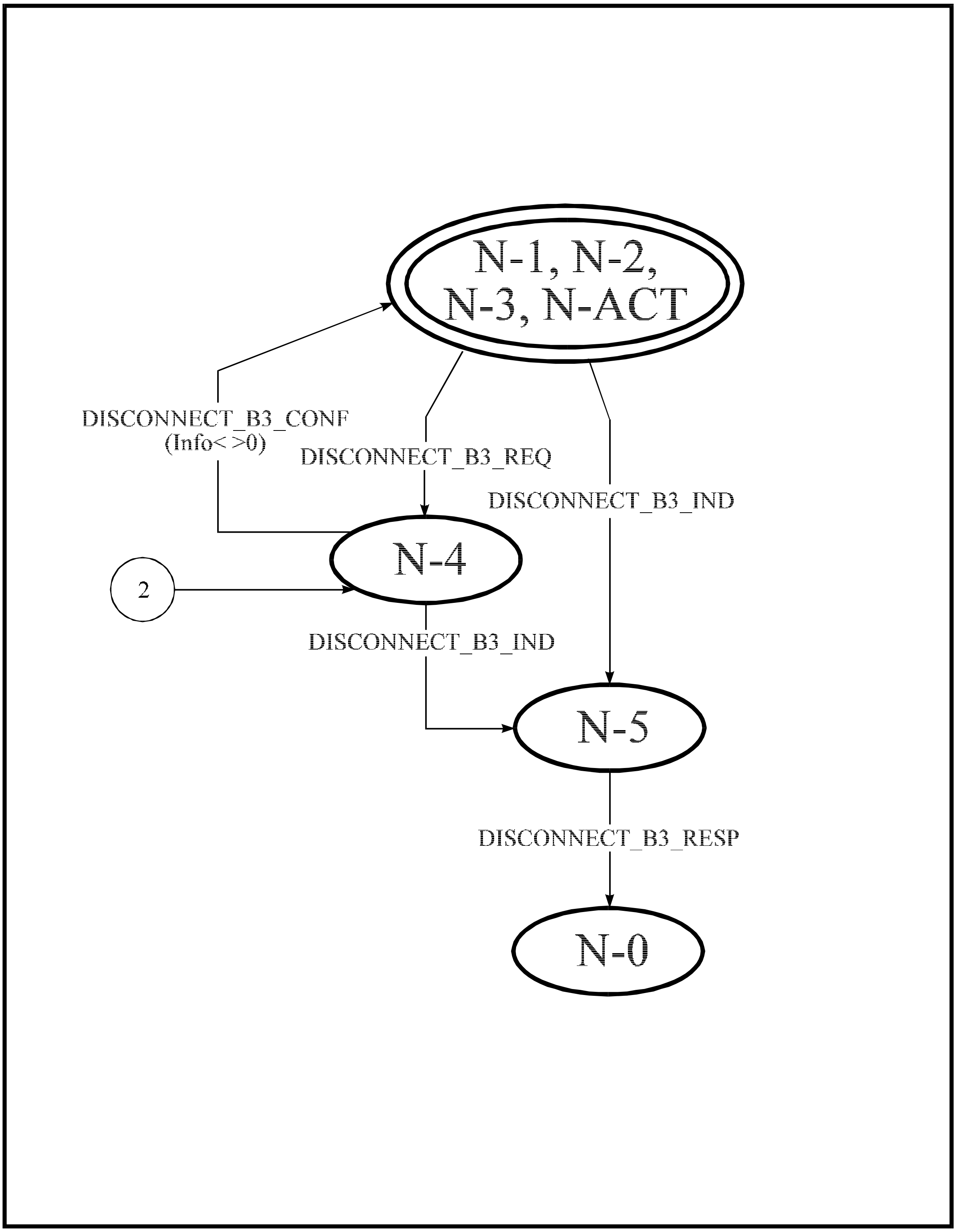
PLCI - state machine 2/2



NCCI - state machine 1/2



NCCI - state machine 2/2



8 SPECIFICATIONS FOR COMMERCIAL OPERATING SYSTEMS

8.1 MS-DOS

As MS-DOS does not provide any multitasking facilities, **COMMON-ISDN-API** is incorporated into the system as a background driver (terminate and stay resident). The interface between the application and **COMMON-ISDN-API** is implemented by way of a software interrupt. The vector used for this must be configurable both in **COMMON-ISDN-API** and in the application. The default value for the software interrupt is 241 (0xF1). If another value is to be used, it can be specified as a parameter when **COMMON-ISDN-API** is installed.

The functions described below are defined by appropriate register assignments in this software interrupt interface. The return values and parameter are normally supplied in register AX and ES:BX. Registers AX, BX, CX, DX and ES can be modified, other registers are retained. **COMMON-ISDN-API** is allowed to enable interrupts during processing of these functions.

COMMON-ISDN-API requires a maximum stack area of 512 bytes for the execution of all the functions incorporated. This stack area must be made available by the application program. During processing the software interrupt **COMMON-ISDN-API** may enable and/or disable interrupts.

The software interrupt for **COMMON-ISDN-API** is defined according to the BIOS interrupt chaining structure.

```
API     PROC    FAR             ; ISDN-API interrupt service
        JMP     SHORT doit      ; jump to start of routine
        DD      ?               ; chained interrupt
        DW      424BH           ; interrupt chaining signature
        DB      80H             ; first in chain flag
        DW      ?               ; reserved, should be 0
        DB      'CAPI'          ; COMMON-ISDN-API signature
        DB      '20'            ; Version number
doit:
```

The characters 'CAPI20' can be requested by the application to check the presence of **COMMON-ISDN-API**.

The pointer stipulated in messages DATA_B3_REQ and DATA_B3_IND is implemented as a FAR pointer under MS-DOS.

Memory layout is according to MS-DOS.

8.1.1 Message Operations

CAPI_REGISTER

Description

This is the function the application uses to report its presence to **COMMON-ISDN-API**. In doing so, the application provides **COMMON-ISDN-API** with a memory area. A FAR pointer to this memory area is transferred in registers *ES:BX*. The size of the memory area is calculated according to the following formula:

CX + (DX * SI * DI)

The size of the message buffer used to store messages is transferred to the *CX* register. Choosing too small this value will result in messages being lost.. A 'normal' application should calculate the necessary amount of memory according to following formula:

CX = 1024 + (1024 * DX)

In the *DX* register the application indicates the maximum number of logical connections opened simultaneously. An attempt to open more logical connections than stipulated here can be acknowledged with an error message from **COMMON-ISDN-API**.

In the *SI* register the application sets the maximum number of received B3 data blocks that can be reported to the application simultaneously. The number of simultaneously available B3 data blocks has a decisive effect on the throughput of B3 data in the system and should be between 2 and 7. There must be room for two B3 data blocks at least.

In the *DI* register the application sets the maximum size of the application data to be transmitted and received, that is the maximum *data length* parameter in messages **DATA_B3_REQ** and **DATA_B3_IND**. The default value for the protocol ISO 7776 (X.75) is 128 octets. **COMMON-ISDN-API** will be able to support at least up to 2048 octets, if an application sets register *DI* with corresponding values.

The application number is supplied in the *AX* register. In the event of an error, the *AX* register is returned with the value 0. The cause of the error is held in the *BX* register in this case.

CAPI_REGISTER	0x01

Parameter	Comment
AH	Version number 20 (0x14)
AL	Function code 0x01
ES:BX	FAR pointer to a memory block provided by the application. This memory area can (but need not) be used by **COMMON-ISDN-API** to manage the message queue of the application. In addition, **COMMON-ISDN-API** can (but also need not) provide the received data in this memory area.
CX	Size of message buffer
DX	Maximum number of level 3 connections
SI	Number of B3 data blocks available simultaneously
DI	Maximum size of a B3 data block

Return Value

Return	Value	Comment
AX	<> 0 0x0000	Application number (ApplID) Registration error, cause of error in BX register
BX		if AX == 0, coded as described in parameter Info class 0x10xx

Note

If the application intends to open a maximum of one layer 3 connection simultaneously and the standard protocols are used, the following register assignment is recommended:

CX = 2048, DX = 1, SI = 7, DI = 128

The resulting memory requirement is 2944 bytes.

CAPI_RELEASE

Description

The application uses this function to log off from **COMMON-ISDN-API**. The memory area indicated in the **CAPI_REGISTER** is released. The application is identified by the application number in the *DX* register. Any errors that occur are returned in register *AX*.

CAPI_RELEASE	**0x02**

Parameter	Comment
AH	Version number 20 (0x14)
AL	Function Code 0x02
DX	Application number

Return Value

Return	Value	Comment
AX	0x0000	no error
	<> 0	Registration error, coded as described in parameter Info class 0x11xx

CAPI_PUT_MESSAGE

Description

With this function the application transfers a message to **COMMON-ISDN-API**. A FAR pointer is transferred to the message in the *ES:BX* registers. The application is identified via application number in the *DX* register. Any errors that occur are returned in register *AX*.

CAPI_PUT_MESSAGE	**0x03**

Parameter	Comment
AH	Version number 20 (0x14)
AL	Function Code 0x03
ES:BX	FAR pointer to the message
DX	Application number

Return Value

Return	Value	Comment
AX	0x0000 <> 0	No error Coded as described in parameter info class 0x11xx

Note

After **CAPI_PUT_MESSAGE** the application can use the memory area of the message again. The message will not be modified by **COMMON-ISDN-API**.

CAPI_GET_MESSAGE

Description

With this function the application retrieves a message from **COMMON-ISDN-API**. The application can only retrieve those messages intended for the stipulated application number. A FAR pointer is set to the message in the *ES:BX* registers. If there is no message for the application, the function returns immediately. Register *AX* contains the corresponding error value. The application is identified via the application number in the *DX* register. Any errors that occur are returned in register *AX*.

CAPI_GET_MESSAGE	**0x04**

Parameter	Comment
AH	Version number 20 (0x14)
AL	Function Code 0x04
DX	Application number

Return Value

Return	Value	Comment
AX	0x0000 <> 0	No error Coded as described in parameter info class 0x11xx
ES:BX		FAR pointer to message, if available

Note

The message may be invalidated the next time **CAPI_GET_MESSAGE** is called.

8.1.2 Other Functions

CAPI_SET_SIGNAL

Description

The application can use this function to activate usage of the interrupt call-back function. A FAR pointer to an interrupt call-back function is specified in the *ES:BX* registers. The signalling function can be deactivated by a **CAPI_SET_SIGNAL** with register assignment *ES:BX* = 0000:0000. The application is identified via the application number in the DX register. Any errors that occurred are returned in the *AX* register.

CAPI_SET_SIGNAL	**0x05**

Parameter	Comment
AH	Version number 20 (0x14)
AL	Function Code 0x05
DX	Application number
SI:DI	Parameter passed to call-back function
ES:BX	FAR pointer to call-back function

Return Value

Return	Value	Comment
AX	0x0000	No error
	<> 0	Coded as described in parameter info class 0x11xx

Note

The call-back function is called as an interrupt by **COMMON-ISDN-API,** after

- any message is queued in application's message queue
- a notified busy condition is cleared
- a notified queue full condition is cleared

Interrupts are disabled. The call-back function must be terminated via IRET. All registers have to be preserved. At the time of calling, at least 32 bytes are available on the stack.

The call-back function will be called with interrupts disabled. **COMMON-ISDN-API** will not call this function recursively, even if the call-back function enables interrupts. Instead, the call-back function will be called again after returning to **COMMON-ISDN-API**.

The call-back function is allowed to use **COMMON-ISDN-API** operations **CAPI_PUT_MESSAGE**, **CAPI_GET_MESSAGE**, and **CAPI_SET_SIGNAL**. In that case the application must be aware that interrupts may be enabled by **COMMON-ISDN-API**.

In case of local confirmations (e.g. LISTEN_CONF) the call-back function may be activated before the operation CAPI_PUT_MESSAGE returns to the application.

Parameter DX, SI and DI will be passed to the call-back function with the same values of the corresponding parameters to **CAPI_SET_SIGNAL**.

CAPI_GET_MANUFACTURER

Description

With this function the application determines the manufacturer identification of **COMMON-ISDN-API.** In registers *ES:BX* a FAR pointer is transferred to a data area of 64 bytes. The manufacturer identification, coded as a zero terminated ASCII string, is present in this data area after the function has been executed.

CAPI_GET_MANUFACTURER	**0xF0**

Parameter	Comment
AH	Version number 20 (0x14)
AL	Function Code 0xF0
ES:BX	FAR pointer to buffer

Return Value

Return	Comment
ES:BX	buffer contains manufacturer identification with ASCII coding. The end of the identification is indicated with a 0 byte.

CAPI_GET_VERSION

Description

With this function the application determines the version of **COMMON-ISDN-API** as well as an internal revision number.

CAPI_GET_VERSION	**0xF1**

Parameter	Comment
AH	Version number 20 (0x14)
AL	Function Code 0xF1

Return Value

Return	Comment
AH	**COMMON-ISDN-API** major version: 2
AL	**COMMON-ISDN-API** minor version: 0
DH	Manufacturer specific major number
DL	Manufacturer specific minor number

CAPI_GET_SERIAL_NUMBER

Description

With this function the application determines the (optional) serial number of **COMMON-ISDN-API.** In registers *ES:BX* a FAR pointer to a data area of 8 bytes is transferred. The serial number, coded as a zero terminated ASCII string, is present in this data area in the form of a seven-digit number after the function has been executed. If no serial number is supplied, the serial number is an empty string.

CAPI_GET_SERIAL_NUMBER	**0xF2**

Parameter	Comment
AH	Version number 20 (0x14)
AL	Function Code 0xF2
ES:BX	FAR pointer to buffer

Return Value

Return	Comment
ES:BX	The (optional) serial number is read in plain text in the form of a 7-digit number. If no serial number is to be used, a 0 byte must be written at the first position in the buffer. The end of the serial number is indicated with a 0 byte.

CAPI_GET_PROFILE

Description

The application uses this function to get the capabilities from **COMMON-ISDN-API**. Registers *ES:BX* contain a FAR pointer to a data area of 64 bytes. In this buffer **COMMON-ISDN-API** copies information about implemented features, number of controllers and supported protocols. Register *CX* contains the controller number (bit 0..6) for which this information is requested.

CAPI_GET_PROFILE	**0xF3**

Parameter	Comment
AH	Version number 20 (0x14)
AL	Functional Code 0xF3
CX	controller number (if 0, only number of controllers is re- turned)
ES:BX	FAR pointer to buffer

Return Value

Return	Value	Comment
AX	0x0000 <> 0	No error Coded as described in parameter info class 0x11xx

Retrieved structure format:

Type	Description
2 octets	number of installed controllers, least significant octet first
2 octets	number of supported B-channels, least significant octet first
4 octets	Global Options (bit field): [0]: internal controller supported [1]: external equipment supported [2]: Handset supported (external equipment must be set also) [3]: DTMF supported [4].[31]: reserved

4 octets	B1 protocols support (bit field): [0]: 64 kBit/s with HDLC framing, always set. [1]: 64 kBit/s bit transparent operation with byte framing from the network [2]: V.110 asynchronous operation with start/stop byte framing [3]: V.110 synchronous operation with HDLC framing [4]: T.30 modem for fax group 3 [5]: 64 kBit/s inverted with HDLC framing. [6]: 56 kBit/s bit transparent operation with byte framing from the network [7]..[31]: reserved
4 octets	B2 protocol support (bit field): [0]: ISO 7776 (X.75 SLP), always set [1]: Transparent [2]: SDLC [3]: LAPD according Q.921 for D channel X.25 [4]: T.30 for fax group 3 [5]: Point to Point Protocol (PPP) [6]: Transparent (ignoring framing errors of B1 protocol) [7]..[31]: reserved
4 octets	B3 protocol support (bit field): [0]: Transparent, always set [1]: T.90NL with compatibility to T.70NL according to T.90 Appendix II. [2]: ISO 8208 (X.25 DTE-DTE) [3]: X.25 DCE [4]: T.30 for fax group 3 [5]..[31]: reserved
24 octets	reserved for **COMMON-ISDN-API** usage
20 octets	manufacturer specific information

Note

This function can be extended, so an application has to ignore unknown bits. **COMMON-ISDN-API** will set every reserved field to 0.

CAPI_MANUFACTURER

Description

This function is manufacturer specific.

CAPI_MANUFACTURER	0xFF

Parameter	Comment
AH	Version number 20 (0x14)
AL	Function Code 0xFF
Manufacturer specific	

Return Value

Return	Comment
Manufacturer specific	

8.2 Windows (application level)

In a PC environment with the MS-DOS extension Windows an application can access **COMMON-ISDN-API** services via a DLL (Dynamic Link Library). The interface between applications and **COMMON-ISDN-API** is realised as a function interface. An application can issue **COMMON-ISDN-API** function calls to perform **COMMON-ISDN-API** operations.

The DLL providing the function interface has to be named "CAPI20.DLL". All functions exported by this library have to be called with a FAR call according to the PASCAL calling convention. This means all parameters are pushed on the stack (first parameter is pushed first), the called function has to clear up the stack before it returns to the caller.

The functions are exported under following names and ordinal numbers:

CAPI_MANUFACTURER (reserved)	CAPI20.99
CAPI_REGISTER	CAPI20.1
CAPI_RELEASE	CAPI20.2
CAPI_PUT_MESSAGE	CAPI20.3
CAPI_GET_MESSAGE	CAPI20.4
CAPI_SET_SIGNAL	CAPI20.5
CAPI_GET_MANUFACTURER	CAPI20.6
CAPI_GET_VERSION	CAPI20.7
CAPI_GET_SERIAL_NUMBER	CAPI20.8
CAPI_GET_PROFILE	CAPI20.9
CAPI_INSTALLED	CAPI20.10

These functions can be called by an application according to the DLL conventions as imported functions. If an application calls any function of the DLL with whatever function it must ensure that there are at least 512 bytes left on the stack.

All pointers that are passed from the application program to **COMMON-ISDN-API**, or vice versa, in function calls or in messages are 16:16 segmented protected mode pointers. This especially applies to the data pointer in **DATA_B3_REQ** and **DATA_B3_IND** messages.

In the Windows 3.x environment following types are used to define the functional interface:

WORD	16 bit unsigned integer
DWORD	32 bit unsigned integer
LPVOID	16:16 (segmented) protected mode pointer to any memory location
LPVOID *	16:16 (segmented) protected mode pointer to a LPVOID
LPBYTE	16:16 (segmented) protected mode pointer to a character string
LPWORD	16:16 (segmented) protected mode pointer to a 16 bit unsigned integer value
CAPIENTRY	WORD FAR PASCAL (according to Windows DLL calling convention)

8.2.1 Message Operations

CAPI_REGISTER

Description

This is the operation the application uses to report its presence to **COMMON-ISDN-API**. By passing the four parameters MessageBufferSize, maxLogicalConnection, maxBDataBlocks and maxBDataLen the application describes its needs.

For a 'normal' application the size of the message buffer should be calculated using following formula:

MessageBufferSize = 1024 + (1024 * maxLogicalConnection)

Function call

```
CAPIENTRY CAPI_REGISTER (   WORD MessageBufferSize,
                            WORD maxLogicalConnection,
                            WORD maxBDataBlocks,
                            WORD maxBDataLen,
                            LPWORD pApplID);
```

Parameter	Comment
MessageBufferSize	Size of Message Buffer
maxLogicalConnection	Maximum number of logical connections
maxBDataBlocks	Number of data blocks available simultaneously
maxBDataLen	Maximum size of a data block
pApplID	Pointer to the location where **COMMON-ISDN-API** should place the assigned application identification number

Return Value

Return Value	Comment
0x0000	Registration successful - application identification number has been assigned
All other values	Coded as described in parameter info class 0x10xx

CAPI_RELEASE

Description

The application uses this operation to log off from **COMMON-ISDN-API**. **COMMON-ISDN-API** will release all resources that have been allocated for the application.

The application is identified by the application identification number that had been assigned in the previous CAPI_REGISTER operation.

Function call

CAPIENTRY CAPI_RELEASE (WORD ApplID);

Parameter	Comment
ApplID	Application identification number that had been assigned by call of the function CAPI_REGISTER

Return Value

Return Value	Comment
0x0000	Release of the application successful
All other values	Coded as described in parameter info class 0x11xx

CAPI_PUT_MESSAGE

Description

With this operation the application transfers a message to **COMMON-ISDN-API**. The application identifies itself with an application identification number.

Function call

CAPIENTRY CAPI_PUT_MESSAGE(	**WORD ApplID,** **LPVOID pCAPIMessage);**

Parameter	Comment
ApplID	Application identification number that had been assigned by call of the function CAPI_REGISTER
pCAPIMessage	16:16 (segmented) protected mode pointer to the message that is passed to **COMMON-ISDN-API**

Return Value

Return Value	Comment
0x0000	No error
All other values	Coded as described in parameter info class 0x11xx

Note

When the process returns from the function call the message memory area can be reused by the application.

CAPI_GET_MESSAGE

Description

With this operation the application retrieves a message from **COMMON-ISDN-API**. The application can only retrieve those messages intended for the stipulated application identification number. If there is no message waiting for retrieval, the function returns immediately with an error code.

Function call

CAPIENTRY CAPI_GET_MESSAGE (	**WORD ApplID,** **LPVOID *ppCAPIMessage);**

Parameter	Comment
ApplID	Application identification number that had been assigned by call of the function CAPI_REGISTER
ppCAPIMessage	16:16 (segmented) protected mode pointer to the memory location where **COMMON-ISDN-API** should place the 16:16 (segmented) protected mode pointer to the retrieved message

Return Value

Return Value	Comment
0x0000	Successful - Message was retrieved from **COMMON-ISDN-API**
All other values	Coded as described in parameter info class 0x11xx

Note

The received message may become invalid the next time the application issues a CAPI_GET_MESSAGE operation for the same application identification number. This especially matters in multi threaded applications where more than one thread may execute CAPI_GET_MESSAGE operations. The synchronisation between threads has to be done by the application.

8.2.2 Other Functions

CAPI_SET_SIGNAL

Description

This operation is used by the application to install a mechanism which signals the application the availability of a message or the clearing of an internal busy/queue full condition. All restrictions of interrupt context will apply to the call-back function.

Function call

```
CAPIENTRY CAPI_SET_SIGNAL (   WORD ApplID,
                              VOID (FAR PASCAL *CAPI_Callback)
                                  (WORD ApplID, DWORD Param),
                              DWORD Param
                              );
```

Parameter	Comment
ApplID	Application identification number that had been assigned by call of the function CAPI_REGISTER
CAPI_Callback	address of the call-back function. The function will be called in an interrupt context (see note). Value **0x00000000** will disable the call-back notification.
Param	additional parameter of call-back function

Return Value

Return Value	Comment
0x0000	No error
All other values	Coded as described in parameter info class 0x11xx

Note

The notification will take place, after

- any message is queued in application's message queue
- a notified busy condition is cleared
- a notified queue full condition is cleared

In case of local confirmations (e.g. LISTEN_CONF) the notification may be activated before the operation CAPI_PUT_MESSAGE returns to the application.

The call-back function will be called using following conventions:

```
VOID FAR PASCAL CAPI_Callback (
        WORD ApplID,
        DWORD Param
);
```

Data segment register DS is undefined (use *MakeProcInstance()* or *_setds*). A stack of at least 512 bytes is set up by **COMMON-ISDN-API.**

The call-back function may be called at interrupt context (i.e., every data and code accessed by the call-back function has to be prevented from being paged out by Windows' VMM, e.g. by using *fixed* segments in its own DLL and/or by applying *GlobalPageLock()* to used selectors).

PostMessage() and *PostAppMessage()* are the only windows API functions which can be called.

CAPI_PUT_MESSAGE, CAPI_GET_MESSAGE and CAPI_SET_SIGNAL are the only **COMMON-ISDN-API** functions which can be called.

The call-back function will not be re-entered by **COMMON-ISDN-API**. Instead it will be called again after returning, if a new event has occurred during processing.

CAPI_GET_MANUFACTURER

Description

With this operation the application determines the manufacturer identification of **COMMON-ISDN-API** (DLL). SzBuffer on call is a 16:16 (segmented) protected mode pointer to a buffer of 64 bytes. **COMMON-ISDN-API** copies the identification string, coded as a zero terminated ASCII string, to this buffer.

Function call

CAPIENTRY CAPI_GET_MANUFACTURER (LPBYTE SzBuffer);

Parameter	Comment
SzBuffer	16:16 (segmented) protected mode pointer to a buffer of 64 bytes

Return Value

Return Value	Comment
0x0000	No error

CAPI_GET_VERSION

Description

With this function the application determines the version of **COMMON-ISDN-API** as well as an internal revision number.

Function call

CAPIENTRY CAPI_GET_VERSION (	**LPWORD pCAPIMajor,** **LPWORD pCAPIMinor,** **LPWORD pManufacturerMajor,** **LPWORD pManufacturerMinor);**

Parameter	Comment
pCAPIMajor	16:16 (segmented) protected mode pointer to a WORD receiving **COMMON-ISDN-API** major version number: 2
pCAPIMinor	16:16 (segmented) protected mode pointer to a WORD receiving **COMMON-ISDN-API** minor version number: 0
pManufacturerMajor	16:16 (segmented) protected mode pointer to a WORD receiving manufacturer specific major number
pManufacturerMinor	16:16 (segmented) protected mode pointer to a WORD receiving manufacturer specific minor number

Return Value

Return	Comment
0x0000	No error, version numbers are copied

CAPI_GET_SERIAL_NUMBER

Description

With this operation the application determines the (optional) serial number of **COMMON-ISDN-API**. SzBuffer on call is a 16:16 (segmented) protected mode pointer to a string buffer of 8 bytes. **COMMON-ISDN-API** copies the serial number string to this buffer. The serial number, coded as a zero terminated ASCII string, represents seven digit number after the function has returned.

Function call

```
CAPIENTRY CAPI_GET_SERIAL_NUMBER (LPBYTE SzBuffer);
```

Parameter	Comment
SzBuffer	16:16 (segmented) protected mode pointer to a buffer of 8 bytes

Return Value

Return	Comment
0x0000	No error SzBuffer contains the serial number in plain text in the form of a 7-digit number. If no serial number is provided by the manufacturer, an empty string is returned.

CAPI_GET_PROFILE

Description

The application uses this function to get the capabilities from **COMMON-ISDN-API**. SzBuffer on call is a 16:16 (segmented) protected mode pointer to a buffer of 64 bytes. In this buffer **COMMON-ISDN-API** copies information about implemented features, number of controllers and supported protocols. *CtrlNr* contains the controller number (bit 0..6), for which this information is requested.

```
CAPIENTRY CAPI_GET_PROFILE (    LPBYTE SzBuffer,
                                WORD CtrlNr
                                );
```

Parameter	Comment
SzBuffer	16:16 (segmented) protected mode pointer to a buffer of 64 bytes
CtrlNr	Number of Controller. If 0, only number of installed controllers is given to the application.

Return Value

Return	Value	Comment
AX	0x0000 <> 0	No error Coded as described in parameter info class 0x11xx

Retrieved structure format:

Type	Description
WORD	number of installed controllers, least significant octet first
WORD	number of supported B-channels, least significant octet first
DWORD	Global Options (bit field): [0]: internal controller supported [1]: external equipment supported [2]: Handset supported (external equipment must be set also) [3]: DTMF supported [4].[31]: reserved
DWORD	B1 protocols support (bit field): [0]: 64 kBit/s with HDLC framing, always set. [1]: 64 kBit/s bit transparent operation with byte framing from the network [2]: V.110 asynchronous operation with start/stop byte framing [3]: V.110 synchronous operation with HDLC framing [4]: T.30 modem for fax group 3 [5]: 64 kBit/s inverted with HDLC framing. [6]: 56 kBit/s bit transparent operation with byte framing from the network [7]..[31]: reserved

DWORD	B2 protocol support (bit field): [0]: ISO 7776 (X.75 SLP), always set [1]: Transparent [2]: SDLC [3]: LAPD according Q.921 for D channel X.25 [4]: T.30 for fax group 3 [5]: Point to Point Protocol (PPP) [6]: Transparent (ignoring framing errors of B1 protocol) [7]..[31]: reserved
DWORD	B3 protocol support (bit field): [0]: Transparent, always set [1]: T.90NL with compatibility to T.70NL according to T.90 Appendix II. [2]: ISO 8208 (X.25 DTE-DTE) [3]: X.25 DCE [4]: T.30 for fax group 3 [5]..[31]: reserved
6 DWORDs	reserved for **COMMON-ISDN-API** usage
5 DWORDs	manufacturer specific information

Note

This function can be extended, so an application has to ignore unknown bits. **COMMON-ISDN-API** will set every reserved field to 0.

CAPI_INSTALLED

Description

This function can be used by an application to determine if the ISDN hardware and necessary drivers are installed.

Function call

CAPIENTRY CAPI_INSTALLED (void)

Return Value

Return	Comment
0x0000	**COMMON-ISDN-API** is installed
any other value	Coded as described in parameter info class 0x10xx

8.3 OS/2 (application level)

In a PC environment with operating system OS/2 Version 2.x an application program can access **COMMON-ISDN-API** services via a DLL (Dynamic Link Library). The interface between applications and **COMMON-ISDN-API** is realised as a function interface. An application can issue **COMMON-ISDN-API** function calls to perform **COMMON-ISDN-API** operations.

The DLL providing the function interface has to be named "CAPI20.DLL". It is a 32 bit DLL exporting 32 bit functions with System-Call-Convention. This means all parameters are pushed on the stack, the calling process has to clear up the stack after it returns from the function call.

The functions are exported under following names and ordinal numbers:

CAPI_MANUFACTURER (reserved)	CAPI20.99
CAPI_REGISTER	CAPI20.1
CAPI_RELEASE	CAPI20.2
CAPI_PUT_MESSAGE	CAPI20.3
CAPI_GET_MESSAGE	CAPI20.4
CAPI_SET_SIGNAL	CAPI20.5
CAPI_GET_MANUFACTURER	CAPI20.6
CAPI_GET_VERSION	CAPI20.7
CAPI_GET_SERIAL_NUMBER	CAPI20.8
CAPI_GET_PROFILE	CAPI20.9
CAPI_INSTALLED	CAPI20.10

These functions can be called by an application according to the DLL conventions as imported functions. If an application calls the DLL it has to ensure that there are at least 512 bytes left on the stack.

All pointers that are passed from the application program to **COMMON-ISDN-API**, or vice versa, in function calls or in messages are 0:32 flat pointers. This especially applies to the data pointer in **DATA_B3_REQ** and **DATA_B3_IND** messages. The referenced data shall not cross a 64 kByte boundary in the flat address space because the DLL may convert the passed flat pointer to a 16:16 bit segmented pointer.

In the OS/2 environment following types are used to define the functional interface:

word	16 bit unsigned integer
dword	32 bit unsigned integer
void*	0:32 flat pointer to any memory location
void**	0:32 flat pointer to a void *
char*	0:32 flat pointer to a character string
dword*	0:32 flat pointer to a 32 bit unsigned integer value

8.3.1 Message Operations

CAPI_REGISTER

Description

This is the operation the application uses to report its presence to **COMMON-ISDN-API**. By passing the four parameters messageBufferSize, maxLogicalConnection, maxBDataBlocks and maxBDataLen the application describes its needs.

For a 'normal' application the size of the message buffer should be calculated using the following formula:

MessageBufferSize = 1024 + (1024 * maxLogicalConnection)

Function call

```
dword FAR PASCAL CAPI_REGISTER (    dword messageBufferSize,
                                    dword maxLogicalConnection,
                                    dword maxBDataBlocks,
                                    dword maxBDataLen,
                                    dword* pApplID);
```

Parameter	Comment
messageBufferSize	Size of Message Buffer
maxLogicalConnection	Maximum number of logical connections
maxBDataBlocks	Number of data blocks available simultaneously
maxBDataLen	Maximum size of a data block
pApplID	Pointer to the location where **COMMON-ISDN-API** should place the assigned application identification number

Return Value

Return Value	Comment
0x0000	Registration successful - application identification number has been assigned
All other values	Coded as described in parameter info class 0x10xx

CAPI_RELEASE

Description

The application uses this operation to log off from **COMMON-ISDN-API**. **COMMON-ISDN-API** will release all resources that have been allocated.

The application is identified by the application identification number that had been assigned in the previous CAPI_REGISTER operation.

Function call

```
dword FAR PASCAL CAPI_RELEASE (dword ApplID);
```

Parameter	Comment
ApplID	Application identification number that had been assigned by call of the function CAPI_REGISTER

Return Value

Return Value	Comment
0x0000	Release of the application successful
All other values	Coded as described in parameter info class 0x11xx

CAPI_PUT_MESSAGE

Description

With this operation the application transfers a message to **COMMON-ISDN-API**. The application identifies itself with an application identification number. The message memory area must not cross a 64 kByte boundary (e.g. use *tiled* memory) in the flat address space because the DLL may convert the passed flat pointer to a 16:16 bit segmented pointer. The same applies to B3 data blocks that are passed within DATA_B3_REQ messages.

Function call

```
dword FAR PASCAL CAPI_PUT_MESSAGE (     dword ApplID,
                                        void* pCAPIMessage);
```

Parameter	Comment
ApplID	Application identification number that had been assigned by call of the function CAPI_REGISTER
pCAPIMessage	0:32 (flat) pointer to the message that is passed to **COMMON-ISDN-API**

Return Value

Return Value	Comment
0x0000	No error
All other values	Coded as described in parameter info class 0x11xx

Note

When the process returns from the function call the message memory area can be reused by the application.

CAPI_GET_MESSAGE

Description

With this operation the application retrieves a message from **COMMON-ISDN-API**. The application can only retrieve those messages intended for the stipulated application identification number. If there is no message waiting for retrieval, the function returns immediately with an error code.

Function call

dword FAR PASCAL CAPI_GET_MESSAGE (	**dword ApplID,** **void** ppCAPIMessage);**

Parameter	Comment
ApplID	Application identification number that had been assigned by call of the function CAPI_REGISTER
ppCAPIMessage	0:32 (flat) pointer to the memory location where **COMMON-ISDN-API** should place the 0:32 (flat) pointer to the retrieved message

Return Value

Return Value	Comment
0x0000	Successful - Message was retrieved from **COMMON-ISDN-API**
All other values	Coded as described in parameter info class 0x11xx

Note

The received message may become invalid the next time the application issues a **CAPI_GET_MESSAGE** operation for the same application identification number. This especially matters in multi threaded applications where more than one thread may execute **CAPI_GET_MESSAGE** operations. The synchronisation between threads has to be done by the application.

8.3.2 Other Functions

CAPI_SET_SIGNAL

Description

This operation is used by the application to install a mechanism which signals the application the availability of a message.

In OS/2 2.x this is done best by using a fast 32 bit system event semaphore. The application has to create the used semaphore by calling the *DosCreateEventSem()* function which is part of the OS/2 system application program interface. This routine provides a semaphore handle which is passed as a parameter in the CAPI_SET_SIGNAL call.

In that case each time **COMMON-ISDN-API** places a message in the application's message queue the specified semaphore is "posted" increasing a post-count value that is associated to the semaphore. To do so **COMMON-ISDN-API** executes the *DosPostEventSem()* function of the OS/2 system API.

The application thread may wait until the post-count of the semaphore is larger than 0 using the *DosWaitEventSem()* OS/2 system call. It can determine the current post count and simultaneously reset the post count executing the *DosResetEventSem()* OS/2 system API call.

By issuing this function call with a semaphore handle of 0 the signalling mechanism is deactivated.

Function call

dword FAR PASCAL CAPI_SET_SIGNAL (	**dword ApplID,** **dword hEventSem);**

Parameter	Comment
ApplID	Application identification number that had been assigned by call of the function CAPI_REGISTER
hEventSem	Event Semaphore handle assigned by operating system

Return Value

Return Value	Comment
0x0000	No error
All other values	Coded as described in parameter info class 0x11xx

CAPI_GET_MANUFACTURER

Description

With this operation the application determines the manufacturer identification of **COMMON-ISDN-API** (DLL). SzBuffer on call is a 0:32 (flat) pointer to a buffer of 64 bytes. **COMMON-ISDN-API** copies the identification string, coded as a zero terminated ASCII string, to this buffer.

Function call

```
void FAR PASCAL CAPI_GET_MANUFACTURER (char* SzBuffer);
```

Parameter	Comment
SzBuffer	0:32 (flat) pointer to a buffer of 64 bytes

CAPI_GET_VERSION

Description

With this function the application determines the version of **COMMON-ISDN-API** as well as an internal revision number.

Function call

dword FAR PASCAL CAPI_GET_VERSION (	**dword* pCAPIMajor,** **dword* pCAPIMinor,** **dword* pManufacturerMajor,** **dword* pManufacturerMinor);**

Parameter	Comment
pCAPIMajor	0:32 (flat) protected mode pointer to a dword receiving **COMMON-ISDN-API** major version number: 2
pCAPIMinor	0:32 (flat) protected mode pointer to a dword receiving **COMMON-ISDN-API** minor version number: 0
pManufacturerMajor	0:32 (flat) protected mode pointer to a dword receiving manufacturer specific major number
pManufacturerMinor	0:32 (flat) protected mode pointer to a dword receiving manufacturer specific minor number

Return Value

Return	Comment
0x0000	No error, version numbers are copied.

CAPI_GET_SERIAL_NUMBER

Description

With this operation the application determines the (optional) serial number of **COMMON-ISDN-API**. SzBuffer on call is a 0:32 (segmented) protected mode pointer to a buffer of 8 bytes. **COMMON-ISDN-API** copies the serial number string to this buffer. The serial number, coded as a zero terminated ASCII string, represents seven digit number after the function has returned.

Function call

dword FAR PASCAL CAPI_GET_SERIAL_NUMBER (char* SzBuffer);

Parameter	Comment
SzBuffer	0:32 (flat) pointer to a buffer of 8 bytes

Return Value

Return	Comment
0x0000	No error SzBuffer contains the serial number in plain text in the form of a 7-digit number. If no serial number is provided by the manufacturer, an empty string is returned.

CAPI_GET_PROFILE

Description

The application uses this function to get the capabilities from **COMMON-ISDN-API**. SzBuffer on call is a 0:32 (flat) protected mode pointer to a buffer of 64 bytes. In this buffer **COMMON-ISDN-API** copies information about implemented features, number of controllers and supported protocols. *CtrlNr* contains the controller number (bit 0..6), for which this information is requested.

dword FAR PASCAL CAPI_GET_PROFILE (	**LPBYTE SzBuffer,** **WORD CtrlNr** **);**

Parameter	Comment
SzBuffer	0:32 (flat) protected mode pointer to a buffer of 64 bytes
CtrlNr	Number of Controller. If 0, only number of installed controllers is given to the application.

Return Value

Return	Value	Comment
AX	0x0000 <> 0	No error Coded as described in parameter info class 0x11xx

Retrieved structure format:

Type	Description
WORD	number of installed controllers, least significant octet first
WORD	number of supported B-channels, least significant octet first
DWORD	Global Options (bit field): [0]: internal controller supported [1]: external equipment supported [2]: Handset supported (external equipment must be set also) [3]: DTMF supported [4].[31]: reserved
DWORD	B1 protocols support (bit field): [0]: 64 kBit/s with HDLC framing, always set. [1]: 64 kBit/s bit transparent operation with byte framing from the network [2]: V.110 asynchronous operation with start/stop byte framing [3]: V.110 synchronous operation with HDLC framing [4]: T.30 modem for fax group 3 [5]: 64 kBit/s inverted with HDLC framing. [6]: 56 kBit/s bit transparent operation with byte framing from the network [7]..[31]: reserved

DWORD	B2 protocol support (bit field): [0]: ISO 7776 (X.75 SLP), always set [1]: Transparent [2]: SDLC [3]: LAPD according Q.921 for D channel X.25 [4]: T.30 for fax group 3 [5]: Point to Point Protocol (PPP) [6]: Transparent (ignoring framing errors of B1 protocol) [7]..[31]: reserved
DWORD	B3 protocol support (bit field): [0]: Transparent, always set [1]: T.90NL with compatibility to T.70NL according to T.90 Appendix II. [2]: ISO 8208 (X.25 DTE-DTE) [3]: X.25 DCE [4]: T.30 for fax group 3 [5]..[31]: reserved
6 DWORDs	reserved for **COMMON-ISDN-API** usage
5 DWORDs	manufacturer specific information

Note

This function can be extended, so an application has to ignore unknown bits. **COMMON-ISDN-API** will set every reserved field to 0.

CAPI_INSTALLED

Description

This function can be ued by an application to determine if the ISDN hardware and necessary drivers are installed.

Function call

```
dword FAR PASCAL CAPI_INSTALLED (void)
```

Return Value

Return	Comment
0x0000	**COMMON-ISDN-API** is installed
Any other value	Coded as described in parameter info class 0x11xx

8.4 OS/2 (device driver level)

In a PC environment with operating system OS/2 Version 2.x there may exist **COMMON-ISDN-API** applications in form of OS/2 physical device drivers (PDD). Those applications are referred as "application PDDs" in the following sections. This specification describes the interface of an OS/2 2.x physical device driver offering **COMMON-ISDN-API** services to other device drivers. **COMMON-ISDN-API** PDD is called "CAPI PDD" in the following sections.

Physical Device Drivers under OS/2 2.x are 16:16 segment modules, thus all functions in this specification are 16 bit functions, all pointers are 16:16 segmented.

In this chapter following data types are used to define the interface:

word	16 bit unsigned integer
dword	32 bit unsigned integer
void*	16:16 (segmented) pointer to any memory location
void**	16:16 (segmented) pointer to a void*
char*	16:16 (segmented) pointer to a character string
word*	16:16 (segmented) pointer to a word

The CAPI PDD offers its services to application PDDs via the Inter Device Driver Interface. An application PDD issues an inter device driver call (IDC) to execute CAPI operations.

The CAPI PDD name which is contained in its device driver header has to be "CAPI20 " (blank extended to 8 characters). The CAPI PDD header must contain the offset to its inter device driver call entry point. The IDC bit of the Device Attribute Field in the device driver header has to be set to 1.

An application PDD gains access to the CAPI PDD by issuing an *AttachDD* device help call. This call returns the protected mode IDC entry point as a 16:16 segmented pointer and the data segment of the CAPI PDD. Before calling the IDC entry point of the CAPI PDD the application PDD has to set-up the data segment register DS appropriately.

This is the prototype of the CAPI PDD IDC function:

```
word CAPI20_IDC (word funcCode, void *funcPara);
```

The function is called with C calling convention, thus the calling application PDD has to clear up the stack. When the application PDD calls the IDC function there has to be at least a space of 512 bytes left on the stack. The parameter funcCode selects the CAPI operation to take place, the parameter funcPara contains a 16:16 segmented pointer to the CAPI operation specific parameters. The structure of these parameters is defined in the following sections. The function returns an error code which is 0 if no error occurred. Which CAPI operations may cause which error codes is also defined in the following sections.

8.4.1 Message Operations

CAPI_REGISTER

Description

This is the operation the application PDD uses to report its presence to **COMMON-ISDN-API**. By passing the four parameters messageBufferSize, maxLogicalConnection, maxBDataBlocks and maxBDataLen the application PDD describes its needs. By use of the parameter Buffer the application PDD passes a memory area to **COMMON-ISDN-API**. **COMMON-ISDN-API** uses this memory area to store messages and data blocks destined to the application PDD. The passed memory has to be either fixed or locked. **COMMON-ISDN-API** does not need to verify if this storage really exists.

The size of the memory area is calculated according to the following formula:

MessageBufferSize + (maxLogicalConnection * maxBDataBlocks * maxBDataLen)

Choosing too small the value will result in messages being lost. The size of the message buffer should be calculated for a 'normal' application PDD according to following formula:

MessageBufferSize = 1024 + (1024 * maxLogicalConnection)

CAPI_REGISTER	**0x01**

Structure of command specific parameters:

Parameter	Type	Comment
Buffer	void*	16:16 (segmented) pointer to a memory region provided by the application PDD. **COMMON-ISDN-API** uses this memory area to store messages and data blocks destined for the application PDD.
messageBufferSize	word	Size of Message Buffer
maxLogicalConnection	word	Maximum number of logical connections
maxBDataBlocks	word	Number of data blocks available simultaneously
maxBDataLen	word	Maximum size of a data block
pApplID	word*	16:16 (segmented) pointer to the location where **COMMON-ISDN-API** should place the assigned application identification number

Return Value

Return Value	Comment
0x0000	Registration successful - application identification number has been assigned
All other values	Coded as described in parameter info class 0x10xx

CAPI_RELEASE

Description

The application PDD uses this operation to log off from **COMMON-ISDN-API**.. **COMMON-ISDN-API** will release all resources that have been allocated for the application.

The application PDD is identified by the application identification number that had been assigned in the previous CAPI_REGISTER operation.

CAPI_RELEASE	**0x02**

Structure of command specific parameters:

Parameter	Type	Comment
ApplID	word	Application identification number that had been assigned by call of the function CAPI_REGISTER

Return Value

Return Value	Comment
0x0000	Release of the application successful
All other values	Coded as described in parameter 0x11xx

CAPI_PUT_MESSAGE

Description

With this operation the application PDD transfers a message to **COMMON-ISDN-API**. The application identifies itself with an application identification number. The pointer passed to **COMMON-ISDN-API** is a 16:16 segmented pointer. The pointer in a DATA_B3_REQ message also is 16:16 segmented. The memory area of the message and the data block have to be either fixed or locked.

CAPI_PUT_MESSAGE	**0x03**

Structure of command specific parameters:

Parameter	Type	Comment
ApplID	word	Application identification number that had been assigned by call of the function CAPI_REGISTER
pCAPIMessage	void*	16:16 segmented pointer to the message that is passed to **COMMON-ISDN-API**

Return Value

Return Value	Comment
0x0000	No error
All other values	Coded as described in parameter 0x11xx

Note

When the process returns from the function call the message memory area can be reused by the application.

CAPI_GET_MESSAGE

Description

With this operation the application PDD retrieves a message from **COMMON-ISDN-API**. The application PDD can only retrieve those messages intended for the stipulated application identification number. If there is no message waiting for retrieval, the function returns immediately with an error.

CAPI_GET_MESSAGE	**0x04**

Structure of command specific parameters:

Parameter	Type	Comment
ApplID	word	Application identification number that had been assigned by call of the function CAPI_REGISTER
ppCAPIMessage	void**	16:16 segmented pointer to the memory location where **COMMON-ISDN-API** should place the 16:16 segmented pointer to the retrieved message

Return Value

Return Value	Comment
0x0000	Successful - Message was retrieved from **COMMON-ISDN-API**
All other values	Coded as described in parameter info class 0x11xx

Note

The received message may become invalid the next time the application issues a **CAPI_GET_MESSAGE** operation for the same application identification number.

8.4.2 Other Functions

CAPI_SET_SIGNAL

Description

This operation is used by the application PDD to install a mechanism which signals the application PDD the availability of a message.

A call back mechanism is used between **COMMON-ISDN-API** and an application PDD. By calling the IDC function with CAPI_SET_SIGNAL function code the application PDD passes a 16:16 (segmented) pointer to a call back function to **COMMON-ISDN-API**.

CAPI_SET_SIGNAL	**0x05**

Structure of command specific parameters:

Parameter	Type	Comment
ApplID	word	Application identification number that had been assigned by call of the function CAPI_REGISTER
signFunc	void*	16:16 segmented pointer to the call-back function

Return Value

Return Value	Comment
0x0000	No error
All other values	Coded as described in parameter info class 0x11xx

Note

The call-back function is called by **COMMON-ISDN-API,** after

- any message is queued in application's message queue
- a notified busy condition changed
- a notified queue full condition changed

Interrupts are disabled. The call-back function must be terminated via RETF. All registers have to be preserved. At the time of calling, at least 32 bytes are available on the stack.

The call-back function will be called with interrupts disabled. **COMMON-ISDN-API** will not call this function recursively, even if the call-back function enables interrupts. Instead the call-back function will be called again after returning to **COMMON-ISDN-API**.

The call-back function is allowed to use **COMMON-ISDN-API** operations **CAPI_PUT_MESSAGE**, **CAPI_GET_MESSAGE**, and **CAPI_SET_SIGNAL**. In that case the call-back function must be aware that interrupts may be enabled by **COMMON-ISDN-API**.

In case of local confirmations (e.g. LISTEN_CONF) the call-back function may be activated before the operation CAPI_PUT_MESSAGE returns to the application.

CAPI_GET_MANUFACTURER

Description

With this operation the application determines the manufacturer identification of **COMMON-ISDN-API** (DLL). SzBuffer on call is a 16:16 (segmented) pointer to a buffer of 64 bytes. **COMMON-ISDN-API** copies the identification string, coded as a zero terminated ASCII string, to this buffer.

Function call

CAPI_GET_MANUFACTURER	**0x06**

Structure of command specific parameters:

Parameter	Type	Comment
SzBuffer	char*	16:16 (segmented) pointer to a buffer of 64 bytes

Return Value

Return	Comment
0x0000	No error
All other values	Coded as described in parameter info class 0x11xx

CAPI_GET_VERSION

Description

With this function the application determines the version of **COMMON-ISDN-API** as well as an internal revision number.

Function call

CAPI_GET_VERSION	**0x07**

Structure of command specific parameters:

Parameter	Type	Comment
pCAPIMajor	word*	16:16 (segmented) protected mode pointer to a word receiving **COMMON-ISDN-API** major version number: 2
pCAPIMinor	word*	16:16 (segmented) protected mode pointer to a word receiving **COMMON-ISDN-API** minor version number: 0
pManufacturerMajor	word*	16:16 (segmented) protected mode pointer to a word receiving manufacturer specific major number
pManufacturerMinor	word*	16:16 (segmented) protected mode pointer to a word receiving manufacturer specific minor number

Return Value

Return	Comment
0x0000	No error, version numbers are copied
All other values	Coded as described in parameter info class 0x11xx

CAPI_GET_SERIAL_NUMBER

Description

With this operation the application determines the (optional) serial number of **COMMON-ISDN-API**. SzBuffer on call is a 16:16 (segmented) protected mode pointer to a buffer of 8 bytes. **COMMON-ISDN-API** copies the serial number string to this buffer. The serial number, coded as a zero terminated ASCII string, represents seven digit number after the function has returned.

Function call

CAPI_GET_SERIAL_NUMBER	**0x08**

Structure of command specific parameters:

Parameter	Type	Comment
SzBuffer	char*	16:16 (segmented) pointer to a buffer of 8 bytes

Return Value

Return	Comment
0x0000	No error SzBuffer contains the serial number in plain text in the form of a 7-digit number. If no serial number is provided by the manufacturer, an empty string is returned.
All other values	Coded as described in parameter info class 0x11xx

CAPI_GET_PROFILE

Description

The application uses this function to get the capabilities from **COMMON-ISDN-API**. SzBuffer on call is a 16:16 (segmented) protected mode pointer to a buffer of 64 bytes. In this buffer **COMMON-ISDN-API** copies information about implemented features, number of controllers and supported protocols. *CtrlNr* contains the controller number (bit 0..6), for which this information is requested.

CAPI_GET_PROFILE	**0x09**

Structure of command specific parameters:

Parameter	Type	Comment
SzBuffer	void*	16:16 (segmented) protected mode pointer to a buffer of 64 bytes
CtrlNr	word	Number of Controller. If 0, only number of installed controllers is given to the application.

Return Value

Return	Comment
0x0000	No error
All other values	Coded as described in parameter info class 0x11xx

Retrieved structure format:

Type	Description
word	number of installed controllers, least significant octet first
word	number of supported B-channels, least significant octet first
dword	Global Options (bit field): [0]: internal controller supported [1]: external equipment supported [2]: Handset supported (external equipment must be set also) [3]: DTMF supported [4].[31]: reserved

dword	B1 protocols support (bit field): [0]: 64 kBit/s with HDLC framing, always set. [1]: 64 kBit/s bit transparent operation with byte framing from the network [2]: V.110 asynchronous operation with start/stop byte framing [3]: V.110 synchronous operation with HDLC framing [4]: T.30 modem for fax group 3 [5]: 64 kBit/s inverted with HDLC framing. [6]: 56 kBit/s bit transparent operation with byte framing from the network [7]..[31]: reserved
dword	B2 protocol support (bit field): [0]: ISO 7776 (X.75 SLP), always set [1]: Transparent [2]: SDLC [3]: LAPD according Q.921 for D channel X.25 [4]: T.30 for fax group 3 [5]: Point to Point Protocol (PPP) [6]: Transparent (ignoring framing errors of B1 protocol) [7]..[31]: reserved
dword	B3 protocol support (bit field): [0]: Transparent, always set [1]: T.90NL with compatibility to T.70NL according to T.90 Appendix II. [2]: ISO 8208 (X.25 DTE-DTE) [3]: X.25 DCE [4]: T.30 for fax group 3 [5]..[31]: reserved
6 dwords	reserved for **COMMON-ISDN-API** usage
5 dwords	manufacturer specific information

Note

This function can be extended, so an application has to ignore unknown bits. **COMMON-ISDN-API** will set every reserved field to 0.

8.5 UNIX

COMMON-ISDN-API is incorporated in the UNIX environment as a kernel driver using streams facilities. Communication between such kernel drivers and applications are typically based on system calls **open**, **ioctl**, **putmsg**, **getmsg**, and **close**. To register at a device driver, an application opens a stream (*open()*), to deregister the system call *close()* is used. Data transfer from and to the driver is achieved by the calls *putmsg()* and *getmsg()*. Additional information exchange is done with the *ioctl()* system call.

COMMON-ISDN-API uses this standardised driver access. Therefore the following specification does not define a complete functional interface (which will not be accepted by UNIX applications, which always are - and have to be - file I/O oriented). Instead **COMMON-ISDN-API** system call level interface will be introduced, which every UNIX like application can use to exchange **COMMON-ISDN-API** messages and associated data. Of course it is possible to offer a functional interface (e.g. according to chapter 8.2), but that would not be the appropriate solution for an application interface for communication applications running under UNIX. Nevertheless the following specification will offer the complete functionality of **COMMON-ISDN-API** access operations used in other operating systems.

COMMON-ISDN-API's device name is **/dev/capi20**. To allow multiple access of different UNIX processes, the device is realised as a clone streams device.

An application (in terms of **COMMON-ISDN-API**) can register at **COMMON-ISDN-API** (CAPI_REGISTER) by opening the device /dev/capi20 and issuing the relevant parameters via the system call *ioctl()* to the opened device. Note that the result of this operation is a file handle, not an application ID. So in UNIX environment the application ID included in **COMMON-ISDN-API** messages will not be used to identify CAPI applications. The only valid handle between the **COMMON-ISDN-API** kernel driver and the application based on a system call level interface is a UNIX file handle. To release from **COMMON-ISDN-API** (CAPI_RELEASE), an application just has to close the opened device. **COMMON-ISDN-API** operations CAPI_PUT_MESSAGE and CAPI_GET_MESSAGE are achieved by system calls *putmsg()* and *getmsg()*. The functionality of CAPI_SET_SIGNAL need not be offered by **COMMON-ISDN-API**; instead the UNIX signalling and/or waiting mechanism based on file descriptors can be used by applications. This includes the multiple wait on different file descriptors (*poll()*); a functionality which is not offered by **COMMON-ISDN-API** based on other operating systems. Every other **COMMON-ISDN-API** operation is realised by the system call *ioctl()* with appropriate parameters.

All messages are passed transparently through the UNIX driver interface.

To define the system call level interface in the UNIX envirmonment, following data types imply following size:

ushort	16 bit unsigned integer
unsigned	32 bit unsigned integer

8.5.1 Message Operations

CAPI_REGISTER

Description

This is the operation the application uses to report its presence to **COMMON-ISDN-API**. By passing the three parameters maxLogicalConnection, maxBDataBlocks and maxBDataLen the application describes its needs for the connections it is going to accept or it will try to establish itself.

CAPI_REGISTER	**ioctl(): 0x01**

Implementation

The following code fragment depicts the UNIX implementation of **COMMON-ISDN-API** register functionality:

```
#include <sys/fcntl.h>                /* open() parameters */
#include <sys/stropts.h>              /* streams ioctl() constants */
#include <sys/socket.h>               /* streams ioctl() macros */
...
struct capi_register_params {
    unsigned        level3cnt;
    unsigned        datablkcnt;
    unsigned        datablklen;
} rp;
int fd;
struct strioctl strioctl;

/* open device */
fd = open("/dev/capi20", O_RDWR, 0);

 /* set register parameters */
rp.level3cnt = No. of simultaneous user data connections
rp.datablkcnt = No. of buffered data messages
rp.datablklen = Size of buffered data messages

 /* perform CAPI_REGISTER */
strioctl.ic_cmd = ( 'C' << 8) | 0x01;     /* CAPI_REGISTER */
strioctl.ic_timout = 0;
strioctl.ic_dp = (void *)(&rp);
strioctl.ic_len = sizeof(struct capi_register_params);
ioctl(fd, I_STR, &strioctl);
```

For simplicity, no error checking is shown in the example.

CAPI_RELEASE

Description

The application uses this operation to log off from **COMMON-ISDN-API**. This way the application signals **COMMON-ISDN-API** that all resources that have been allocated by **COMMON-ISDN-API** for the application can be released again.

The application is identified by the application identification number that had been assigned in the previous CAPI_REGISTER operation.

CAPI_RELEASE	**close()**

Implementation

To release a connection between an application and **COMMON-ISDN-API** driver, the system call *close()* is used. All related resources are released.

CAPI_PUT_MESSAGE

Description

With this operation the application transfers a message to **COMMON-ISDN-API**. The application identifies itself with an application identification number.

CAPI_PUT_MESSAGE	**putmsg()**

Implementation

To transfer a message from an application to **COMMON-ISDN-API** driver and the controller behind, the system call *putmsg()* is used.

The application puts **COMMON-ISDN-API** message into the ctl part of the *putmsg()* call. Parameter *data* and *data length* of message **DATA_B3_REQ** have to be stored in the data part of *putmsg()*.

Note

COMMON-ISDN-API message is stored in the ctl part of *putmsg()*. In case of **DATA_B3_REQ** parameters *data* and *data length* in this ctl part of *putmsg()* are not interpreted from **COMMON-ISDN-API** implementations.

CAPI_GET_MESSAGE

Description

With this operation the application retrieves a message from **COMMON-ISDN-API**. The application retrieves all messages associated with the corresponding file descriptor from operation **CAPI_REGISTER**.

CAPI_GET_MESSAGE	**getmsg()**

Implementation

To receive a message from **COMMON-ISDN-API** the application uses the system call *getmsg()*.

The application has to supply sufficient buffers for receiving the ctl and data parts of the message. In case of receiving **COMMON-ISDN-API** message **DATA_B3_IND**, parameter *data* and *data length* of this message are not supported. Instead the data part of *getmsg()* is used to offer the transferred data.

Note

To receive a message from **COMMON-ISDN-API** the application uses the system call *getmsg()*.

8.5.2 Other Functions

CAPI_GET_MANUFACTURER

Description

With this operation the application determines the manufacturer identification of **COMMON-ISDN-API**. The offered buffer must have a size of at least 64 bytes. **COMMON-ISDN-API** copies the identification string, coded as a zero terminated ASCII string, to this buffer.

CAPI_GET_MANUFACTURER	**ioctl(): 0x06**

Implementation

This operation is realised using ioctl(0x06). The caller must supply a buffer in struct strioctl ic_dp and ic_len.

```
int fd;                           /* a valid COMMON-ISDN-API handle */
struct strioctl strioctl;
char buffer[64];

strioctl.ic_cmd = ( 'C' << 8) | 0x06;     /* CAPI_GET_MANUFACTURER */
strioctl.ic_timout = 0;
strioctl.ic_dp = buffer;
strioctl.ic_len = sizeof(buffer);
ioctl(fd, I_STR, &strioctl);
```

The manufacturer identification is transferred to the given buffer. The string is always zero-terminated.

CAPI_GET_VERSION

Description

With this function the application determines the version of **COMMON-ISDN-API** as well as an internal revision number. The offered buffer must have a size of 4 * sizeof(unsigned).

CAPI_GET_VERSION	ioctl(): 0x07

Implementation

This operation is realised using ioctl(0x07). The caller must supply a buffer in struct strioctl ic_dp and ic_len.

```
int fd;                  /* a valid COMMON-ISDN-API handle */
struct strioctl strioctl;
unsigned unsigned buffer[4];

strioctl.ic_cmd = ( 'C' << 8) | 0x07;     /* CAPI_GET_VERSION */
strioctl.ic_timout = 0;
strioctl.ic_dp = buffer;
strioctl.ic_len = sizeof(buffer);
ioctl(fd, I_STR, &strioctl);
```

The buffer consists of four elemtents:

first	**COMMON-ISDN-API** major version: 0x02
second	**COMMON-ISDN-API** minor version: 0x00
third	manufacturer-specific major number
fourth	manufacturer-specific minor number

CAPI_GET_SERIAL_NUMBER

Description

With this operation the application determines the (optional) serial number of **COMMON-ISDN-API**. The offered buffer must have a size of 8 bytes. **COMMON-ISDN-API** copies the serial number string to this buffer. The serial number, coded as a zero terminated ASCII string, represents seven digit number after the function has returned.

CAPI_GET_SERIAL_NUMBER	**ioctl(): 0x08**

Implementation

This operation is realised using ioctl(0x08). The caller must supply a buffer in struct strioctl ic_dp and ic_len.

```
int fd;                    /* a valid COMMON-ISDN-API handle */
struct strioctl strioctl;
char buffer[8];

strioctl.ic_cmd = ( 'C' << 8) | 0x08;     /* CAPI_GET_SERIAL_NUMBER */
strioctl.ic_timout = 0;
strioctl.ic_dp = buffer;
strioctl.ic_len = sizeof(buffer);
ioctl(fd, I_STR, &strioctl);
```

The serial number consists of up to seven decimal-digit ASCII characters. It is always zero-terminated.

CAPI_GET_PROFILE

Description

The application uses this function to get the capabilities from **COMMON-ISDN-API**. In the allocated buffer of 64 byte **COMMON-ISDN-API** copies information about implemented features, number of controllers and supported protocols. *CtrlNr*, which is an input parameter for **COMMON-ISDN-API**, is coded in the first bytes of the buffer and contains the controller number (bit 0..6), for which this information is requested.

CAPI_GET_PROFILE	0x09

Implementation

This operation is realised using ioctl(0x09). The caller must supply a buffer in struct strioctl ic_dp and ic_len.

```
int fd;                          /* a valid COMMON-ISDN-API handle */
struct strioctl strioctl;
char buffer[64];

                                 /* Set Controller number */
* ( (unsigned*)(&buffer[0])) = CtrlNr;

strioctl.ic_cmd = ( 'C' << 8) | 0x09;     /* CAPI_GET_PROFILE */
strioctl.ic_timout = 0;
strioctl.ic_dp = buffer;
strioctl.ic_len = sizeof(buffer);
ioctl(fd, I_STR, &strioctl);
```

Structure of command specific parameters:

Parameter	Comment
CtrlNr	Number of Controller. If 0, only number of installed controllers is given to the application.

Retrieved structure format:

Type	Description
ushort	number of installed controllers, least significant octet first
ushort	number of supported B-channels, least significant octet first

unsigned	Global Options (bit field): [0]: internal controller supported [1]: external equipment supported [2]: Handset supported (external equipment must be set also) [3]: DTMF supported [4].[31]: reserved
unsigned	B1 protocols support (bit field): [0]: 64 kBit/s with HDLC framing, always set. [1]: 64 kBit/s bit transparent operation with byte framing from the network [2]: V.110 asynchronous operation with start/stop byte framing [3]: V.110 synchronous operation with HDLC framing [4]: T.30 modem for fax group 3 [5]: 64 kBit/s inverted with HDLC framing. [6]: 56 kBit/s bit transparent operation with byte framing from the network [7]..[31]: reserved
unsigned	B2 protocol support (bit field): [0]: ISO 7776 (X.75 SLP), always set [1]: Transparent [2]: SDLC [3]: LAPD according Q.921 for D channel X.25 [4]: T.30 for fax group 3 [5]: Point to Point Protocol (PPP) [6]: Transparent (ignoring framing errors of B1 protocol) [7]..[31]: reserved
unsigned	B3 protocol support (bit field): [0]: Transparent, always set [1]: T.90NL with compatibility to T.70NL according to T.90 Appendix II. [2]: ISO 8208 (X.25 DTE-DTE) [3]: X.25 DCE [4]: T.30 for fax group 3 [5]..[31]: reserved
6 unsigned	reserved for **COMMON-ISDN-API** usage
5 unsigned	manufacturer specific information

Note

This function can be extended, so an application has to ignore unknown bits. **COMMON-ISDN-API** will set every reserved field to 0.

8.6 NetWare

The NetWare server operating system provides an open, non-preemptive, multitasking platform including file, print, communications and other services. A typical NetWare server can support tens to hundreds of simultaneous users. Extensibility of communication services in particular is accommodated through open service interfaces allowing integration of third party hardware and software. Therefore when considering the addition of a new communications subsystem to the NetWare operating system, scalability and flexibility are considered primary design goals.

This implementation of COMMON-ISDN-API in the NetWare server operating system addresses both scalability and flexibility by allowing concurrent operation of multiple CAPI compliant applications and multiple ISDN controllers provided by different manufacturers. COMMON-ISDN-API service provider in the NetWare operating system environment is a subset of the overall NetWare CAPI Manager subsystem. The NetWare CAPI Manager includes all standard functions defined by COMMON-ISDN-API v2.0 as well as auxiliary functions providing enhanced ISDN resource management for NetWare systems running multiple concurrent CAPI applications. The NetWare CAPI Manager subsystem also includes a secondary service interface which integrates each manufacturer specific ISDN controller driver below COMMON-ISDN-API. Although the driver interface maintains the general structure and syntax of CAPI functions and messages, it is not part of COMMON-ISDN-API v2.0 definition. The driver interface is unique to the NetWare CAPI Manager implementation.

The following description of COMMON-ISDN-API within the NetWare server operating system provides a detailed description of each standard COMMON-ISDN-API function which makes up the application programming interface, containing sufficient information to implement CAPI compliant applications within the NetWare environment. A general overview of the NetWare CAPI Manager is also provided to identify which services are standard COMMON-ISDN-API and which are unique to the NetWare CAPI Manager subsystem. Detailed description of the NetWare CAPI Manager unique functions for enhanced resource management and ISDN controller software integration is beyond the scope of this document. The complete definition is contained in the Novell specification **NetWare CAPI Manager and CAPI Driver specification** (Version 2.0).

Architectural Overview

The NetWare CAPI Manager, which is implemented as a NetWare Loadable Module (NLM) acts as a service multiplexer and common interface point between CAPI compliant applications and each manufacturer specific ISDN controller driver residing below COMMON-ISDN-API. Each CAPI application and each controller driver is implemented as separate NLM which independently registers with the NetWare CAPI Manager at initialization time. COMMON-ISDN-API exists between the CAPI applications and the NetWare CAPI Manager. NetWare CAPI Manager auxiliary management functions also exits at this point. A Novell defined service interface exists between the NetWare CAPI Manager and the ISDN controller drivers however applications have no knowledge of this lower level interface. From the application perspective, the lower level driver interface is an internal detail of the NetWare CAPI Manager implementation of COMMON-ISDN-API.

Figure 1 illustrates the relationship between CAPI applications, the NetWare CAPI Manager, and manufacturer specific controller drivers and controller hardware.

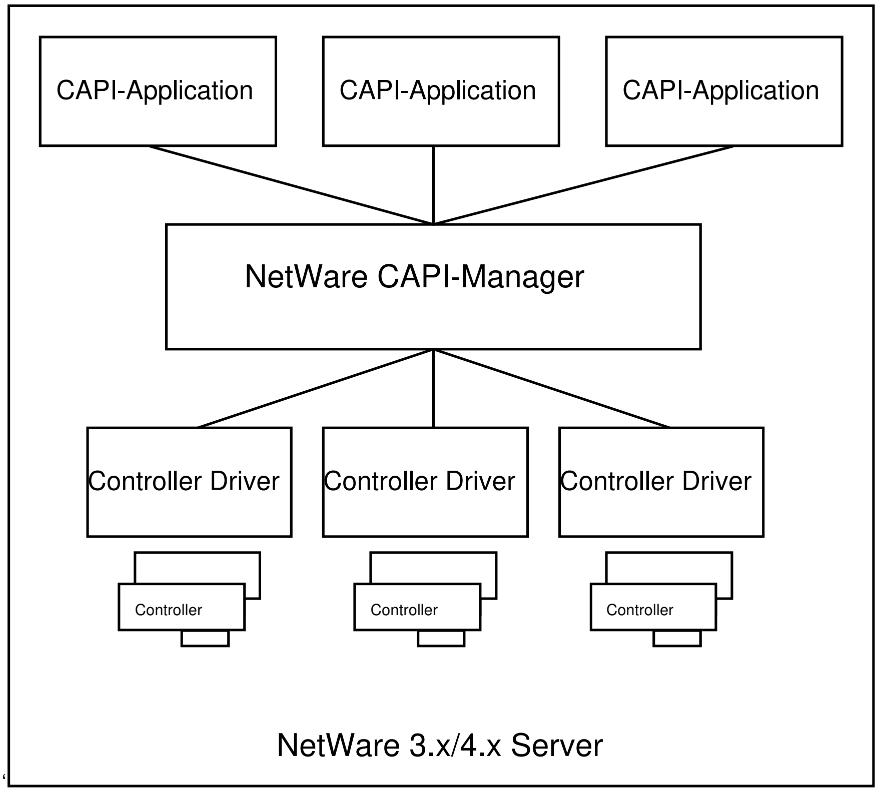


Figure 1: Architectural Overview

Services provided by the CAPI Manager are presented as a set of exported public symbols. To avoid public symbol conflicts within the server environment, services provided by each controller driver are presented as a set of entry point addresses supplied to the NetWare CAPI Manager at driver registration time. CAPI Manager services include the standard COMMON-ISDN-API function set, auxiliary functions supporting driver registration and deregistration of controller services and auxiliary management functions referenced by CAPI applications.

The additional management functions implement a powerful search mechanism for locating specific controller resources and a locking mechanism to reserve controller resources for exclusive use by an application. The CAPI_GetFirstCntlrInfo searches for the first occurrence of a controller whose capabilities match the search criteria specified by the application. The search criteria can include a symbolic controller name, specific protocols, required bandwidth etc. The CAPI_GetNextCntlrInfo function searches for additional controllers which meet the previously specified search criteria. The CAPI_LockResource function is provided for applications which must have guaranteed access to a previously identified controller channel or protocol resources. The specified resource remains reserved until the application calls the CAPI_FreeResource function. These additional management functions are intended to provide enhanced management capabilities in server systems configured with a variety of controllers or a large number of concurrently executing applications.

To insure efficient operation of multiple applications and drivers in the server environment, inbound message signaling is required by the NetWare CAPI Manager. The CAPI_Register function defines additional signal parameters must be provided by the application to successfully register. Applications are not permitted to poll for inbound messages. Because signaling is required and signal parameters are specified at registration time, the CAPI_SetSignal function is not included in this implementation of COMMON-ISDN-API.

Refer to the **NetWare CAPI Manager & CAPI Driver Specification** for a complete definition of the auxiliary and driver functions. The function descriptions provided in this section reflect only the standard COMMON-ISDN-API function set provided by the NetWare CAPI Manager. Note that in some cases the parameter lists required by the NetWare CAPI Manager version of COMMON-ISDN-API functions are different from other operating system implementations.

Function Call Conventions in NetWare environment:

- All interface functions conform to standard C language calling conventions.
- All functions can be called from either a process or interrupt context.
- COMMON-ISDN-API defines a standard 16 bit error code format where bits 8-15 identify the error class and bits 0-7 identify the specific error. With one exception, this approach is used throughout this specification. The exception is that all functions return either a DWORD (unsigned long) or a void type rather than a 16 bit WORD type. Bits 31-16 of the return value will always be zero.

Data Type Conventions in NetWare environment:

- Structures were used with byte alignment.
- The following additional simple data types were used:

BYTE	unsigned 8 bit integer value
WORD	unsigned 16 bit integer value
DWORD	unsigned 32 bit integer value
BYTE *	32 bit pointer to an unsigned char
WORD *	32 bit pointer to an unsigned 16 bit integer
VOID *	32 bit pointer
VOID **	32 bit pointer to a 32 bit pointer

8.6.1 Message operations

CAPI_Register

Description

Applications use *CAPI_Register* to register their presence with **COMMON-ISDN-API**. Registration parameters specify the maximum number of ISDN logical connections, message buffer size, number of data buffers and data buffer size required by the application. Message buffer size is normally calculated according to following formula:

Message buffer size = 1024 + (1024 * number of ISDN logical connections)

Inbound message signalling parameters are also supplied. Successful registration causes **COMMON-ISDN-API** to assign a system unique application identifier to the caller. The application identifier is used in subsequent **COMMON-ISDN-API** function calls as well as in **COMMON-ISDN-API** defined messages. Two inbound message availability signalling options are supported. The signalType and signalHandle parameters allow an application to select either CLIB Local Semaphore or direct function call-back notification. Application polling of the inbound message queue is not permitted. Successful application registration requires selection of an inbound message signalling mechanism.

Applications which maintain a CLIB process context should select Local Semaphore signalling via the signalType parameter and supply a previously allocated Local Semaphore handle as the signalHandle parameter. The application receive process can then wait on the local semaphore. When an inbound message is available, the CAPI driver will signal the local semaphore causing the application process to wakeup and retrieve a message, by calling the *CAPI_GetMessage* function.

Applications which do not maintain a CLIB process context should select direct call-back signalling via the signalType parameter, supply a pointer to an application resident notification function as the signalHandle parameter and an application defined context value as the signalContext parameter. When an inbound message is available, **COMMON-ISDN-API** will call the specified application notification function, supplying the application context value. The application has to call the *CAPI_GetMessage* function to retrieve any available messages.

Function call

```
DWORD CAPI_Register(    WORD messageBufSize,
                        WORD connectionCnt,
                        WORD dataBlockCnt,
                        WORD dataBlockLen,
                        WORD *applicationID
                        WORD signalType,
                        DWORD signalHandle,
                        DWORD signalContext,
                        );
```

Parameter	Comment
messageBufSize	specifies the message buffer size
connectionCnt	specifies the maximum number of logical connections this application can concurrently maintain. Any application attempt to exceed the logical connection count by accepting or initiating additional connections will result in a connection establishment failure and an error indication from the CAPI driver
dataBlockCnt	specifies the maximum number of received data blocks that can be reported to the application simultaneously for each B channel connection. The number B channel data blocks has a decisive effect on the throughput of B channel data in the system and should be between 2 and 7. At least two B channel data block must be specified
dataBlockLen	specifies maximum size of a B channel data unit which can be transmitted and received. Selection of a protocol that requires larger data units and attempts to transmit or receive larger data units will result in an error from **COMMON-ISDN-API**.
applicationID	this parameter specifies a pointer to a location where the CAPI Manager will place the assigned application identifier during registration . This value is valid only if the registration operation was successful, as indicated by a return code of 0x0000.
signalType	specifies the inbound message signalling mechanism selected by the application. The signalling mechanism is used by the driver to notify the application when inbound control or data messages are available or when queue full / busy conditions change. The signalType parameter also defines the meaning of the signalHandle parameter. Two signalType constants are defined as follows: **0x0001 SIGNAL_TYPE_LOCAL_SEMAPHORE** **0x0002 SIGNAL_TYPE_CALLBACK**
signalHandle	depending on the value of the signalType parameter, signalHandle specifies either the local semaphore handle previously allocated by the application or the address of an application resident receive notification function with the following format: **void CAPI_ReceiveNotify(DWORD signalContext);** (see below)
signalContext	if the signalType parameter contains SIGNAL_TYPE_CALLBACK, the signalContext specifies an application defined context value. This value will be passed to the application notification function. The signalContext value has no meaning to the CAPI. It may be used by an application to reference internal data structures etc during receive notification callback process. If the signalType parameter specifies SIGNAL_TYPE_LOCAL_SEMAPHORE this value is ignored.

Return Value

Return Value	Comment
0x0000	Registration successful - application identification number has been assigned
All other values	Coded as described in parameter info class 0x10xx

CAPI_ReceiveNotify

Description

This optional application resident receive notification function is called by the NetWare CAPI Manager implementation of the COMMON-ISDN-API whenever an inbound message addressed to the application is available. This function is intended for exclusive use by NetWare system applications which do not maintain a CLIB context. Use of this function is enabled at application registration time by specifying the CAPI_Register signalType parameter as SIGNAL_TYPE_CALLBACK. Note that non system level applications should always use local semaphores for receive message notification by specifying the CAPI_Register signalType parameter as SIGNAL_TYPE_LOCAL_SEMAPHORE.

Each time the CAPI_ReceiveNotify function is called, it should in turn call the CAPI_GetMessage to retrive the next available message addressed to the application. The signalContext parameter passed to the CAPI_ReceiveNotify function contains an application defined context value previously supplied to the CAPI_Register function. This value is meaningful only to the application, for example as an internal data structure pointer

Note

The CAPI_ReceiveNotify function can be called from either the process or interrupt context. To avoid adverse system impact, blocking operations such as disk input output should not performed by the receive notify function. If blocking operations are required they should be executed from a separate application supplied process.

CAPI_Release

Description

Applications uses *CAPI_Release* to deregister from **COMMON-ISDN-API**. All memory allocated on behalf of the application by **COMMON-ISDN-API** will be released.

Function call

```
DWORD CAPI_Release (WORD ApplID);
```

Parameter	Comment
ApplID	Application identification number that had been assigned by call of the function CAPI_Register

Return Value

Return Value	Comment
0x0000	Release of the application successful
All other values	Coded as described in parameter info class 0x11xx

CAPI_PutMessage

Description

Applications call *CAPI_PutMessage* to transfer a single message to **COMMON-ISDN-API**.

Function call

```
DWORD CAPI_PutMessage(    WORD ApplID,
                          VOID *pCAPIMessage
                          );
```

Parameter	Comment
ApplID	Application identification number that had been assigned by call of the function CAPI_Register
pCAPIMessage	points to a memory block which contains a message for the CAPI Driver

Return Value

Return Value	Comment
0x0000	No error
All other values	Coded as described in parameter info class 0x11xx

Note

When the process returns from the function call the message memory area can be reused by the application.

CAPI_GetMessage

Description

Applications call *CAPI_GetMessage* to retrieve a single message from **COMMON-ISDN-API**. If a message is available, it address is returned to the application in location specified by the *ppCAPIMessage* parameter. If there are no messages available from any of the registered drivers, *CAPI_GetMessage* returns with an error indication

The contents of the message blocks returned by this function is valid until the same application calls *CAPI_GetMessage* again. In cases where the application will process the message asynchronously or needs to maintain the message beyond the next call to *CAPI_GetMessage,* a local copy of the message has to be made by the application.

Function call

```
DWORD CAPI_GetMessage(    WORD ApplID,
                          VOID** ppCAPIMessage
                          );
```

Parameter	Comment
ApplID	Application identification number that had been assigned by call of the function CAPI_Register
ppCAPIMessage	pointer to the memory location where the CAPI Manager should place the retrieved message address. The contents of the output variable specified by msgPtr is valid only if the return code indicates no error

Return Value

Return Value	Comment
0x0000	Successful - Message was retrieved from **COMMON-ISDN-API**
All other values	Coded as described in parameter info class 0x11xx

8.6.2 Other functions

CAPI_GetManufacturer

Description

Applications call *CAPI_GetManufacturer* to retrieve manufacturer specific identification information from the specified ISDN controller.

Function call

DWORD CAPI_GetManufacturer(	**DWORD Controller,** **BYTE *szBuffer** **);**

Parameter	Comment
Controller	specifies the system unique controller number for which the manufacturer information is to be retrieved. Coding is described in Chapter 6.
szBuffer	specifies a pointer to an application data area 64 bytes long which will contain the manufacturer identification information upon successful return. The identification information is represented as a zero terminated ASCII text string.

Return Value

Return Value	Comment
0x0000	Successful - information was retrieved from **COMMON-ISDN-API**
All other values	Coded as described in parameter info class 0x11xx

CAPI_GetVersion

Description

Applications call *CAPI_GetVersion* to retrieve version information from the specified ISDN controller. Major and minor version numbers are returned for both **COMMON-ISDN-API** and the manufacturer specific implementation.

Function call

```
DWORD CAPI_GetVersion(   DWORD Controller,
                         WORD* pCAPIMajor,
                         WORD* pCAPIMinor,
                         WORD* pManufacturerMajor,
                         WORD* pManufacturerMinor
                         WORD *pManagerMajor
                         WORD *pManagerMinor
                         );
```

Parameter	Comment
Controller	specifies the system unique controller number for which the manufacturer information is to be retrieved. Coding is described in Chapter 6.
pCAPIMajor	pointer to a WORD receiving **COMMON-ISDN-API** major version number: 0x0002
pCAPIMinor	pointer to a WORD receiving **COMMON-ISDN-API** minor version number: 0x0000
pManufacturerMajor	pointer to a WORD receiving manufacturer specific major number
pManufacturerMinor	pointer to a WORD receiving manufacturer specific minor number
pManagerMajor	pointer to a WORD receiving CAPI Manager major version number
pManagerMinor	pointer to a WORD receiving CAPI Manager minor version number

Return Value

Return	Comment
0x0000	No error, version numbers are copied
All other values	Coded as described in parameter info class 0x11xx

CAPI_GetSerialNumber

Description

Applications call *CAPI_GetSerialNumber* to retrieve the optional serial number of the specified ISDN controller.

Function call

DWORD CAPI_GetSerialNumber(	**DWORD Controller,** **BYTE *szBuffer** **);**

Parameter	Comment
Controller	specifies the system unique controller number for which the serial number information is to be retrieved. Coding is described in Chapter 6.
szBuffer	pointer to a buffer of 8 bytes

Return Value

Return	Comment
0x0000	No error szBuffer contains the serial number in plain text in the form of a 7-digit number. If no serial number is provided by the manufacturer, an empty string is returned.
All other values	Coded as described in parameter info class 0x11xx

CAPI_GetProfile

Description

The application uses this function to get the capabilities from **COMMON-ISDN-API**. *Buffer* on call is a pointer to a buffer of 64 bytes. In this buffer **COMMON-ISDN-API** copies information about implemented features, number of controllers and supported protocols. *Controller* contains the controller number (bit 0..6), for which this information is requested.

DWORD CAPI_GetProfile (	**VOID *Buffer,** **DWORD Controller** **);**

Parameter	Comment
Buffer	pointer to a buffer of 64 bytes
Controller	Number of Controller. If 0, only number of installed controllers is given to the application.

Return Value

Return	Comment
0x0000	No error Buffer contains the requested information.
All other values	Coded as described in parameter info class 0x11xx

Retrieved structure format:

Type	Description
WORD	number of installed controllers, least significant octet first
WORD	number of supported B-channels, least significant octet first
DWORD	Global Options (bit field): [0]: internal controller supported [1]: external equipment supported [2]: Handset supported (external equipment must be set also) [3]: DTMF supported [4].[31]: reserved

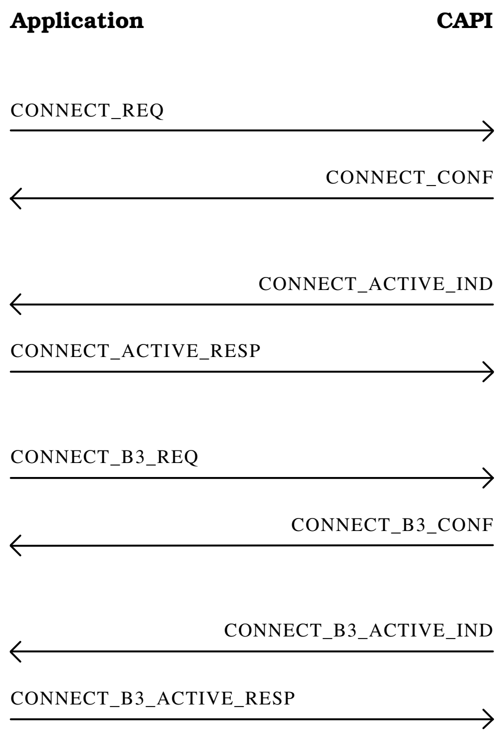
DWORD	B1 protocols support (bit field): [0]: 64 kbit/s with HDLC framing, always set. [1]: 64 kbit/s bit transparent operation with byte framing from the network [2]: V.110 asynchronous operation with start/stop byte framing [3]: V.110 synchronous operation with HDLC framing [4]: T.30 modem for fax group 3 [5]: 64 kbit/s inverted with HDLC framing. [6]: 56 kbit/s bit transparent operation with byte framing from the network [7]..[31]: reserved
DWORD	B2 protocol support (bit field): [0]: ISO 7776 (X.75 SLP), always set [1]: Transparent [2]: SDLC [3]: LAPD according Q.921 for D channel X.25 [4]: T.30 for fax group 3 [5]: Point to Point Protocol (PPP) [6]: Transparent (ignoring framing errors of B1 protocol) [7]..[31]: reserved
DWORD	B3 protocol support (bit field): [0]: Transparent, always set [1]: T.90 NL with compatibility to T.70 NL according to T.90 Appendix II. [2]: ISO 8208 (X.25 DTE-DTE) [3]: X.25 DCE [4]: T.30 for fax group 3 [5]..[31]: reserved
6 DWORDs	reserved for **COMMON-ISDN-API** usage
5 DWORDs	manufacturer specific information

Note

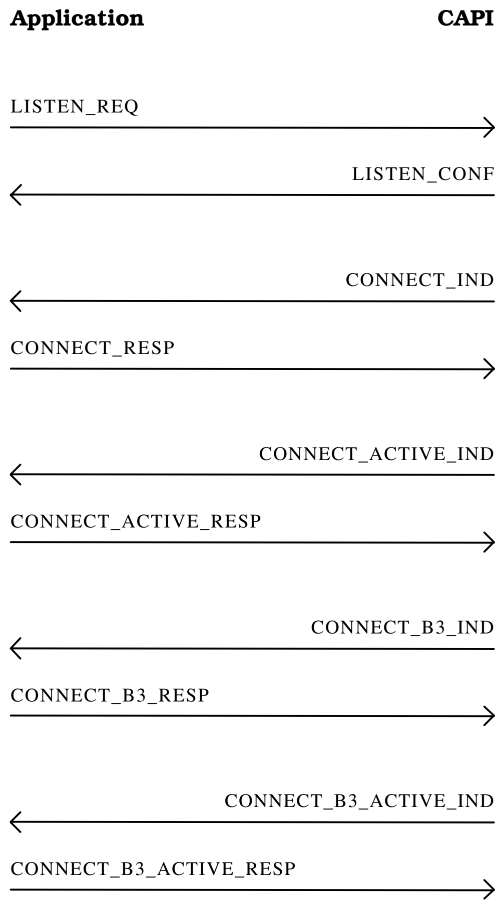
This function can be extended, so an application has to ignore unknown bits. **COMMON-ISDN-API** will set every reserved field to 0.

ANNEX A (INFORMATIVE): SAMPLE FLOW CHART DIAGRAMS

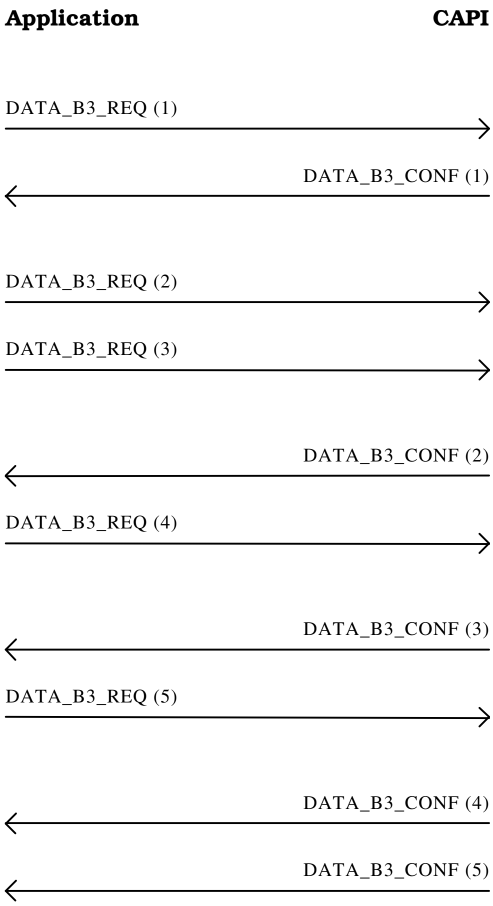
A.1 Outgoing call



A.2 Incoming call

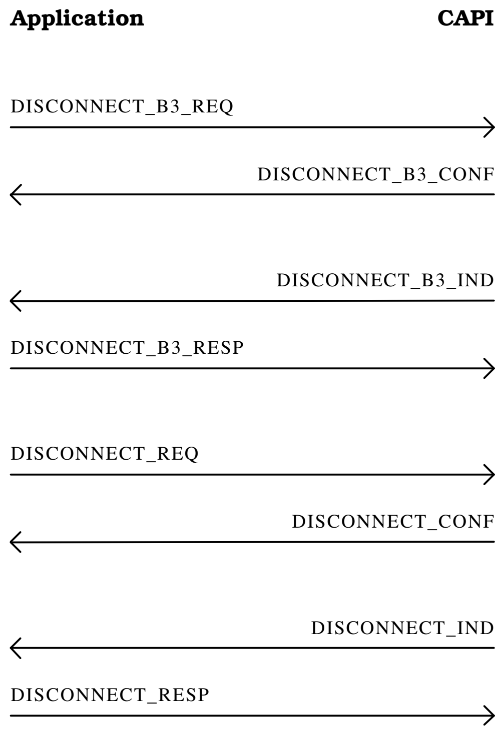


A.3 Transmitting Data

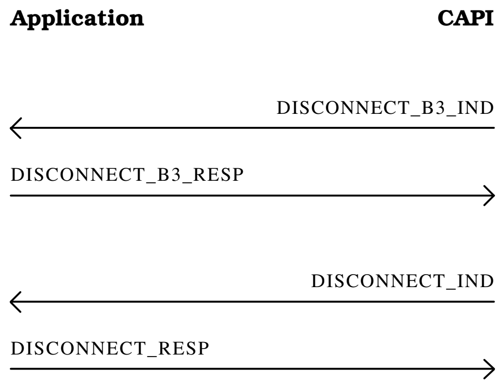


A.4 Receiving Data

A.5 Active disconnect

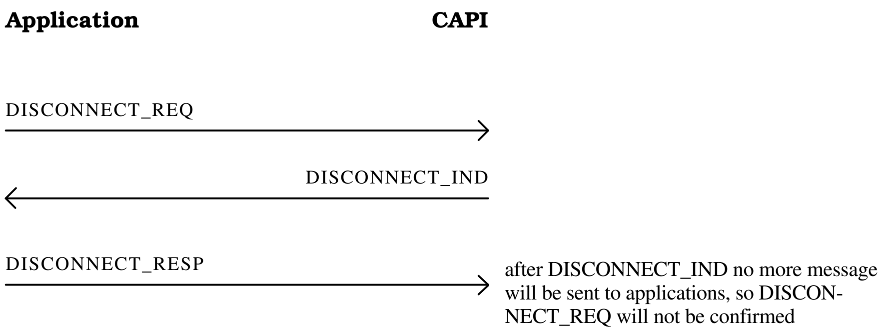


A.6 Passive disconnect

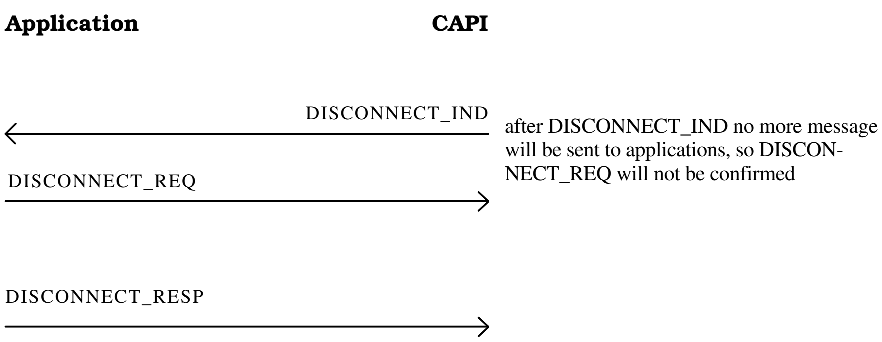


A.7 Disconnect Collision

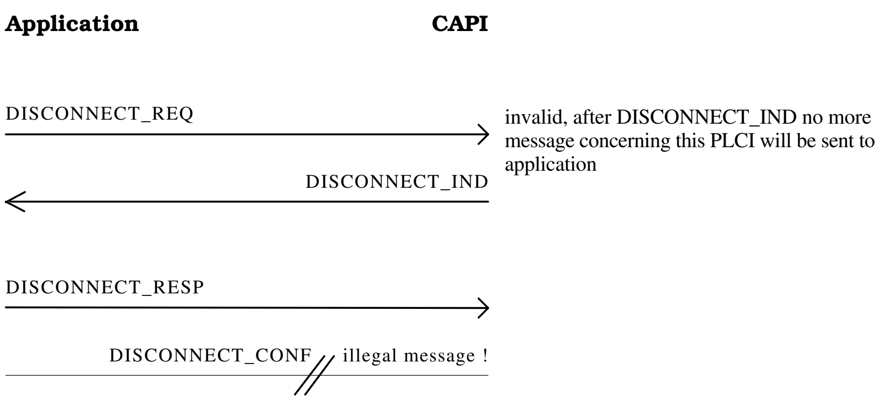
Simultaneous release of a physical connection by application and **COMMON-ISDN-API**



also possible:



illegal:



ANNEX B (NORMATIVE): SFF FORMAT

B.1 Introduction

SFF (**S**tructured **F**ax **F**ile) is a representation specially for fax group 3 documents. It consists of information concerning the page structure and compressed line data of the fax document. A SFF formatted document always starts with a header, valid for the complete document. Every page will start with a page header. After this the pixel information follows line by line. As the SFF format is a file format specification, some entries in header structures (e.g. double chaining of pages) may not used or supported by **COMMON-ISDN-API**.

document header	page 1 header	page 1 data	page 2 header	page 2 data	.....	page n data

Figure 6: SFF format

B.2 SFF coding rules

Following type conventions are used:

byte	8 bit unsigned
word	16 bit unsigned integer, least significant octet first
dword	32 bit unsigned integer, least significant word first

B.2.1 Document header

Parameter	Type	Comment
SFF_Id	dword	magic value (identification) of SFF Format: coded as **0x66666653** ("**SFFF**")
Version	byte	version number of SFF document: coded **0x01**
reserved	byte	reserved for future extensions, coded **0x00**
User Information	word	manufacturer specific user information (not used by **COMMON-ISDN-API**, coded as **0x0000**)
Page Count	word	number of document's pages. If not known (in case of receiving a document) it has to be coded **0x0000**.
OffsetFirstPageHeader	word	byte offset of first page header from start of document header. This value is normally equal to the size of the document header (**0x14**), but there might be additional user specific data between document header and first page header. **COMMON-ISDN-API** will ignore and not offer such additional data.
OffsetLastPageHeader	dword	byte offset of last page header from start of document header. If not known (in case of receiving a document) it has to be coded **0x00000000**.

OffsetDocumentEnd	dword	byte offset to document end from start of document header. If not known (in case of receiving a document) it has to be coded **0x00000000**.

B.2.2 Page header

Parameter	**Type**	**Comment**
PageHeaderID	byte	**254** (Record Type of Page Data)
PageHeaderLen	byte	**0:** Document end **1..255**: byte offset of first page data from entry *Resolution Vertical* of page header. This value is normally equal to the size of the following part of the header (**0x10**), but there might be additional user specific data between page header and page data. **COMMON-ISDN-API** will ignore and not offer such additional data.
Resolution Vertical	byte	definition of vertical resolution; different resolutions in one document may be ignored by **COMMON-ISDN-API**. **0:** 98 lpi (standard) **1:**: 196 lpi (high resolution) **2..254:** reserved **255:** end of document (should not be used, instead *Page-HeaderLen* should be coded **0**)
Resolution Horizontal	byte	definition of horizontal resolution **0:** 203 dpi (standard) **1..255:** reserved
Coding	byte	definition of pixel line coding **0:** modified Huffman coding **1..255**: reserved
reserved	byte	coded as **0**
Line Length	word	number of pixels per line **1728:** standard fax g3 **2048:** B4 (optional) **2432:** A3 (optional) Support of other values also is optional for **COMMON-IS-DN-API**.
Page Length	word	number of pixel lines per page. If not known, coded as **0x0000**.
OffsetPreviousPage	dword	byte offset to previous page header or **0x00000000**. Coded as **0x00000001** if first page.
OffsetNextPage	dword	byte offset to next page header or **0x00000000**. Coded as **0x00000001** if last page.

B.2.3 Page data

Page data is coded line by line, i.e. for each pixel row exists a data definition. Lines are coded as records with variable length, each line is coded according to element *coding* in page header. For the moment only modified Huffman coding is supported. MH-coding is byte oriented, the first bit or a code word is stored least significant first. There are no EOL code words or fill bits included. If data include EOL code words, **COMMON-ISDN-API** will ignore these coding.

Each record is identified by the first byte:

- **1..216:** pixel row with 1..216 MH-coded bytes are following immediately
- **0:** escape for pixel row with more than 216 bytes MH-coding. In this case, a following word in the range **217..32767** defines the number of MH-coded bytes, which are following.
- **217..253**: white skip, 1..37 empty lines
- **254**: start or page header (see there)
- **255:** if followed by a byte with value **0**, illegal line coding. An application can decide if this line should be interpreted empty or as a copy of the previous line. If this byte is followed by a byte with a value **1..255**, 1..255 bytes additional user information are following (reserved for future extensions).

INDEX