COMMON-ISDN-API

Version 2.0

Part II

Operating Systems

4th Edition

June 2001

Author:
CAPI Association e.V.
All rights reserved

Editor:
AVM GmbH, Germany
E-mail: hj.ortmann@avm.de

4th Edition / June 2001

Publisher:
CAPI Association e.V.
http://www.capi.org/

Contents (Part II)

8 Specifications for Commercial Operating Systems

COMMON-ISDN-API can be used with the following operating systems:

All operating systems support the following **COMMON-ISDN-API** operations:

• CAPI_REGISTER	Register application with **COMMON-ISDN-API**
• CAPI_RELEASE	Release application from **COMMON-ISDN-API**
• CAPI_PUT_MESSAGE	Transfer message to **COMMON-ISDN-API**
• CAPI_GET_MESSAGE	Retrieve message from **COMMON-ISDN-API**
• CAPI_GET_MANUFACTURER	Get manufacturer information from **COMMON-ISDN-API**
• CAPI_GET_VERSION	Get version information from **COMMON-ISDN-API**
• CAPI_GET_SERIAL_NUMBER	Get serial number of **COMMON-ISDN-API**
• CAPI_GET_PROFILE	Get capability information from **COMMON-ISDN-API**

Depending on the operating system, the following COMMON-ISDN-API operations may also be available:

• CAPI_SET_SIGNAL	Install call-back function
• CAPI_WAIT_FOR_SIGNAL	Wait for **COMMON-ISDN-API** message
• CAPI_INSTALLED	Check whether **COMMON-ISDN-API** is installed
• CAPI_MANUFACTURER	Manufacturer-specific **COMMON-ISDN-API** operation

CAPI_GET_PROFILE

CAPI_GET_PROFILE is used to obtain information on **COMMON-ISDN-API**'s implemented capabilities. This operation fills in a buffer with the following structure:

Type	**Description**
2 bytes	Number of controllers installed, least significant byte first
2 bytes	Number of supported B-channels, least significant byte first
4 bytes	Global Options (bit field): [0]: Internal controller supported [1]: External equipment supported [2]: Handset supported (external equipment must also be set) [3]: DTMF supported [4]: Supplementary Services supported (see Part III) [5]: Channel allocation supported (leased lines) [6]: Parameter *B channel operation* supported [7]: Line Interconnect supported [8]...[31]: reserved
4 bytes	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 Group 3 fax [5]: 64 kbit/s inverted with HDLC framing. [6]: 56 kbit/s bit-transparent operation with byte framing from the network [7]: Modem with all negotiations [8]: Modem asynchronous operation with start/stop byte framing [9]: Modem synchronous operation with HDLC framing [10]..[31]: reserved
4 bytes	B2 protocol support (bit field): [0]: ISO 7776 (X.75 SLP), always set [1]: Transparent [2]: SDLC [3]: LAPD in accordance with Q.921 for D channel X.25 (SAPI 16) [4]: T.30 for Group 3 fax [5]: Point-to-Point Protocol (PPP) [6]: Transparent (ignoring framing errors of B1 protocol) [7]: Modem error correction and compression (V.42 bis or MNP5) [8]: ISO 7776 (X.75 SLP) modified supporting V.42 bis compression [9]: V.120 asynchronous mode [10]: V.120 asynchronous mode supporting V.42 bis [11]: V.120 bit-transparent mode [12]: LAPD in accordance with Q.921 including free SAPI selection [13]..[31]: reserved

4 bytes	B3 protocol support (bit field): [0]: Transparent, always set [1]: T.90NL with compatibility to T.70NL in accordance with T.90 Appendix II. [2]: ISO 8208 (X.25 DTE-DTE) [3]: X.25 DCE [4]: T.30 for Group 3 fax [5]: T.30 for Group 3 fax with extensions [6]: reserved [7]: Modem [8]..[31]: reserved
24 bytes	reserved for **COMMON-ISDN-API** use
20 bytes	Manufacturer-specific information

CAPI_GET_PROFILE structure format

An application must ignore unknown bits. **COMMON-ISDN-API** sets every reserved field to 0.

8.1 MS-DOS

As MS-DOS does not provide any multitasking facilities, **COMMON-ISDN-API** is incorporated into the system as a background (terminate and stay resident) driver. The interface between the application and **COMMON-ISDN-API** is implemented using 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 processor register assignments in this software interrupt interface. The return values and parameters are normally provided in registers AX and ES:BX. Registers AX, BX, CX, DX and ES can be modified; other registers are preserved. **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 space must be furnished by the application program. While 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 ascertain the presence of **COMMON-ISDN-API**.

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

Memory layout is in conformance with MS-DOS.

8.1.1 Message Operations

8.1.1.1 CAPI_REGISTER

Description

This is the function the application uses to announce 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 by the following formula:

CX + (DX * SI * DI)

The size of the message buffer used to store messages is transferred in the *CX* register. Setting this value too small will result in messages being lost. For a typical application, the amount of memory required should be calculated by the following formula:

CX = 1024 + (1024 * DX)

In the *DX* register, the application indicates the maximum number of logical connections this application can maintain concurrently. Any attempt by the application to exceed the logical connection count by accepting or initiating additional logical connections will result in a connection set-up failure and an error indication from **COMMON-ISDN-API**.

In the *SI* register, the application sets the maximum number of received data blocks that can be reported to the application simultaneously for each logical connection. The number of simultaneously available data blocks has a decisive effect on the data throughput in the system and should be between 2 and 7. At least two data blocks must be specified.

In the *DI* register the application specifies the maximum size of the application data block to be transmitted and received. Selection of a protocol that requires larger data units, or attempts to transmit or receive larger data units, will result in an error indication from **COMMON-ISDN-API**. The default value for the protocol ISO 7776 (X.75) is 128 octets. **COMMON-ISDN-API** is able to support at least up to 2048 octets.

The application ID number is returned in *AX*. In the event of an error, the value 0 is returned in *AX*, and the cause of the error is indicated in *BX*.

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) present the received data in this memory area.
CX	Size of message buffer
DX	Maximum number of Layer 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 at a time and use the standard protocols, the following register assignments are recommended:

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

The resulting memory requirement is 2944 bytes.

8.1.1.2 CAPI_RELEASE

Description

The application uses this function to log out from **COMMON-ISDN-API**. The memory area indicated in the application’s **CAPI_REGISTER** call is released. The application is identified by the application number in the *DX* register. Any errors that occur are returned in *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 <> 0	No error Registration error, coded as described in parameter *Info*, class 0x11xx

8.1.1.3 CAPI_PUT_MESSAGE

Description

With this function the application transfers a message to **COMMON-ISDN-API**. A FAR pointer to the message is passed in the *ES:BX* registers. The application is identified by the application number in the *DX* register. Any errors that occur are returned in *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 returning from the **CAPI_PUT_MESSAGE** call, the application can re-use the memory area of the message. The message is not modified by **COMMON-ISDN-API**.

8.1.1.4 CAPI_GET_MESSAGE

Description

With this function the application retrieves a message from **COMMON-ISDN-API**. The application can only retrieve messages intended for the specified application number. A FAR pointer to the message is passed in the *ES:BX* registers. The function returns immediately, even if no message was queued for retrieval. Register *AX* contains the corresponding error value. The application is identified by the application number in the *DX* register. Any errors that occur are returned in *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 made invalid by the next **CAPI_GET_MESSAGE** call.

8.1.2 Other Functions

8.1.2.1 CAPI_SET_SIGNAL

Description

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

Return Value

Return	**Value**	**Comment**
AX	0x0000 <> 0	No error 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 the application's message queue,
- an announced busy condition is cleared, or
- an announced queue-full condition is cleared.

Interrupts are disabled. The call-back function must be terminated by IRET. All registers must be preserved. When the function is called, at least 32 bytes are available on the stack.

The call-back function is called with interrupts disabled. **COMMON-ISDN-API** shall not call this function recursively, even if the call-back function enables interrupts. Instead, the call-back function shall be called again after it returns control to **COMMON-ISDN-API**.

The call-back function is allowed to use the **COMMON-ISDN-API** operations **CAPI_PUT_MESSAGE**, **CAPI_GET_MESSAGE**, and **CAPI_SET_SIGNAL**. If it does so, the application must take into account the fact that interrupts may be enabled by **COMMON-ISDN-API**.

In the case of local confirmations (such as LISTEN_CONF), the call-back function may be called before the operation CAPI_PUT_MESSAGE returns control to the application.

Registers DX, SI and DI are passed to the call-back function with the same values as the corresponding parameters to **CAPI_SET_SIGNAL**.

8.1.2.2 CAPI_GET_MANUFACTURER

Description

By calling this function the application obtains the **COMMON-ISDN-API** manufacturer identification. The application provides a FAR pointer to a data area of 64 bytes in registers *ES:BX*. 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 in ASCII. The end of the identification is indicated by a zero byte.

8.1.2.3 CAPI_GET_VERSION

Description

With this function the application obtains 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

8.1.2.4. CAPI_GET_SERIAL_NUMBER

Description

With this function the application obtains the (optional) serial number of **COMMON-ISDN-API**. The application provides a FAR pointer to a data area of 8 bytes in registers *ES:BX*. The serial number, a seven-digit number coded as a zero-terminated ASCII string, is present in this data area 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 a 7-digit number in plain text. The end of the serial number is indicated by a zero byte. If no serial number is to be used, a zero byte must be written at the first position in the buffer.

8.1.2.5 CAPI_GET_PROFILE

Description

The application uses this function to determine the capabilities of **COMMON-ISDN-API**. Registers *ES:BX must* contain a FAR pointer to a data area of 64 bytes. **COMMON-ISDN-API** copies information about implemented features, the number of controllers and supported protocols to this buffer. Register *CX* contains the number of the controller (bits 0..6) for which this information is requested. The profile structure is described at the beginning of Chapter 8.

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	No error
	<> 0	Coded as described in parameter *Info*, class 0x11xx

8.1.2.6 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 3.x (Application Level)

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

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

The functions are exported under the 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 as imported functions in accordance with the DLL conventions. Whenever an application calls any function of the DLL for any purpose, it must ensure that there are at least 512 bytes available 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 applies in particular to the data pointer in **DATA_B3_REQ** and **DATA_B3_IND** messages.

In the Windows 3.x environment, the following data types are used in defining 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 an 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 (in accordance with the Windows DLL calling convention)

8.2.1 Message Operations

8.2.1.1 CAPI_REGISTER

Description

This is the function the application uses to announce its presence to **COMMON-ISDN-API**. The application describes its needs by passing the four parameters *MessageBufferSize*, *maxLogicalConnection*, *maxBDataBlocks* and *maxBDataLen*.

For a typical application, the amount of memory required should be calculated by the following formula:

MessageBufferSize = 1024 + (1024 * maxLogicalConnection)

The parameter *maxLogicalConnection* specifies the maximum number of logical connections that the application can maintain concurrently. Any attempt by the application to exceed the logical connection count by accepting or initiating additional connections will result in a connection set-up failure and an error indication from **COMMON-ISDN-API**.

The parameter *maxBDataBlocks* specifies the maximum number of received data blocks that can be reported to the application simultaneously for each logical connection. The number of simultaneously available data blocks has a decisive effect on the data throughput in the system and should be between 2 and 7. At least two data blocks must be specified.

The parameter *maxBDataLen* specifies the maximum size of the application data block to be transmitted and received. Selection of a protocol that requires larger data units, or attempts to transmit or receive larger data units, will result in an error indication from **COMMON-ISDN-API**. The default value for the protocol ISO 7776 (X.75) is 128 octets. **COMMON-ISDN-API** is able to support at least up to 2048 octets.

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** is to 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

8.2.1.2 CAPI_RELEASE

Description

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

The application is identified by the application identification number that was assigned in the earlier CAPI_REGISTER operation.

Function call

```
CAPIENTRY CAPI_RELEASE (WORD ApplID);
```

Parameter	Comment
ApplID	Application identification number assigned by 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

8.2.1.3 CAPI_PUT_MESSAGE

Description

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

Function call

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

Parameter	**Comment**
ApplID	Application identification number assigned by the function CAPI_REGISTER
pCAPIMessage	16:16 (segmented) protected-mode pointer to the message to be 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 function call returns control to the application, the message memory area can be re-used.

8.2.1.4 CAPI_GET_MESSAGE

Description

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

Function call

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

Parameter	Comment
ApplID	Application identification number assigned by 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 message

Return Value

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

Note

The message received may be made invalid by the next CAPI_GET_MESSAGE call for the same application identification number. This is especially important to note in multi-threaded applications where more than one thread may execute CAPI_GET_MESSAGE operations. Synchronization between threads must be done by the application.

8.2.2 Other Functions

8.2.2.1 CAPI_SET_SIGNAL

Description

This operation is used by the application to install a mechanism by which **COMMON-ISDN-API** signals the availability of a message or the clearing of an internal busy or queue-full condition. All restrictions pertaining to an interrupt context 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 assigned by the function CAPI_REGISTER
CAPI_Callback	Address of the call-back function. The function is called in an interrupt context (see note). The value **0x00000000** disables the call-back function**.**
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

Call-back notification takes place after:

- any message is queued in the application's message queue,
- an announced busy condition is cleared, or
- an announced queue-full condition is cleared.

In the case of local confirmations (such as LISTEN_CONF), the call-back notification may occur before the operation CAPI_PUT_MESSAGE returns control to the application.

The call-back function is called using the following conventions:

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

The data segment register DS is undefined (*MakeProcInstance()* or *_setds* may be used). A stack of at least 512 bytes is set up by **COMMON-ISDN-API.**

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

PostMessage() and *PostAppMessage()* are the only Windows API functions which may 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 is not re-entered by **COMMON-ISDN-API**. Instead, it is called again after returning if a new event has occurred during processing.

8.2.2.2 CAPI_GET_MANUFACTURER

Description

By calling this function the application obtains the **COMMON-ISDN-API** (DLL) manufacturer identification. The application furnishes a 16:16 (segmented) protected-mode pointer to a buffer of 64 bytes in szBuffer. **COMMON-ISDN-API** copies the identification, 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

8.2.2.3 CAPI_GET_VERSION

Description

With this function the application obtains 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 which receives the **COMMON-ISDN-API** major version number: 2
pCAPIMinor	16:16 (segmented) protected-mode pointer to a WORD which receives the **COMMON-ISDN-API** minor version number: 0
pManufacturerMajor	16:16 (segmented) protected-mode pointer to a WORD which receives the manufacturer-specific major number
pManufacturerMinor	16:16 (segmented) protected-mode pointer to a WORD which receives the manufacturer-specific minor number

Return Value

Return	Comment
0x0000	No error, version numbers have been copied

8.2.2.4 CAPI_GET_SERIAL_NUMBER

Description

With this operation the application obtains the (optional) serial number of **COMMON-ISDN-API**. The application provides a 16:16 (segmented) protected-mode pointer to a string buffer of 8 bytes in szBuffer. **COMMON-ISDN-API** copies the serial number string to this buffer. The serial number, a seven-digit number coded as a zero-terminated ASCII string, is present in this buffer 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.

8.2.2.5 CAPI_GET_PROFILE

Description

The application uses this function to determine the capabilities of **COMMON-ISDN-API**. The application provides a 16:16 (segmented) protected-mode pointer to a buffer of 64 bytes in szBuffer. **COMMON-ISDN-API** copies information about implemented features, the number of controllers and supported protocols to this buffer. *CtrlNr* contains the number of the controller (bits 0..6) for which this information is requested. The profile structure retrieved is described at the beginning of Chapter 8.

```
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 the number of controllers installed is provided to the application.

Return Value

Return	Comment
0x0000 <> 0	No error Coded as described in parameter *Info,* class 0x11xx

8.2.2.6 CAPI_INSTALLED

Description

This function can be used by an application to determine whether 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 the operating system OS/2 Version 2.x, application programs can access **COMMON-ISDN-API** services via a DLL (Dynamic Link Library). The interface between applications and **COMMON-ISDN-API** is realized as a function interface. Applications issue **COMMON-ISDN-API** function calls to perform **COMMON-ISDN-API** operations.

The DLL providing the function interface must be named "CAPI20.DLL". It is a 32-bit DLL which exports 32-bit functions in accordance with the System Call Convention. This means that all parameters are passed on the stack, and the calling process must clear the stack after control returns from the function call.

The functions are exported under the 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

Applications may call these functions as imported functions in accordance with the DLL conventions. When an application calls the DLL, it must ensure that there are at least 512 bytes available 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 applies in particular 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 flat pointer it receives into a 16:16-bit segmented pointer.

In the OS/2 environment, the following data types are used in defining 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

8.3.1.1 CAPI_REGISTER

Description

This is the function the application uses to announce its presence to **COMMON-ISDN-API**. The application describes its needs by passing the four parameters *MessageBufferSize*, *maxLogicalConnection*, *maxBDataBlocks* and *maxBDataLen*.

For a typical application, the amount of memory required should be calculated by the following formula:

MessageBufferSize = 1024 + (1024 * maxLogicalConnection)

The parameter *maxLogicalConnection* specifies the maximum number of logical connections this application can maintain concurrently. Any attempt by the application to exceed the logical connection count by accepting or initiating additional connections will result in a connection set-up failure and an error indication from **COMMON-ISDN-API**.

The parameter *maxBDataBlocks* specifies the maximum number of received data blocks that can be reported to the application simultaneously for each logical connection. The number of simultaneously available data blocks has a decisive effect on the data throughput in the system and should be between 2 and 7. At least two data blocks must be specified.

The parameter *maxBDataLen* specifies the maximum size of the application data block to be transmitted and received. Selection of a protocol that requires larger data units, or attempts to transmit or receive larger data units, will result in an error indication from **COMMON-ISDN-API**. The default value for the protocol ISO 7776 (X.75) is 128 octets. **COMMON-ISDN-API** is able to support at least up to 2048 octets.

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** is to 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

8.3.1.2 CAPI_RELEASE

Description

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

The application is identified by the application identification number that was assigned in the earlier CAPI_REGISTER operation.

Function call

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

Parameter	Comment
ApplID	Application identification assigned by the function CAPI_REGISTER

Return Value

Return Value	Comment
0x0000	Application successfully released
All other values	Coded as described in parameter *Info*, class 0x11xx

8.3.1.3 CAPI_PUT_MESSAGE

Description

With this operation the application transfers a message to **COMMON-ISDN-API**. The application identifies itself by its application identification number. The message memory area must not cross a 64 kbyte boundary in the flat address space (*tiled* memory may be used, for example), because the DLL may convert the flat pointer it receives from the application to a 16:16-bit segmented pointer. The same applies to the B3 data blocks passed as pointers in DATA_B3_REQ messages.

Function call

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

Parameter	Comment
ApplID	Application identification number assigned by the function CAPI_REGISTER
pCAPIMessage	0:32 (flat) pointer to the message being 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 function call returns control to the application, the message memory area can be re-used.

8.3.1.4 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 specified application identification number. If there is no message queued 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 assigned by the function CAPI_REGISTER
ppCAPIMessage	0:32 (flat) pointer to the memory location where **COMMON-ISDN-API** is to 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 be made invalid by the next **CAPI_GET_MESSAGE** operation for the same application identification number. This is particularly important in multi-threaded applications where more than one thread may execute **CAPI_GET_MESSAGE** operations. Synchronization between threads must be performed by the application.

8.3.2 Other Functions

8.3.2.1 CAPI_SET_SIGNAL

Description

This operation is used by the application to install a mechanism by which **COMMON-ISDN-API** signals the availability of a message to the application.

In OS/2 2.x this is best done using a fast 32-bit system event semaphore. The application must create the semaphore to be used 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.

When the signal is set, the specified semaphore is "posted" each time **COMMON-ISDN-API** places a message in the application's message queue, thus incrementing a post-count value associated with the semaphore. **COMMON-ISDN-API** posts the semaphore by calling the *DosPostEventSem()* function of the OS/2 system API.

The application thread may wait until the semaphore’s post-count is greater than zero using the OS/2 system call *DosWaitEventSem()*. It can also determine the current post count and simultaneously reset the post counter by executing the OS/2 system API call *DosResetEventSem()*.

The signaling mechanism is deactivated by calling the CAPI_SET_SIGNAL function with a semaphore handle of 0.

Function call

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

Parameter	Comment
ApplID	Application identification number assigned by the function CAPI_REGISTER
hEventSem	Event Semaphore handle allocated by operating system

Return Value

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

8.3.2.2 CAPI_GET_MANUFACTURER

Description

By calling this function the application obtains the **COMMON-ISDN-API** (DLL) manufacturer identification. The application provides a 0:32 (flat) pointer to a buffer of 64 bytes in szBuffer. **COMMON-ISDN-API** copies the identification, 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

8.3.2.3 CAPI_GET_VERSION

Description

With this function the application obtains 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 which receives the **COMMON-ISDN-API** major version number: 2
pCAPIMinor	0:32 (flat) protected-mode pointer to a dword which receives the **COMMON-ISDN-API** minor version number: 0
pManufacturerMajor	0:32 (flat) protected-mode pointer to a dword which receives the manufacturer-specific major number
pManufacturerMinor	0:32 (flat) protected-mode pointer to a dword which receives the manufacturer-specific minor number

Return Value

Return	Comment
0x0000	No error, version numbers have been copied.

8.3.2.4 CAPI_GET_SERIAL_NUMBER

Description

With this operation the application obtains the (optional) serial number of **COMMON-ISDN-API**. The application provides a 0:32 (flat) protected-mode pointer to a string buffer of 8 bytes in szBuffer. **COMMON-ISDN-API** copies the serial number string to this buffer. The serial number, a seven-digit number coded as a zero-terminated ASCII string, is present in this buffer 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.

8.3.2.5 CAPI_GET_PROFILE

Description

The application uses this function to determine the capabilities of **COMMON-ISDN-API**. The application provides a 0:32 (flat) protected-mode pointer to a buffer of 64 bytes in szBuffer. **COMMON-ISDN-API** copies information about implemented features, the number of controllers and supported protocols to this buffer. *CtrlNr* contains the number of the controller (bits 0..6) for which this information is requested. The profile structure retrieved is described at the beginning of Chapter 8.

```
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 controllers installed is provided to the application.

Return Value

Return	**Comment**
0x0000	No error
<> 0	Coded as described in parameter *Info*, class 0x11xx

8.3.2.6 CAPI_INSTALLED

Description

This function can be used by an application to determine whether 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 the operating system OS/2 Version 2.x, **COMMON-ISDN-API** applications may exist in the form of OS/2 physical device drivers (PDD). Such applications are referred to in the following sections as "application PDDs". This specification describes the interface of an OS/2 2.x physical device driver providing **COMMON-ISDN-API** services to other device drivers. This **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. All functions in this specification are thus 16-bit functions, and all pointers are 16:16 segmented.

In this chapter, the following data types are used in defining 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 must be "CAPI20 " (with trailing spaces to extended the name 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 must be set to 1.

Manufacturers who also wish to support **COMMON-ISDN-API** in OS/2's DOS/Windows environment must also provide the DOS/Windows 3.x interface of **COMMON-ISDN-API** in accordance with Subclauses 8.1/8.2. In this case, the PDD's name causes conflicts for Windows 3.x applications in accessing the **COMMON-ISDN-API** DLL named CAPI20.DLL. To resolve this conflict, the following new mechanism was introduced in 1996 (with the Second Edition of **COMMON-ISDN-API Version 2.0)**:

The CAPI PDD name which is contained in its device driver may be "CAPI20$ " or "CAPI20 " (both space-extended to 8 characters). The preferred method is to use "CAPI20$ ", but in order to achieve compatibility with existing PDD-applications it shall be possible to install the **COMMON-ISDN-API** PDD with the device name "CAPI20 ". In this case, the DOS/Windows 3.x interface may be disabled. PDD applications should first try to access the "CAPI20$ " device.

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 must set 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 the C calling convention: thus the calling application PDD must clear the stack after the function returns control. There must be at least 512 bytes available on the stack when the application PDD calls the IDC function. The parameter funcCode selects the CAPI operation to be performed; 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.

8.4.1 Message Operations

8.4.1.1 CAPI_REGISTER

Description

This is the function the application uses to announce its presence to **COMMON-ISDN-API**. In doing so, the application provides **COMMON-ISDN-API** with a memory area. A pointer to this memory area is transferred in parameter *Buffer*. The application describes its needs by passing the four parameters *MessageBufferSize, maxLogicalConnection*, *maxBDataBlocks* and *maxBDataLen*.

COMMON-ISDN-API uses the memory area referenced by parameter *Buffer* to store messages and data blocks sent to the application PDD. The passed memory must be either fixed or locked. **COMMON-ISDN-API** need not verify whether this storage really exists. The size of the memory area is calculated by the following formula:

MessageBufferSize + (maxLogicalConnection * maxBDataBlocks * maxBDataLen)

For a typical application PDD, the amount of memory required should be calculated by the following formula:

MessageBufferSize = 1024 + (1024 * maxLogicalConnection)

The parameter *maxLogicalConnection* specifies the maximum number of logical connections this application can maintain concurrently. Any attempt by the application to exceed the logical connection count by accepting or initiating additional connections will result in a connection set-up failure and an error indication from **COMMON-ISDN-API**.

The parameter *maxBDataBlocks* specifies the maximum number of received data blocks that can be reported to the application simultaneously for each logical connection. The number of simultaneously available data blocks has a decisive effect on the data throughput in the system and should be between 2 and 7. At least two data blocks must be specified.

The parameter *maxBDataLen* specifies the maximum size of the application data block to be transmitted and received. Selection of a protocol that requires larger data units, or attempts to transmit or receive larger data units, will result in an error indication from **COMMON-ISDN-API**. The default value for the protocol ISO 7776 (X.75) is 128 octets. **COMMON-ISDN-API** is able to support at least up to 2048 octets.

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 sent 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** is to place the assigned application identification number

Return Value

Return Value	Comment
0x0000	Registration successful: application identification number was assigned
All other values	Coded as described in parameter *Info,* class 0x10xx

8.4.1.2 CAPI_RELEASE

Description

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

The application PDD is identified by the application identification number that was assigned in the earlier CAPI_REGISTER operation.

CAPI_RELEASE	**0x02**

Structure of command-specific parameters:

Parameter	Type	Comment
ApplID	word	Application identification number assigned by the function CAPI_REGISTER

Return Value

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

8.4.1.3 CAPI_PUT_MESSAGE

Description

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

CAPI_PUT_MESSAGE	**0x03**

Structure of command-specific parameters:

Parameter	**Type**	**Comment**
ApplID	word	Application identification number assigned by the function CAPI_REGISTER
pCAPIMessage	void*	16:16 segmented pointer to the message being 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 function call returns control to the application PDD, the message memory area can be re-used.

8.4.1.4 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 specified application identification number. If there is no message queued 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 assigned by the function CAPI_REGISTER
ppCAPIMessage	void**	16:16 segmented pointer to the memory location where **COMMON-ISDN-API** is to 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 be made invalid by the next **CAPI_GET_MESSAGE** operation for the same application identification number.

8.4.2 Other Functions

8.4.2.1 CAPI_SET_SIGNAL

Description

This operation is used by the application PDD to install a mechanism by which **COMMON-ISDN-API** signals the availability of a message.

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

CAPI_SET_SIGNAL	**0x05**

Structure of command-specific parameters:

Parameter	Type	Comment
ApplID	word	Application identification number assigned by the function CAPI_REGISTER
sigFunc	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 the application's message queue,
- an announced busy condition is cleared, or
- an announced queue-full condition is cleared.

Interrupts are disabled. The call-back function must be terminated by RETF. All registers must be preserved. A stack of at least 32 bytes is provided by **COMMON-ISDN-API.**

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

The call-back function is allowed to use the **COMMON-ISDN-API** operations **CAPI_PUT_MESSAGE**, **CAPI_GET_MESSAGE**, and **CAPI_SET_SIGNAL**. If it does so, it must take into account the fact that interrupts may be enabled by **COMMON-ISDN-API**.

In case of local confirmations (such as LISTEN_CONF), the call-back function may be called before the operation CAPI_PUT_MESSAGE returns control to the application.

8.4.2.2 CAPI_GET_MANUFACTURER

Description

With this operation the application obtains the **COMMON-ISDN-API** (DLL) manufacturer identification. The application provides a 16:16 (segmented) protected-mode pointer to a buffer of 64 bytes in szBuffer. **COMMON-ISDN-API** copies the identification, 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

8.4.2.3 CAPI_GET_VERSION

Description

With this function the application obtains 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 which receives the **COMMON-ISDN-API** major version number: 2
pCAPIMinor	word*	16:16 (segmented) protected-mode pointer to a word which receives the **COMMON-ISDN-API** minor version number: 0
pManufacturerMajor	word*	16:16 (segmented) protected-mode pointer to a word which receives the manufacturer-specific major number
pManufacturerMinor	word*	16:16 (segmented) protected-mode pointer to a word which receives the manufacturer-specific minor number

Return Value

Return	**Comment**
0x0000	No error, version numbers have been copied
All other values	Coded as described in parameter *Info,* class 0x11xx

8.4.2.4 CAPI_GET_SERIAL_NUMBER

Description

With this operation the application obtains the (optional) serial number of **COMMON-ISDN-API**. The application provides a 16:16 (segmented) protected-mode pointer to a string buffer of 8 bytes in szBuffer. **COMMON-ISDN-API** copies the serial number string to this buffer. The serial number, a seven-digit number coded as a zero-terminated ASCII string, is present in this buffer 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

8.4.2.5 CAPI_GET_PROFILE

Description

The application uses this function to determine the capabilities of **COMMON-ISDN-API**. The application provides a 16:16 (segmented) protected-mode pointer to a buffer of 64 bytes in szBuffer. **COMMON-ISDN-API** copies information about implemented features, the number of controllers and supported protocols to this buffer. *CtrlNr* contains the number of the controller (bits 0..6) for which this information is requested. The profile structure retrieved is described at the beginning of Chapter 8.

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 controllers installed is provided to the application.

Return Value

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

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 is typically based on the system calls **open**, **ioctl**, **putmsg**, **getmsg**, and **close**. To register with a device driver, an application opens a stream (*open()*). Applications log off from **COMMON-ISDN-API** using the system call *close()*. Data transfer to and from the driver is accomplished by the calls *putmsg()* and *getmsg()*. Additional information is exchanged using the *ioctl()* system call.

COMMON-ISDN-API uses this standardized driver access mechanism. For this reason, the following specification does not define a complete functional interface (which would not be accepted by UNIX applications, which always are—and must be—file-I/O oriented). Instead, the **COMMON-ISDN-API** system call level interface is introduced, which any UNIX-like application can use to exchange **COMMON-ISDN-API** messages and related data. Of course it is possible to provide a functional interface (as described in Chapter 8.2, for example), but that would not be the appropriate application interface solution for communications applications running on UNIX. The following specification nonetheless provides the complete capabilities of the **COMMON-ISDN-API** access operations used in other operating systems.

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

An application (in **COMMON-ISDN-API** terms) can register with **COMMON-ISDN-API** (CAPI_REGISTER) by opening the device /dev/capi20 and issuing the relevant parameters to the opened device by means of the system call *ioctl()*. Note that the result of this operation is a file handle, not an application ID. Thus in the UNIX environment, the application ID contained in **COMMON-ISDN-API** messages is not used to identify CAPI applications. The only handle valid between the **COMMON-ISDN-API** kernel driver and the application, based on a system call level interface, is a UNIX file handle. To release itself from **COMMON-ISDN-API** (CAPI_RELEASE), an application must simply close the opened device. The **COMMON-ISDN-API** operations CAPI_PUT_MESSAGE and CAPI_GET_MESSAGE are performed by means of the system calls *putmsg()* and *getmsg()*. **COMMON-ISDN-API** need not provide a CAPI_SET_SIGNAL function: instead, applications may use the UNIX signaling and/or waiting mechanism based on file descriptors. This includes waiting on multiple file descriptors (*poll()*); a capability which is not offered by **COMMON-ISDN-API** in other operating systems. All other **COMMON-ISDN-API** operations are realized by means of the system call *ioctl()* with appropriate parameters.

All messages are passed transparently through the UNIX driver interface.

The following data types are used in defining the system call level interface in the UNIX environment:

ushort	16-bit unsigned integer
unsigned	32-bit unsigned integer

8.5.1 Message Operations

8.5.1.1 CAPI_REGISTER

Description

This is the function the application uses to announce its presence to **COMMON-ISDN-API**. The application describes its needs by passing the three parameters *maxLogicalConnection*, *maxBDataBlocks* and *maxBDataLen*.

The parameter *maxLogicalConnection* specifies the maximum number of logical connections this application can maintain concurrently. Any attempt by the application to exceed the logical connection count by accepting or initiating additional connections will result in a connection set-up failure and an error indication from **COMMON-ISDN-API**.

The parameter *maxBDataBlocks* specifies the maximum number of received data blocks that can be reported to the application simultaneously for each logical connection. The number of simultaneously available data blocks has a decisive effect on the data throughput in the system and should be between 2 and 7. At least two data blocks must be specified.

The parameter *maxBDataLen* specifies the maximum size of the application data block to be transmitted and received. Selection of a protocol that requires larger data units, or attempts to transmit or receive larger data units, will result in an error indication from **COMMON-ISDN-API**. The default value for the protocol ISO 7776 (X.75) is 128 octets. **COMMON-ISDN-API** is able to support at least up to 2048 octets.

CAPI_REGISTER	**ioctl(): 0x01**

Implementation

The following code fragment illustrates the UNIX implementation of the **COMMON-ISDN-API** register function:

```
#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        maxLogicalConnection;
        unsigned        maxBDataBlocks;
        unsigned        maxBDataLen;
} rp;
int fd;
struct strioctl strioctl;

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

 /* set registration parameters */
rp.maxLogicalConnection = No. of simultaneous user data connections
rp.maxBDataBlocks = No. of buffered data messages
rp.maxBDataLen = 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 the sake of simplicity, no error checking is shown in the example.

8.5.1.2 CAPI_RELEASE

Description

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

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.

8.5.1.3 CAPI_PUT_MESSAGE

Description

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

CAPI_PUT_MESSAGE	**putmsg()**

Implementation

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

The application places the **COMMON-ISDN-API** message in the ctl part of the *putmsg()* call. The parameters *data* and *data length* of the **DATA_B3_REQ** message must be stored in the data part of *putmsg()*.

Note

The **COMMON-ISDN-API** message is stored in the ctl part of *putmsg()*. For the message **DATA_B3_REQ,** the parameters *data* and *data length* in the ctl part of *putmsg()* are not interpreted by **COMMON-ISDN-API** implementations.

8.5.1.4 CAPI_GET_MESSAGE

Description

With this operation the application retrieves a message from **COMMON-ISDN-API**. The application retrieves each message associated with the specified file descriptor, which is obtained through the 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 must supply sufficient buffers to receive the ctl and data parts of the message. When receiving the **COMMON-ISDN-API** message **DATA_B3_IND**, the message parameters *data* and *data length* are not supported. Instead, the data part of *getmsg()* is used to pass the data.

8.5.2 Other Functions

8.5.2.1 CAPI_GET_MANUFACTURER

Description

With this operation the application obtains the **COMMON-ISDN-API** manufacturer identification. The application provides a buffer which 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 realized 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 specified buffer. The string is always zero-terminated.

8.5.2.2 CAPI_GET_VERSION

Description

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

CAPI_GET_VERSION	**ioctl(): 0x07**

Implementation

This operation is realized 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 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 );
```

On return, the buffer contains four elements:

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

8.5.2.3 CAPI_GET_SERIAL_NUMBER

Description

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

CAPI_GET_SERIAL_NUMBER	**ioctl(): 0x08**

Implementation

This operation is realized 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.

8.5.2.4 CAPI_GET_PROFILE

Description

The application uses this function to determine the capabilities of **COMMON-ISDN-API**. The application provides a buffer of 64 bytes. **COMMON-ISDN-API** copies information about implemented features, the number of controllers and supported protocols to this buffer. *CtrlNr*, which is an input parameter for **COMMON-ISDN-API**, is coded in the first byte of the buffer and contains the number of the controller (bits 0..6) for which this information is requested. The profile structure retrieved is described at the beginning of Chapter 8.

CAPI_GET_PROFILE	**0x09**

Implementation

This operation is realized 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 the number of controllers installed is provided to the application.

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. Scalability and flexibility are therefore considered primary design goals when considering the addition of a new communications subsystem to the NetWare operating system.

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 supplied by different manufacturers. The 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 the COMMON-ISDN-API v2.0 definition, but 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 all the standard COMMON-ISDN-API functions which make 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 functions unique to the NetWare CAPI Manager 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 situated below COMMON-ISDN-API. Each CAPI application and each controller driver is implemented as a separate NLM which registers independently 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 exist 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.

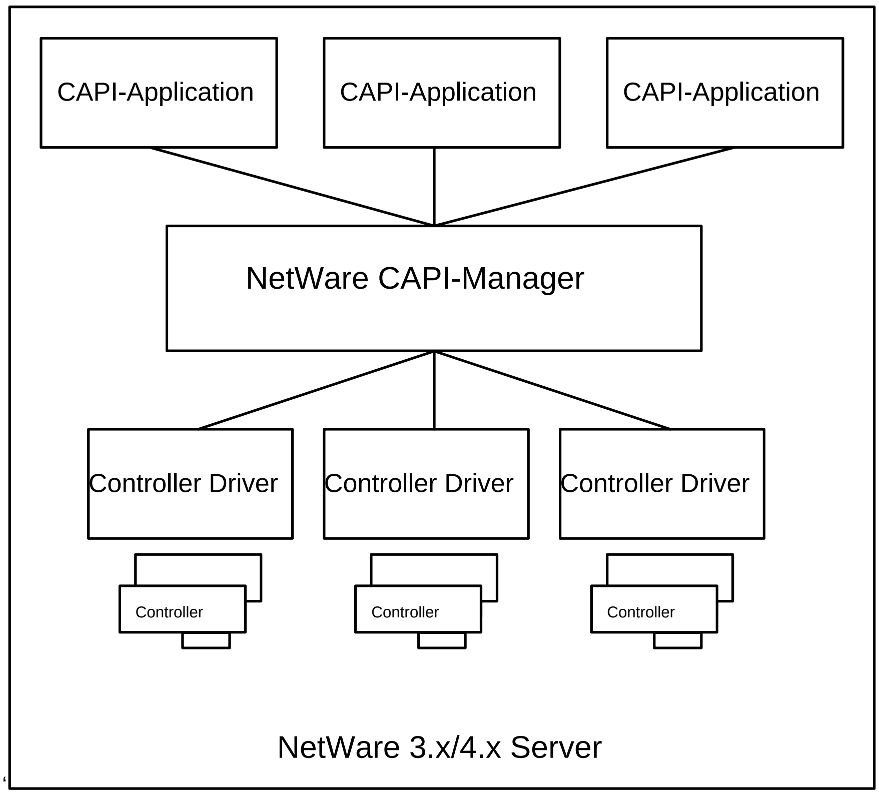

Figure 1: Architectural Overview

Services provided by the CAPI Manager are presented as a set of exported public symbols. To avoid public symbol conflicts within the server environment, the services provided by each controller driver are presented to the NetWare CAPI Manager at driver registration time as a set of entry point addresses. CAPI Manager services include the standard COMMON-ISDN-API function set, auxiliary functions supporting driver registration and de-registration 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 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, incoming message signaling is required by the NetWare CAPI Manager. The CAPI_Register function defines additional signal parameters, which must be provided by the application in order to register successfully. Applications are not permitted to poll for incoming 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.

For a complete definition of the auxiliary and driver functions, please refer to the NetWare CAPI Manager and CAPI Driver Specification. 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 the NetWare Environment:

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

Data Type Conventions in NetWare environment:

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

8.6.1.1 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, the message buffer size, the number of data buffers and the data buffer size required by the application. The message buffer size is normally calculated according to following formula:

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

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

Applications which maintain a CLIB process context should select Local Semaphore signaling in the signalType parameter, and supply a previously allocated Local Semaphore handle as the signalHandle parameter. The application's receiving process can then wait on the local semaphore. When an incoming message is available, the CAPI driver will signal the local semaphore, causing the application process to wake up and retrieve a message by calling the *CAPI_GetMessage* function.

Applications which do not maintain a CLIB process context should select direct call-back signaling in the signalType parameter, supply a pointer to an application-resident notification function as the signalHandle parameter, and pass an application-defined context value as the signalContext parameter. When an incoming message is available, **COMMON-ISDN-API** will call the specified application notification function, presenting the application context value. The application then calls 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 maintain concurrently. Any attempt by the application 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 of 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 blocks must be specified.
dataBlockLen	Specifies the maximum size of a B channel data unit which can be transmitted and received. Selection of a protocol that requires larger data units, or 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 incoming message signaling mechanism selected by the application. The signaling mechanism is used by the driver to notify the application when incoming 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 to 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 CAPI. It may be used by an application to reference internal data structures etc. during the 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 as-signed
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 incoming 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 setting the signalType parameter in *CAPI_Register* to SIGNAL_TYPE_CALLBACK. Note that non-system-level applications should always use local semaphores for receive message notification by setting the signalType parameter in *CAPI_Register* to SIGNAL_TYPE_LOCAL_SEMAPHORE.

Each time the *CAPI_ReceiveNotify* function is called, it should in turn call *CAPI_GetMessage* to retrieve 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 in the *CAPI_Register* function. This value is meaningful only to the application, as an internal data structure pointer, for example.

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.

8.6.1.2 CAPI_Release

Description

Applications use *CAPI_Release* to log off 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 assigned by the function CAPI_Register

Return Value

Return Value	Comment
0x0000	Application successfully released
All other values	Coded as described in parameter *Info,* class 0x11xx

8.6.1.3 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 assigned by the function CAPI_Register
pCAPIMessage	Pointer 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.

8.6.1.4 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 block returned by this function are valid until the same application calls *CAPI_GetMessage* again. Applications which process the message asynchronously or need to maintain the message beyond the next call to *CAPI_GetMessage* must make a local copy of the message.

Function call

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

Parameter	Comment
ApplID	Application identification number assigned by 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 ppCAPIMessage 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

8.6.2.1 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 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

8.6.2.2 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 which will receive the **COMMON-ISDN-API** major version number: 0x0002
pCAPIMinor	Pointer to a WORD which will receive the **COMMON-ISDN-API** minor version number: 0x0000
pManufacturerMajor	Pointer to a WORD which will receive the manufacturer-specific major number
pManufacturerMinor	Pointer to a WORD which will receive the manufacturer-specific minor number
pManagerMajor	Pointer to a WORD which will receive the CAPI Manager major version number
pManagerMinor	Pointer to a WORD which will receive the CAPI Manager minor version number

Return Value

Return	**Comment**
0x0000	No error, version numbers have been copied
All other values	Coded as described in parameter *Info*, class 0x11xx

8.6.2.3 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

8.6.2.4 CAPI_GetProfile

Description

The application uses this function to get information on the ISDN controller’s capabilities from **COMMON-ISDN-API**. *Buffer* is a pointer to a buffer of 64 bytes. **COMMON-ISDN-API** copies information to this buffer about implemented features, the number of controllers and supported protocols. *Controller* contains the number of the controller (bit 0..6) for which this information is requested. The profile structure retrieved is described at the beginning of Chapter 8.

```
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 returned 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

8.7 Windows NT (Application Level)

In the operating system Windows NT 3.x / 4.x, the **COMMON-ISDN-API** services are provided via a DLL (Dynamic Link Library).

Windows-based applications (32-bit) can use the DLL mechanism as described in Chapter **8: Specifications for Commercial Operating Systems**, Subclause **8.18: Windows XP 32bit (Application Level)**, without modification.

8.8 Windows NT (Device Driver Level)

For kernel-mode applications, **COMMON-ISDN-API** 2.0 must be implemented as kernel-mode device driver. The interface to such a kernel-mode device driver in Windows NT is based on I/O request packets (IRPs), which can be sent to the driver by either kernel-mode or user-mode applications.

COMMON-ISDN-API can be accessed as described in Chapter **8: Specifications for Commercial Operating Systems**, Subclause **8.20: Windows XP (Device Driver Level)**, without modification.

8.9 Windows 95 (Application Level)

Under the operating system Windows 95, three types of user-mode applications can access **COMMON-ISDN-API:**

- DOS-based applications
- Windows 3.x-based applications (16-bit)
- Windows 95-based applications (32-bit)

Each of these application types is able to use **COMMON-ISDN-API.**

8.9.1 DOS-based Applications

DOS-based applications continue to use the software interrupt mechanism of **COMMON-ISDN-API** as described in Chapter **8: Specifications for Commercial Operating Systems**, Subclause **8.1:** **MS-DOS**. The implementation must also support a FAR CALL (after pushing flags) to the entry address of **COMMON-ISDN-API**.

8.9.2 Windows 3.x-based Applications (16-bit)

Windows-based applications (16-bit) use the DLL mechanism of **COMMON-ISDN-API** as described in Chapter **8: Specifications for Commercial Operating Systems**, Subclause **8.2:** **Windows (Application Level)**, without modification. The **CAPI20.DLL** provided in Windows 95 has an identical interface to applications as that in Windows 3.x.

8.9.3 Windows 95-based Applications (32-bit)

Windows-based applications (32-bit) can use the DLL mechanism as described in Chapter **8: Specifications for Commercial Operating Systems**, Subclause **8.18:** **Windows XP 32bit (Application Level),** without modification. The **CAPI2032.DLL** provided in Windows 95 has an identical interface to applications as that in Windows NT.

8.10 Windows 95 (Device Driver Level)

COMMON-ISDN-API for Windows 95 must be implemented as a Virtual Device Driver (**VxD**). The interface to such a kernel-mode driver consists of exported **Virtual Device Services** for other virtual devices and a **Virtual Device API** for protected-mode applications (16 or 32-bit) which access the features of the virtual device (i.e. **CAPI20.DLL / CAPI2032.DLL**). Both interfaces exchange information in CPU registers. The exchange mechanism described in Chapter **8: Specifications for Commercial Operating Systems**, Subclause **8.1: MS-DOS** is used, and adapted to the 32-bit environment where necessary. The CAPI VxD shall also hook the software interrupt F1 to offer **COMMON-ISDN-API** to DOS-based applications.

User-mode applications shall not use the device driver level interface directly. Instead, they must use the specified access methods (i.e. software interrupt or DLL mechanism) to access **COMMON-ISDN-API**.

Architectural Overview:

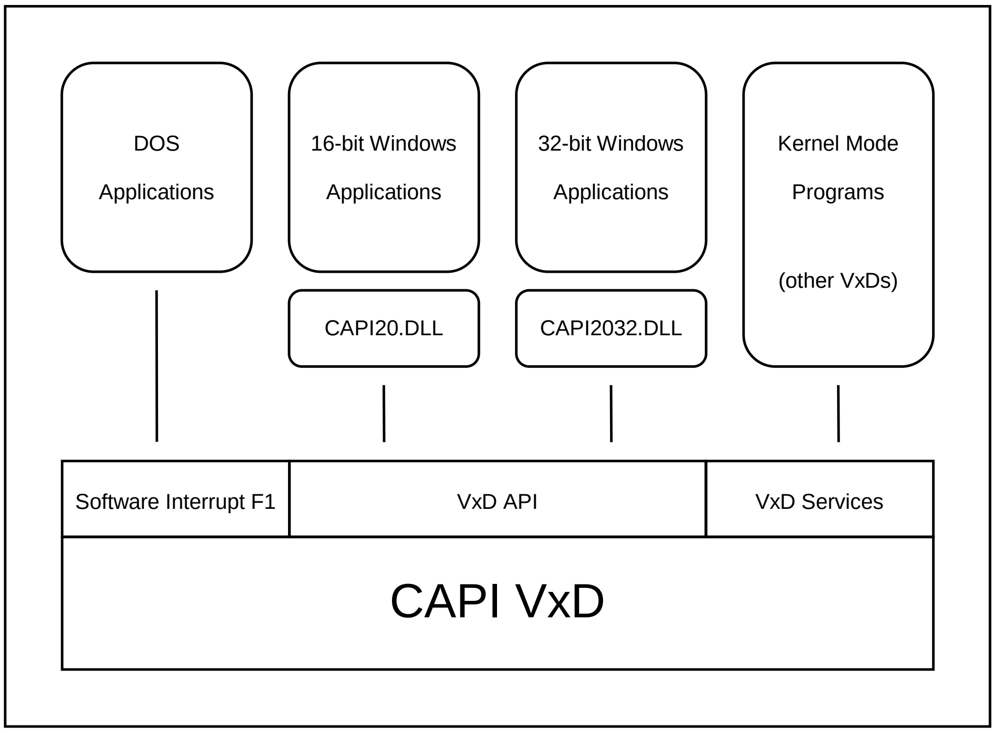
DOS Applications
16-bit Windows Applications
32-bit Windows Applications
Kernel Mode Programs (other VxDs)
CAPI20.DLL
CAPI2032.DLL
Software Interrupt F1
VxD API
VxD Services
CAPI VxD

Figure 2: Architectural Overview

Virtual Device Services can be used by other virtual devices by including an appropriate header file which contains the service table declaration. A virtual device calls the CAPI VxD's services using the **VxDcall** macro. To verify the availability of CAPI VxD services, the calling virtual device must attempt to call the **Get_Version** service of CAPI VxD. If the CAPI VxD has not been loaded, the VMM sets the carry flag and returns zero in the **AX** register. The virtual device which provides **COMMON-ISDN-API** exports *one* CAPI-specific service, namely an access to the message exchange functions described in this chapter. Information is exchanged directly in CPU registers.

The **Virtual Device API** is used by **CAPI20.DLL** and **CAPI2032.DLL**. These DLLs retrieve an entry point address for the **Virtual Device API** procedure for their virtual machine. The CAPI VxD can obtain the calling application's register values via the **Client_Reg_Struc** structure.

The CAPI VxD provides synchronous services. If any **COMMON-ISDN-API** service is entered while an asynchronous interrupt is being processed, the value **0x1107** (internal busy condition) is returned in the **AX** register.

Every VxD has a unique device ID. The CAPI VxD has the device ID **0x3215**.

Service table declaration from CAPI VxD:

```
VCAPID_DEVICE_ID EQU 3215h
Begin_Service_Table    VCAPID
    VCAPID_Service     VCAPID_Get_Version, LOCAL
    VCAPID_Service     VCAPID_MessageOperations, LOCAL
End_Service_Table            VCAPID
```

In this section, the term *pointer* has two meanings: with reference to the 16-bit Virtual Device API, *pointer* refers to a 16:16 (segmented) pointer to a memory location; where the 32-bit Virtual Device API is concerned, a *pointer* is a 0:32 near flat pointer to a memory location.

8.10.1 Message Operations

8.10.1.1 CAPI_REGISTER

Description

This is the function the application uses to announce its presence to **COMMON-ISDN-API**. The application describes its needs by passing the four registers *ECX*, *EDX*, *ESI* and *EDI*.

For a typical application, the amount of memory required should be calculated by the following formula:

ECX = 1024 + (1024 * EDX)

In the *EDX* register, the application specifies the maximum number of logical connections this application can maintain concurrently. Any attempt by the application to exceed the logical connection count by accepting or initiating additional connections will result in a connection set-up failure and an error indication from **COMMON-ISDN-API**.

In the *ESI* register, the application specifies the maximum number of received data blocks that can be reported to the application simultaneously for each logical connection. The number of simultaneously available data blocks has a decisive effect on the data throughput in the system and should be between 2 and 7. At least two data blocks must be specified.

In the *EDI* register, the application specifies the maximum size of the application data block to be transmitted and received. Selection of a protocol that requires larger data units, or attempts to transmit or receive larger data units, will result in an error indication from **COMMON-ISDN-API**. The default value for the protocol ISO 7776 (X.75) is 128 octets. **COMMON-ISDN-API** is able to support at least up to 2048 octets.

The application number is returned in *AX*. In the event of an error, the value 0 is returned in *AX*, and the cause of the error is indicated in *BX*.

CAPI_REGISTER	**0x01**

Parameter	Comment
AH	Version number 20 (0x14)
AL	Function code 0x01
ECX	Size of message buffer
EDX	Maximum number of Layer 3 connections
ESI	Number of B3 data blocks available simultaneously
EDI	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 at a time and use the standard protocols, the following register assignments are recommended:

ECX = 2048, EDX = 1, ESI = 7, EDI = 128

The resulting memory requirement is 2944 bytes.

8.10.1.2 CAPI_RELEASE

Description

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

CAPI_RELEASE	**0x02**

Parameter	**Comment**
AH	Version number 20 (0x14)
AL	Function code 0x02
EDX	Application number

Return Value

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

8.10.1.3 CAPI_PUT_MESSAGE

Description

With this function the application transfers a message to **COMMON-ISDN-API**. A pointer to the message is passed in the *EBX* register. The application is identified by the application number in the *EDX* register. Any errors that occur are returned in *AX*.

CAPI_PUT_MESSAGE	**0x03**

Parameter	**Comment**
AH	Version number 20 (0x14)
AL	Function code 0x03
EBX	Pointer to message
EDX	Application number

Return Value

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

Note

After returning from the **CAPI_PUT_MESSAGE** call, the application can re-use the memory area of the message. The message is not modified by **COMMON-ISDN-API**.

8.10.1.4 CAPI_GET_MESSAGE

Description

With this function the application retrieves a message from **COMMON-ISDN-API**. The application can only retrieve messages intended for the specified application number. A pointer to the message is passed in the *EBX* register. If there is no message queued for the application, the function returns immediately. Register *AX* contains the corresponding error value. The application is identified by the application number in the *EDX* register. Any errors that occur are returned in *AX*.

CAPI_GET_MESSAGE	**0x04**

Parameter	**Comment**
AH	Version number 20 (0x14)
AL	Function code 0x04
EDX	Application number

Return Value

Return	**Value**	**Comment**
AX	0x0000 <> 0	No error Coded as described in parameter *Info*, class 0x11xx
EBX		Pointer to message, if available

Note

The message may be made invalid by the next **CAPI_GET_MESSAGE** call.

8.10.2 Other Functions

8.10.2.1 CAPI_SET_SIGNAL

Description

The application can use this function to activate the use of the synchronous (non-interrupt) call-back function. A pointer to a call-back function is specified in the *EBX* register. The signaling function can be deactivated by a **CAPI_SET_SIGNAL** call with the value 0 in the *EBX* register. The application is identified by the application number in the *EDX* register. Any errors that occur are returned in the *AX* register.

CAPI_SET_SIGNAL	**0x05**

Parameter	**Comment**
AH	Version number 20 (0x14)
AL	Function code 0x05
EDX	Application number
EDI	Parameter passed to call-back function
EBX	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 always called in a synchronous environment, i.e. outside any hardware interrupt condition. It is called as a NEAR function in a 32-bit environment, so it must return by a RET. If used via the Virtual Device API (i.e. not from another Virtual Device Driver), the context of the calling VM is available.

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

- any message is queued in the application's message queue,
- an announced busy condition is cleared, or
- an announced queue full-condition is cleared.

Interrupts are enabled. The call-back function must be terminated by RET. All registers must be preserved.

COMMON-ISDN-API does not call this function recursively. If necessary, the call-back function is called again after it returns control to **COMMON-ISDN-API**.

The call-back function is allowed to use the **COMMON-ISDN-API** operations **CAPI_PUT_MESSAGE**, **CAPI_GET_MESSAGE**, and **CAPI_SET_SIGNAL**.

In the case of local confirmations (such as LISTEN_CONF), the call-back function may be called before the operation **CAPI_PUT_MESSAGE** returns control to the application.

Registers *EDX* and *EDI* are passed to the call-back function with the same values as the corresponding parameters of **CAPI_SET_SIGNAL**.

8.10.2.2 CAPI_GET_MANUFACTURER

Description

By calling this function the application obtains the **COMMON-ISDN-API** manufacturer identification. The application provides a pointer to a data area of 64 bytes in the *EBX* register. 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
EBX	Pointer to buffer
ECX	Number of Controller. If 0, the manufacturer identification of the software components is returned.

Return Value

Return	**Value**	**Comment**
AX	0x0000 <> 0	No error Coded as described in parameter *Info,* class 0x11xx
EBX		Buffer contains manufacturer identification as an ASCII string, terminated by a 0 byte.

8.10.2.3 CAPI_GET_VERSION

Description

With this function the application obtains 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
ECX	Number of Controller. If 0, the version of the software components is returned.

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

8.10.2.4 CAPI_GET_SERIAL_NUMBER

Description

With this function the application obtains the (optional) serial number of **COMMON-ISDN-API.** The application provides a pointer to a data area of 8 bytes in register *EBX*. The serial number, a seven-digit number coded as a zero-terminated ASCII string, is present in this data area 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
EBX	Pointer to buffer
ECX	Number of Controller. If 0, the serial number of the software components is returned.

Return Value

Return	Value	Comment
AX	0x0000 <> 0	No error Coded as described in parameter *Info*, class 0x11xx
EBX		Pointer to the (optional) serial number in plain text in the form of a 7-digit number. If no serial number is used, a 0 byte is written at the first position in the buffer. The end of the serial number is indicated by a 0 byte.

8.10.2.5 CAPI_GET_PROFILE

Description

The application uses this function to determine the capabilities of **COMMON-ISDN-API**. Register *EBX* must contain a pointer to a data area of 64 bytes. **COMMON-ISDN-API** copies information about implemented features, the number of controllers and supported protocols to this buffer. Register *ECX* contains the number of the controller (bits 0..6) for which this information is requested. The profile structure retrieved is described at the beginning of Chapter 8.

CAPI_GET_PROFILE	**0xF3**

Parameter	**Comment**
AH	Version number 20 (0x14)
AL	Function code 0xF3
ECX	Controller number (if 0, only the number of controllers is re-turned)
EBX	Pointer to buffer

Return Value

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

Note

Applications must ignore unknown bits in the profile structure since this function may be extended. **COMMON-ISDN-API** sets every reserved field to 0.

8.10.2.6 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	

8.11 Windows 95 DeviceIoControl

COMMON-ISDN-API can also be accessed by using DeviceIoControl operations. The definition of this interface is as close as possible to that of the Windows NT DeviceIoControl interface. Since not all Windows NT device operations are available under Windows 95, however, this interface cannot be defined as completely compatible with the Windows NT definition.

The following DEVICE_CONTROL codes are defined for **COMMON-ISDN-API** functions:

```
/*
*       the common device type code for CAPI20 conforming drivers
*/
#define FILE_DEVICE_CAPI20 0x8001

/*
*       DEVICE_CONTROL codes
*/
#define CAPI_CTL_BASE                      0x800
#define CAPI_CTL_REGISTER                  ( CAPI_CTL_BASE + 0x0001 )
#define CAPI_CTL_RELEASE                   ( CAPI_CTL_BASE + 0x0002 )
#define CAPI_CTL_PUT_MESSAGE               ( CAPI_CTL_BASE + 0x0003 )
#define CAPI_CTL_GET_MESSAGE               ( CAPI_CTL_BASE + 0x0004 )
#define CAPI_CTL_GET_MANUFACTURER          ( CAPI_CTL_BASE + 0x0005 )
#define CAPI_CTL_GET_VERSION               ( CAPI_CTL_BASE + 0x0006 )
#define CAPI_CTL_GET_SERIAL                ( CAPI_CTL_BASE + 0x0007 )
#define CAPI_CTL_GET_PROFILE               ( CAPI_CTL_BASE + 0x0008 )
#define CAPI_CTL_WAIT_MESSAGE              ( CAPI_CTL_BASE + 0x0009 )
#define CAPI_CTL_MANUFACTURER              ( CAPI_CTL_BASE + 0x00ff )




/*
*       The wrapped control codes as required by the system.
*       Note: while use of these macros is not required,
*       no other control parameters are allowed for the
*       DeviceIoControl control codes.
*/
#define CAPI_CTL_CODE(function,method) \
        CTL_CODE(FILE_DEVICE_CAPI20,function,method,FILE_ANY_ACCESS)

#define IOCTL_CAPI_REGISTER \
        CAPI_CTL_CODE(CAPI_CTL_REGISTER, METHOD_BUFFERED)

#define IOCTL_CAPI_RELEASE \
        CAPI_CTL_CODE(CAPI_CTL_RELEASE, METHOD_BUFFERED)

#define IOCTL_CAPI_GET_MANUFACTURER\
        CAPI_CTL_CODE(CAPI_CTL_GET_MANUFACTURER, METHOD_BUFFERED)

#define IOCTL_CAPI_GET_VERSION \
        CAPI_CTL_CODE(CAPI_CTL_GET_VERSION, METHOD_BUFFERED)

#define IOCTL_CAPI_GET_SERIAL \
        CAPI_CTL_CODE(CAPI_CTL_GET_SERIAL, METHOD_BUFFERED)
```

```
#define IOCTL_CAPI_GET_PROFILE \
            CAPI_CTL_CODE(CAPI_CTL_GET_PROFILE, METHOD_BUFFERED)

#define IOCTL_CAPI_MANUFACTURER \
            CAPI_CTL_CODE(CAPI_CTL_MANUFACTURER, METHOD_BUFFERED)

#define IOCTL_CAPI_PUT_MESSAGE \
            CAPI_CTL_CODE(CAPI_CTL_PUT_MESSAGE, METHOD_IN_DIRECT)

#define IOCTL_CAPI_GET_MESSAGE \
            CAPI_CTL_CODE(CAPI_CTL_GET_MESSAGE, METHOD_OUT_DIRECT)

#define IOCTL_CAPI_WAIT_MESSAGE \
            CAPI_CTL_CODE(CAPI_CTL_WAIT_MESSAGE, METHOD_BUFFERED)
```

CAPI20-specific return values are mapped to Win32 error codes according to the following table. The error code is returned by GetLastError( ) after a failure of DeviceIoControl( ).

Info	Win32 Error Code
0x1001	ERROR_TOO_MANY_SESSIONS
0x1002	ERROR_INVALID_PARAMETER
0x1003	
0x1004	ERROR_INSUFFICIENT_BUFFER
0x1005	ERROR_NOT_SUPPORTED
0x1006	
0x1007	ERROR_NETWORK_BUSY
0x1008	ERROR_NOT_ENOUGH_MEMORY
0x1009	
0x100a	ERROR_SERVER_DISABLED
0x100b	ERROR_SERVER_NOT_DISABLED
0x1101	ERROR_INVALID_HANDLE
0x1102	ERROR_INVALID_FUNCTION
0x1103	ERROR_TOO_MANY_CMDS
0x1104	ERROR_IO_PENDING
0x1105	ERROR_IO_DEVICE
0x1106	STATUS_INVALID_PARAMETER
0x1107	ERROR_BUSY
0x1108	ERROR_NOT_ENOUGH_MEMORY
0x1109	
0x110a	ERROR_SERVER_DISABLED
0x110b	ERROR_SERVER_NOT_DISABLED

In Windows 95, all communications between a device and an application are associated with a file handle. For this reason, a file handle is used instead of the application ID to link an application to the CAPI20 device. Any application IDs contained in **COMMON-ISDN-API** messages are therefore ignored.

In the following, the interface between the application and the **COMMON-ISDN-API** device driver is described by means of Win32 functions.

8.11.1 Message Operations

8.11.1.1 CAPI_REGISTER

Description

This is the function the application uses to announce its presence to **COMMON-ISDN-API**. The application describes its needs by passing the four parameters *MessageBufferSize*, *maxLogicalConnection*, *maxBDataBlocks* and *maxBDataLen*.

For a typical application, the amount of memory required should be calculated by the following formula:

MessageBufferSize = 1024 + (1024 * maxLogicalConnection)

The parameter *maxLogicalConnection* specifies the maximum number of logical connections this application can maintain concurrently. Any attempt by the application to exceed the logical connection count by accepting or initiating additional connections will result in a connection set-up failure and an error indication from **COMMON-ISDN-API**.

The parameter *maxBDataBlocks* specifies the maximum number of received data blocks that can be reported to the application simultaneously for each logical connection. The number of simultaneously available data blocks has a decisive effect on the data throughput in the system and should be between 2 and 7. At least two data blocks must be specified.

The parameter *maxBDataLen* specifies the maximum size of the application data block to be transmitted and received. Selection of a protocol that requires larger data units, or attempts to transmit or receive larger data units, will result in an error indication from **COMMON-ISDN-API**. The default value for the protocol ISO 7776 (X.75) is 128 octets. **COMMON-ISDN-API** is able to support at least up to 2048 octets.

CAPI_REGISTER	**CAPI_CTL_REGISTER**

Implementation

For the CAPI_REGISTER operation, the application must first obtain a handle to the **COMMON-ISDN-API** device using the Win32 CreateFile() function, then send a CAPI_CTL_REGISTER to the **COMMON-ISDN-API** device. CAPI_REGISTER passes the following data structure to the driver:

```
struct capi_register_params {
        WORD MessageBufferSize,
        WORD maxLogicalConnection,
        WORD maxBDataBlocks,
        WORD maxBDataLen
    };
```

Only one CAPI_CTL_REGISTER may be sent with a given handle before a CAPI_CTL_RELEASE is sent. If an application program wants to register as more than one **COMMON-ISDN-API** application, it must obtain several handles using CreateFile() and send one CAPI_CTL_REGISTER with each handle. The FILE_FLAG_OVERLAPPED option for fdwAttrsAndFlags must be set for proper operation.

Example:

```
    capi_handle = CreateFile( "\\\\.\\CAPI20",
                    GENERIC_READ | GENERIC_WRITE,
                    0,
                    NULL,
                    OPEN_EXISTING,
                    FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,
                    NULL );

    r.MessageBufferSize = MessageBufferSize;
    r.maxLogicalConnection = maxLogicalConnection;
    r.maxBDataBlocks = maxBDataBlocks;
    r.maxBDataLen = maxBDataLen;

    ret = DeviceIoControl( capi_handle,
                    CAPI_CTL_REGISTER,
                     ( PVOID ) &r,
                    sizeof( r ),
                    NULL,
                    0,
                    &ret_bytes,
                    NULL );
```

8.11.1.2 CAPI_RELEASE

Description

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

CAPI_RELEASE	**CAPI_CTL_RELEASE**

Implementation

A CAPI_RELEASE can be performed in one of two ways. If the same handle is to be used again, a CAPI_CTL_RELEASE must be sent. If the handle is no longer needed, the **COMMON-ISDN-API** device may be simply closed using CloseHandle.

Example:

```
ret = DeviceIoControl( capi_handle,
                CAPI_CTL_RELEASE,
                NULL,
                0,
                NULL,
                0,
                &ret_bytes,
                NULL );
CloseHandle( capi_handle );
```

8.11.1.3 CAPI_PUT_MESSAGE

Description

With this operation the application transfers a message to **COMMON-ISDN-API**. The application identifies itself by a file handle.

CAPI_PUT_MESSAGE	**CAPI_CTL_PUT_MESSAGE**

Implementation

The CAPI_PUT_MESSAGE function is performed using a CAPI_CTL_PUT_MESSAGE DeviceIoControl.

With this DeviceIoControl operation, one data buffer is sent to the CAPI20 device driver. This buffer must contain the message *and*, in the case of a DATA_B3_REQ message, the associated data. The data (if applicable) must be placed in the buffer immediately following the message.

```
ret = DeviceIoControl( capi_handle,
                CAPI_CTL_PUT_MESSAGE,
                ( PVOID )msg,                /* buffer for message + data */
                msg_length,           /* length of message + data */
                NULL,
                0,
                &ret_bytes,
                NULL );
```

This operation is completed immediately, without waiting for any network event (in normal CAPI_PUT_MESSAGE operation).

The buffer can be re-used by the application as soon as the operation is completed.

8.11.1.4 CAPI_GET_MESSAGE

Description

With this operation the application retrieves a message from **COMMON-ISDN-API**. The application retrieves each message associated with the specified file handle obtained in the **CAPI_REGISTER** operation.

CAPI_GET_MESSAGE	**CAPI_CTL_GET_MESSAGE**

Implementation

The CAPI_GET_MESSAGE function is performed using the CAPI_CTL_GET_MESSAGE DeviceIoControl operation.

With the CAPI_CTL_GET_MESSAGE DeviceIoControl operation, one data buffer is received from the CAPI20 device driver. This buffer contains the message *and*, in the case of a DATA_B3_IND message, the associated data. The data (if applicable) is located in the buffer immediately following the message.

CAPI_CTL_GET_MESSAGE supports overlapped operation. If it returns TRUE, the number of bytes in the message retrieved is available.

If the buffer provided by the application is to small to hold the message and the data, an ERROR_INSUFFICIENT_BUFFER error is returned and no message is retrieved.

Example:

```
ret = DeviceIoControl( capi_handle,
                CAPI_CTL_GET_MESSAGE,
                NULL,
                0,
                ( PVOID )buffer,         /* buffer for message + data */
                buffer_size,             /* length of message + data */
                &ret_bytes,
                &0_read );

if ( ret == TRUE ) {
    /* operation succeeded immediately */
    /* ret_bytes contains the number of bytes accepted */
            ;
} else if ( GetLastError() == ERROR_IO_PENDING ) {
    /* operation pending, must wait for completion */
    WaitForSingleObject( result.hEvent, INFINITE );
    ret = GetOverlappedResult( capi_handle, &result, &ret_bytes, TRUE );
    if (ret == TRUE) {
            /* operation successfully completed now */
            /* ret_bytes contains the number of bytes accepted */
            ;
    } else {
            /* sorry, failure */
            ...
    }
} else {
    /* operation failed immediately */
    ...
}
```

8.11.1.5 CAPI_SET_SIGNAL

There is no CAPI_SET_SIGNAL function. Asynchronous signaling of a received message is implicit in the completion of the CAPI_CTL_GET_MESSAGE operation.

8.11.2 Other Functions

8.11.2.1 CAPI_GET_MANUFACTURER

Description

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

CAPI_GET_MANUFACTURER	**CAPI_CTL_GET_MANUFACTURER**

Implementation

The manufacturer identification is read from the **COMMON-ISDN-API** driver using CAPI_CTL_GET_MANUFACTURER. A buffer of 64 bytes must be provided by the application. The manufacturer identification is returned as a zero-terminated ASCII string. The controller number 0 returns the manufacturer name of the CAPI20 device driver; other controller numbers return the manufacturer of the corresponding controller.

```
DWORD   controller;             /* 32-bit */
char    manufacturer[64];

controller = 0;                 /* to retrieve the manufacturer of the device driver */
ret = DeviceIoControl( capi_handle,
            CAPI_CTL_GET_MANUFACTURER,
            ( PVOID ) &controller,
            sizeof ( controller ),
            ( PVOID ) manufacturer,
            sizeof ( manufacturer ),
            &ret_bytes,
            ( POVERLAPPED ) NULL );
```

8.11.2.2 CAPI_GET_VERSION

Description

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

CAPI_GET_VERSION	**CAPI_CTL_GET_VERSION**

Implementation

The version of **COMMON-ISDN-API** is read using CAPI_CTL_GET_VERSION. A buffer with the following structure must be provided by the application:

```
struct capi_version_params {
    WORD CAPIMajor;                  /* 16-bit */
    WORD CAPIMinor;
    WORD ManufacturerMajor;
    WORD ManufacturerMinor;
} buf;
```

The controller number 0 returns the version info of the CAPI20 device driver; other controller numbers return the version of the corresponding controller.

```
DWORD controller;             /* 32-bit */

controller = 0;               /* to retrieve the version of the device driver */
ret = DeviceIoControl( capi_handle,
                CAPI_CTL_GET_VERSION,
                ( PVOID ) &controller,
                sizeof ( controller ),
                ( PVOID ) &buf,
                sizeof ( buf ),
                &ret_bytes,
                ( POVERLAPPED ) NULL );
```

8.11.2.3 CAPI_GET_SERIAL_NUMBER

Description

With this operation the application obtains the (optional) serial number of **COMMON-ISDN-API**. The application provides a buffer of 8 bytes. **COMMON-ISDN-API** copies the serial number string to this buffer. The serial number, a seven-digit number coded as a zero-terminated ASCII string, is present in this buffer after the function has returned.

CAPI_GET_SERIAL_NUMBER	**CAPI_CTL_GET_SERIAL_NUMBER**

Implementation

With CAPI_CTL_GET_SERIAL_NUMBER the **COMMON-ISDN-API** serial number can be obtained from the driver. A buffer of 8 bytes must be provided by the application. The serial number is returned in this buffer as a zero-terminated ASCII string. The controller number 0 returns the serial number of the CAPI20 device driver; other controller numbers return the serial number of the corresponding controller.

```
char        serial[8];
DWORD       controller;                 /* 32-bit */

controller = 0;                         /* to retrieve the serial number of the device driver */
ret = DeviceIoControl( capi_handle,
                CAPI_CTL_GET_SERIAL_NUMBER,
                ( PVOID ) &controller,
                sizeof ( controller ),
                ( PVOID ) serial,
                sizeof ( serial ),
                &ret_bytes,
                ( POVERLAPPED ) NULL );
```

8.11.2.4 CAPI_GET_PROFILE

Description

The application uses this function to determine the capabilities of **COMMON-ISDN-API**. **COMMON-ISDN-API** copies information about implemented features, the number of controllers and supported protocols to the buffer *profile*. The double-word *controller* contains the number of the controller (bit 0..6) for which this information is requested. The profile structure retrieved is described at the beginning of Chapter 8.

CAPI_GET_PROFILE	**CAPI_CTL_GET_PROFILE**

Implementation

The **COMMON-ISDN-API** capabilities can be obtained from the driver by this DeviceIoControl. The application must provide a buffer formatted according to the COMMON-ISDN-API profile structure in the *profile* parameter. This buffer is filled in with the appropriate values by the DeviceIoControl call.

```
char        profile[64];
DWORD       controller;                 /* 32-bit */

controller = 1;                         /* to retrieve the profile of controller number one */
ret = DeviceIoControl( capi_handle,
                CAPI_CTL_GET_PROFILE,
                ( PVOID ) &controller,
                sizeof ( controller ),
                ( PVOID )profile,
                sizeof ( profile ),
                &ret_bytes,
                ( POVERLAPPED ) NULL );
```

8.12 Windows 98 (Application Level)

Under the operating system Windows 98, three types of user-mode applications can access **COMMON-ISDN-API:**

- DOS-based applications,
- Windows 3.x-based applications (16-bit, Win3.x), and
- Win32-based applications (32-bit, Windows 95 / Windows NT).

Each of these application types is able to use **COMMON-ISDN-API**.

8.12.1 DOS-based Applications

DOS-based applications continue to use the software interrupt mechanism of **COMMON-ISDN-API** as described in Chapter **8: Specifications for Commercial Operating Systems**, Subclause **8.1:** MS-DOS. The implementation must also support a FAR CALL (after pushing flags) to the entry address of **COMMON-ISDN-API**.

8.12.2 Windows 3.x-based Applications (16-bit)

Windows-based applications (16-bit) use the DLL mechanism of **COMMON-ISDN-API** as described in Chapter **8: Specifications for Commercial Operating Systems**, Subclause **8.2:** **Windows 3.x (Application Level),** without modification. The **CAPI20.DLL** provided in Windows 98 has the identical interface to applications as that in Windows 3.x.

8.12.3 Win32-based Applications (32-bit)

Windows-based applications (32-bit) can use the DLL mechanism as described in Chapter **8: Specifications for Commercial Operating Systems**, Subclause **8.18:** **Windows XP 32bit (Application Level),** without modification.

8.13 Windows 98 (Device Driver Level)

8.13.1 Windows 95-based Virtual Device Driver (VxD)

Windows 95-based Virtual Device Drivers (VxD) use the interface of **COMMON-ISDN-API** as described in Chapter **8: Specifications for Commercial Operating Systems**, Subclause **8.10: Windows 95 (Device Driver Level)**, without modification.

8.13.2 Win32 Driver Model-based Device Driver (WDM)

Win32 Driver Model-based device drivers (WDM) use the interface of **COMMON-ISDN-API** as described in Chapter **8: Specifications for Commercial Operating Systems**, Subclause **8.20: Windows XP (Device Driver Level)** without modification.

8.14 Windows 2000 (Application Level)

In the operating system Windows XP 32bit, the **COMMON-ISDN-API** services are provided via a DLL (Dynamic Link Library).

Windows-based applications (32-bit) can use the DLL mechanism as described in Chapter **8: Specifications for Commercial Operating Systems**, Subclause **8.18: Windows XP 32bit (Application Level)**, without modification

.

8.15 Windows 2000 (Device Driver Level)

For kernel-mode applications, **COMMON-ISDN-API** 2.0 must be implemented as kernel-mode device driver. The interface to such a kernel-mode device driver in Windows NT is based on I/O request packets (IRPs), which can be sent to the driver by either kernel-mode or user-mode applications.

COMMON-ISDN-API can be accessed as described in Chapter **8: Specifications for Commercial Operating Systems**, Subclause **8.20: Windows XP (Device Driver Level)**, without modification.

8.16 Linux

Under the operating system Linux the **COMMON-ISDN-API** services are provided via a *(shared) library*. The interface between applications and **COMMON-ISDN-API** is realized as a function interface. An application can issue **COMMON-ISDN-API** function calls to perform **COMMON-ISDN-API** operations.

The functions are exported under following names:

```
capi20_register
capi20_release
capi20_put_message
capi20_get_message
capi20_waitformessage
capi20_get_manufacturer
capi20_get_version
capi20_get_serial_number
capi20_get_profile
capi20_isinstalled
capi20_fileno
```

In the Linux environment all required types for the functional interface to the **COMMON-ISDN-API** services can be included as follows:

```
#include <sys/types.h>
#include <linux/capi.h>
#include <capi20.h>
```

8.16.1 Message Operations

8.16.1.1 CAPI_REGISTER

Description

This is the function the application uses to announce its presence to **COMMON-ISDN-API**. The application describes its needs by passing the three parameters *MaxLogicalConnection*, *MaxBDataBlocks* and *MaxBDataLen*.

Parameter *MaxLogicalConnection* 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 set-up failure and an error indication from **COMMON-ISDN-API**.

Parameter *MaxBDataBlocks* specifies the maximum number of received data blocks that can be reported to the application simultaneously for each logical connection. The number of simultaneously available data blocks has a decisive effect on the data throughput in the system and should be between 2 and 7. At least two data blocks must be specified.

Parameter *MaxBDataLen* specifies the maximum size of the application data block to be transmitted and received. Selection of a protocol that requires larger data units, or attempts to transmit or receive larger data units will result in an error indication from **COMMON-ISDN-API**. The default value for the protocol ISO 7776 (X.75) is 128 octets. **COMMON-ISDN-API** is able to support at least up to 2048 octets.

Function call

```
unsigned capi20_register (              unsigned maxLogicalConnection,
                                        unsigned maxBDataBlocks,
                                        unsigned maxBDataLen,
                                        unsigned *pApplID );
```

Parameter	**Comment**
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 is to place the assigned application identification number

Return Value

Return Value	Comment
0x0000	Registration successful: application identification number was assigned
All other values	Coded as described in parameter Info, class 0x10xx.

8.16.1.2 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

```
unsigned capi20_release (                              unsigned ApplID);
```

Parameter	Comment
ApplID	Application identification number assigned by 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

8.16.1.3 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

```
unsigned capi20_put_message (          unsigned ApplID,
                                       unsigned char *Msg );
```

Parameter	Comment
ApplID	Application identification number (ApplID)
Msg	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.

8.16.1.4 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

```
unsigned capi20_get_message (                    unsigned ApplID,
                                                 unsigned char **Buf);
```

Parameter	Comment
ApplID	Application identification number (ApplID)
Buf	Pointer to the memory location where COMMON-ISDN-API should place the pointer to the retrieved message

Return Value

Return Value	Comment
0x0000	No error– message was retrieved from COMMON-ISDN-API
All other values	Coded as described in parameter Info, class 0x11xx

Note

The received message becomes 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 synchronization between threads has to be done by the application.

8.16.2 Other Functions

8.16.2.1 CAPI_WAIT_FOR_MESSAGE

Description

This operation is used by the application to wait for an asynchronous event from the CAPI.

Function call

unsigned capi20_waitformessage (	**unsigned ApplID,** **struct timeval *TimeOut);**

Parameter	**Comment**
ApplID	Application identification number (ApplID)
TimeOut	Pointer to a *struct timeval* value containing the maximum time to wait. If NULL, the function waits until a message is available or a capi_release is done.

Return Value

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

Note

This function returns as soon as a message from CAPI is available or another application’s thread issues a capi20_release() call.

8.16.2.2 CAPI_GET_MANUFACTURER

Description

With this operation the application determines the manufacturer identification of **COMMON-ISDN-API** or of the controller(s). Buf on call is a 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

```
unsigned char *capi20_get_manufacturer (       unsigned Ctrl,
                                               unsigned char *Buf);
```

Parameter	Comment
Ctrl	Number of the controller. If 0, the manufacturer identification of the kernel driver is provided to the application.
Buf	Pointer to a buffer of 64 bytes

Return Value

Return Value	Comment
0x0000	Error: no information available
All other values	Pointer to the buffer

8.16.2.3 CAPI_GET_VERSION

Description

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

Function call

```
unsigned char *capi20_get_version (         unsigned Ctrl,
                                            unsigned char *Buf);
```

Parameter	Comment
Ctrl	Number of the controller. If 0, the version of the kernel driver is provided to the application.
Buf	Pointer to a buffer long enough for COMMON-ISDN-API to store four 32 bit values: the first pair of values is the version number of COMMON-ISDN-API or the controller (first value: major version number (2), second value: minor version number (0)), the second pair is a manufacturer-specifc version (third value: major manufacturer version number, fourth value: minor manufacturer version number)

Return Value

Return Value	Comment
0x0000	Error: no information available
All other values	Pointer to the buffer

8.16.2.4 CAPI_GET_SERIAL_NUMBER

Description

With this operation the application determines the (optional) serial number of **COMMON-ISDN-API** or of the controller(s). Buf on call is a 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

```
unsigned char *capi20_get_serial_number (      unsigned Ctrl,
                                               unsigned char *Buf);
```

Parameter	Comment
Ctrl	Number of the controller. If 0, the serial number of the kernel driver is provided to the application.
Buf	Pointer to a buffer of 8 bytes.

Return Value

Return Value	Comment
0x0000	Error: no information available
All other values	Pointer to the buffer

8.16.2.5 CAPI_GET_PROFILE

Description

The application uses this function to get the capabilities from **COMMON-ISDN-API**. Buf contains a 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. Ctrl contains the controller number (bit 0..6) for which this information is requested. The retrieved structure format is described at the beginning of chapter 8.

Function call

```
unsigned capi20_get_profile (          unsigned Ctrl,
                                       unsigned char *Buf);
```

Parameter	Comment
Ctrl	Number of the controller. If 0, only the number of controllers installed is provided to the application.
Buf	Pointer to a buffer of 64 bytes.

Return Value

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

Note

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

8.16.2.6 CAPI_INSTALLED

Description

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

Function call

```
unsigned capi20_isinstalled (                void);
```

Return Value

Return Value	Comment
0x0000	**COMMON-ISDN-API** is installed.
All other values	Coded as described in parameter Info, class 0x11xx

8.16.2.7 CAPI_FILENO

Description

This function can be used for old-style applications which require *poll()* or *select()*. The recommended approach (especially with regards to future versions which may no longer support this call) is to use *capi_waitformessage* and threads.

Function call

```
int capi20_fileno (                    unsigned ApplID);
```

Parameter	Comment
ApplID	Application identification number assigned by the function CAPI20_REGISTER

Return Value

Return Value	Comment
-1	Application identification number is illegal.
All other values	The file descriptor for the application identified by ApplID. This file descriptor may be used only for *poll()* or *select()* system calls.

8.17 Linux (Kernel Level)

For kernel-mode applications, the **COMMON-ISDN-API** 2.0 is implemented as an interface structure containing function pointers for the individual services. The structure has the following definition:

```
struct capi_interface {
        __u16   (*capi_isinstalled) (void);
        __u16   (*capi_register) (capi_register_params * rparam, __u16 * applidp);
        __u16   (*capi_release) (__u16 applid);
        __u16   (*capi_put_message) (__u16 applid, struct sk_buff * msg);
        __u16   (*capi_get_message) (__u16 applid, struct sk_buff ** msgp);
        __u16   (*capi_set_signal)
                (
                __u16 applid,
                void (*signal)(__u16 applid, __u32 param),
                __u32 param
                );
        __u16   (*capi_get_version) (__u32 contr, struct capi_version * verp);
        __u16   (*capi_get_serial) (__u32 contr, __u8 serial[8]);
        __u16   (*capi_get_profile) (__u32 contr, struct capi_profile * profp);
        __u16   (*capi_get_manufacturer) (__u32 contr,     __u8 buf[64]);
         int    (*capi_manufacturer) (unsigned int cmd, void *data);
};
```

The data types used in this structure are:

__u8, __u16, __u32	unsigned int types of indicated bit length
struct capi_register_params	defined in <linux/capi.h>
struct capi_version	defined in <linux/capi.h>
struct capi_profile	defined in <linux/capi.h>
struct capi_interface	defined in <linux/kernelcapi.h>
struct capi_interface_user	defined in <linux/kernelcapi.h>
struct sk_buf	defined in <linux/skbuff.h>

Two functions are provided to set up the kernel-mode **COMMON-ISDN-API**:

```
        struct capi_interface * attach_capi_interface(struct capi_interface_user *);
        int detach_capi_interface(struct capi_interface_user *);
```

Function *attach_capi_interface()* must be used to get access to the kernel-mode **COMMON-ISDN-API** by means of a capi_interface structure. All further requests are performed with the function pointers of the structure. The link between a client and kernel-mode **COMMON-ISDN-API** can be released by calling *detach_capi_interface().* A client of the kernel.mode **COMMON-ISDN-API** must provide an interface structure containing the name of the client, a pointer to a callback function used to signal a controller's up and down status. If a client of kernel-mode **COMMON-ISDN-API** is not interested in this callback feature a NULL pointer can be assigned to this structure field. The third field in the structure is used by kernel-mode **COMMON-ISDN-API** internally:

```
struct capi_interface_user {
        char                        name[20];
        void                         (*callback)(unsigned cmd, __u32 ctrl, void *data);
        struct capi_interface_user *next;
};
```

8.17.1 Message Operations

8.17.1.1 CAPI_REGISTER

Description

This is the function the application uses to announce its presence to **COMMON-ISDN-API**. The application describes its needs by passing three parameters via a pointer to a *capi_register_params* structure. Pointer applidp is used to store the application identification number in case of a successful registration.

The data type capi_register_params is defined as follows:

```
typedef struct capi_register_params {
        __u32 level3cnt;
        __u32 datablkcnt;
        __u32 datablklen;
} capi_register_params;
```

Parameter field level3cnt specifies the maximum number of logical connections this application can concurrently maintain. The special value **–2** is used to assign as many connections as supported by the controller. Any application attempt to exceed the logical connection count by accepting or initiating additional connections will result in a connection set-up failure and an error indication from kernel-mode **COMMON-ISDN-API**.

Parameter field datablkcnt specifies the maximum number of received data blocks that can be reported to the application simultaneously for each logical connection. The number of simultaneously available data blocks has a decisive effect on the data throughput in the system and should be between 2 and 7. At least two data blocks must be specified.

Parameter datablklen specifies the maximum size of the application data block to be transmitted and received. Selection of a protocol that requires larger data units, or attempts to transmit or receive larger data units will result in an error indication from kernel-mode **COMMON-ISDN-API**. The default value for the protocol ISO 7776 (X.75) is 128 octets. Kernel-mode **COMMON-ISDN-API** is able to support at least up to 2048 octets.

Function call

__u16 (*capi_register) (	**capi_register_params * Rparam, __u16 * Applidp);**

Parameter	Comment
Rparam	Pointer to registration parameter structure.
Applidp	Pointer to a 16 bit buffer for the application identification number. The buffer will only be written as result of a successful registration.

Return Value

Return Value	Comment
0x0000	No error, the application identification number has been stored.
All other values	Coded as described in parameter Info, class 0x10xx.

8.17.1.2 CAPI_RELEASE

Description

The application uses this operation to log off from kernel-mode **COMMON-ISDN-API**. Kernel-mode **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

```
__u16 (*capi_release) (                    __u16 Applid);
```

Parameter	Comment
Applid	Application identification number assigned by the CAPI_REGISTER operation.

Return Value

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

8.17.1.3 CAPI_PUT_MESSAGE

Description

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

Function call

```
__u16 (*capi_put_message) (                 __u16 Applid,
                                            struct sk_buff * Msg);
```

Parameter	Comment
Applid	Application identification number assigned by the CAPI_REGISTER operation.
Msg	Pointer to the message that is passed to kernel-mode COMMON-ISDN-API.

Return Value

Return Value	Comment
0x0000	No error
0x1103	The send queue is full – the operation could not be performed.
All other values	Coded as described in parameter Info, class 0x11xx.

Note

The message buffer Msg must have been allocated with *alloc_skb()* (see: <linux/skbuff.h>). The low-level driver is responsible to release the buffer.

8.17.1.4 CAPI_GET_MESSAGE

Description

With this operation the application retrieves a message from kernel-mode **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

__u16 (*capi_get_message) (	**__u16 Applid, struct sk_buff ** Msgp);**

Parameter	Comment
Applid	Application identification number assigned by the CAPI_REGISTER operation.
Msgp	Pointer to the memory location where kernel-mode COMMON-ISDN-API should place the pointer to the retrieved message.

Return Value

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

Note

The message buffer pointed to by Msgp must be released with *kfree_skb()* (see: <linux/skbuff.h> after it has been processed by the client of kernel-mode **COMMON-ISDN-API**.

8.17.2 Other Functions

8.17.2.1 CAPI_SET_SIGNAL

Description

The application can use this function to activate the use of a call-back function. The signaling function can be deactivated by a **CAPI_SET_SIGNAL** with parameter *signal* = NULL. The application is identified by parameter *Applid*. An additional parameter *Param* is passed to the call-back function.

Function call

```
__u16 (*capi_set_signal) (                    __u16 Applid,
                                              void (*Signal)(__u16 applid, __u32 Param),
                                              __u32 Param);
```

Parameter	Comment
Applid	Application identification number assigned by the CAPI_REGISTER operation.
Signal	Pointer to a signal handler function that kernel-mode COMMON-ISDN-API will call when new messages have been received and can be fetched with a CAPI_GET_MESSAGE operation. The two parameters of the signal handler are equal to the 1st and 3rd parameter of this CAPI_SET_SIGNAL operation.
Param	This parameter will be transferred to the signal handler function without changes.

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 the application's message queue,
- an announced busy condition is cleared, or
- an announced queue-full condition is cleared.

8.17.2.2 CAPI_GET_MANUFACTURER

Description

With this operation the application determines the manufacturer identification of kernel-mode **COMMON-ISDN-API** or of the controller(s). Buf on call is a pointer to a buffer of 64 bytes. Kernel-mode **COMMON-ISDN-API** copies the identification string, coded as a zero terminated ASCII string, to this buffer.

Function call

```
__u16 (*capi_get_manufacturer) (                __u32 Contr,
                                                __u8 Buf[64]);
```

Parameter	Comment
Contr	Number of the controller. If 0, the manufacturer identification of the kernel driver is given to the application.
Buf	Pointer to a buffer of 64 bytes

Return Value

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

8.17.2.3 CAPI_GET_VERSION

Description

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

Function call

```
__u16 (*capi_get_version) (                  __u32 Contr,
                                             struct capi_version * Verp);
```

Parameter	Comment
Contr	Number of the controller. If 0, the version of the kernel driver is given to the application.
Verp	Pointer to a buffer of data type capi_version. The buffer will not be written in case of an error.

Return Value

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

8.17.2.4 CAPI_GET_SERIAL_NUMBER

Description

With this operation the application determines the (optional) serial number of kernel-mode **COMMON-ISDN-API** or of the controller(s). Buf on call is a pointer to a buffer of 8 bytes. Kernel-mode **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

```
__u16 (*capi_get_serial) (                      __u32 Contr,
                                                __u8 Buf[8]);
```

Parameter	Comment
Contr	Number of the controller. If 0, the serial number of the kernel driver is given to the application.
Buf	Pointer to a buffer of 8 bytes. The buffer will not be written in case of an error.

Return Value

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

8.17.2.5 CAPI_GET_PROFILE

Description

The application uses this function to get the capabilities from **COMMON-ISDN-API**. Buf contains a 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. CtrlNr contains the controller number (bit 0..6) for which this information is requested. The retrieved structure format is described at the beginning of chapter 8.

Function call

```
__u16 (*capi_get_profile) (                    __u32 Contr,
                                               struct capi_profile * Profp);
```

Parameter	Comment
Contr	Number of the controller. If 0, only the number of installed controller is given to the application.
ProfP	Pointer to a buffer of type struct capi_profile. The buffer will not be written in case of an error.

Return Value

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

Note

This function can be extended, so an application has to ignore unknown bits. **COMMON-ISDN-API** will set every reserved field to 0. For a detailed description of the capi_profile structure see section 4.2.2.7 of **COMMON-ISDN-API** Version 2.0 Part I.

8.17.2.6 CAPI_INSTALLED

Description

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

Function call

```
__u16 (*capi_installed) (                              void);
```

Return Value

Return Value	Comment
0x0000	COMMON-ISDN-API is installed.
All other values	Coded as described in parameter Info, class 0x11xx

8.17.2.7 CAPI_MANUFACTURER

Description

This function can be used by an application to perform manufacturer-dependent operations.

Function call

```
__u16 (*capi_manufacturer) (          unsigned int Cmd,
                                      void *Data);
```

Parameter	Comment
Cmd	Code of a manufacturer specific command.
Data	Pointer to a buffer containing the parameters of the manufacturer specific command. The buffer is located in user memory!

Return Value

Return Value	Comment
All values	Manufacturer-dependent.

8.18 Windows XP 32bit (Application Level)

Under the operating system Windows 2000, the **COMMON-ISDN-API** services are provided via a DLL (Dynamic Link Library). The interface between applications and **COMMON-ISDN-API** is realized 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 "CAPI2032.DLL". It is a 32-bit DLL exporting 32-bit APIENTRY type functions.

The DLL functions are exported under following names and ordinal numbers:

CAPI_MANUFACTURER (reserved)	CAPI2032.99
CAPI_REGISTER	CAPI2032.1
CAPI_RELEASE	CAPI2032.2
CAPI_PUT_MESSAGE	CAPI2032.3
CAPI_GET_MESSAGE	CAPI2032.4
CAPI_WAIT_FOR_SIGNAL	CAPI2032.5
CAPI_GET_MANUFACTURER	CAPI2032.6
CAPI_GET_VERSION	CAPI2032.7
CAPI_GET_SERIAL_NUMBER	CAPI2032.8
CAPI_GET_PROFILE	CAPI2032.9
CAPI_INSTALLED	CAPI2032.10

These functions can be called by an application according to the DLL conventions as imported functions.

In the Windows 2000 environment, the following data types are used in defining the functional interface:

WORD	16-bit unsigned integer
DWORD	32-bit unsigned integer
PVOID	Pointer to any memory location
PVOID *	Pointer to a PVOID
char *	Pointer to a character string
DWORD *	Pointer to a 32-bit unsigned integer value

8.18.1 Message Operations

8.18.1.1 CAPI_REGISTER

Description

This is the function the application uses to announce its presence to **COMMON-ISDN-API**. The application describes its needs by passing the four parameters *MessageBufferSize*, *maxLogicalConnection*, *maxBDataBlocks* and *maxBDataLen*.

For a typical application, the amount of memory required should be calculated by the following formula:

MessageBufferSize = 1024 + (1024 * maxLogicalConnection)

The parameter *maxLogicalConnection* specifies the maximum number of logical connections this application can maintain concurrently. Any attempt by the application to exceed the logical connection count by accepting or initiating additional connections will result in a connection set-up failure and an error indication from **COMMON-ISDN-API**.

The parameter *maxBDataBlocks* specifies the maximum number of received data blocks that can be reported to the application simultaneously for each logical connection. The number of simultaneously available data blocks has a decisive effect on the data throughput in the system and should be between 2 and 7. At least two data blocks must be specified.

The parameter *maxBDataLen* specifies the maximum size of the application data block to be transmitted and received. Selection of a protocol that requires larger data units, or attempts to transmit or receive larger data units, will result in an error indication from **COMMON-ISDN-API**. The default value for the protocol ISO 7776 (X.75) is 128 octets. **COMMON-ISDN-API** is able to support at least up to 2048 octets.

Function call

```
DWORD APIENTRY 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

8.18.1.2 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 assigned in the earlier CAPI_REGISTER operation.

Function call

```
DWORD APIENTRY CAPI_RELEASE (DWORD ApplID);
```

Parameter	**Comment**
ApplID	Application identification number assigned by the function CAPI_REGISTER

Return Value

Return Value	**Comment**
0x0000	Application successfully released
All other values	Coded as described in parameter *Info,* class 0x11xx

8.18.1.3 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

```
DWORD APIENTRY CAPI_PUT_MESSAGE (     DWORD ApplID,
                                      PVOID pCAPIMessage);
```

Parameter	**Comment**
ApplID	Application identification number assigned by the function CAPI_REGISTER
pCAPIMessage	Pointer to the message being 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.

8.18.1.4 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 APIENTRY CAPI_GET_MESSAGE (        DWORD ApplID,
                                         PVOID * ppCAPIMessage);
```

Parameter	Comment
ApplID	Application identification number assigned by the function CAPI_REGISTER
ppCAPIMessage	Pointer to the memory location where **COMMON-ISDN-API** should place the 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 is especially important in multi-threaded applications where more than one thread may execute **CAPI_GET_MESSAGE** operations. The synchronization between threads has to be done by the application.

8.18.2 Other Functions

8.18.2.1 CAPI_WAIT_FOR_SIGNAL

Description

This operation is used by the application to wait for an asynchronous event from **COMMON-ISDN-API**.

Function call

This function returns as soon as a message from **COMMON-ISDN-API** is available.

```
DWORD APIENTRY CAPI_WAIT_FOR_SIGNAL (      DWORD ApplID);
```

Parameter	Comment
ApplID	Application identification number assigned by the function CAPI_REGISTER

Return Value

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

Note

This function also returns as soon as the application calls **CAPI_RELEASE**, even if no pending **COMMON-ISDN-API** message is available in the **COMMON-ISDN-API** message queue. The **COMMON-ISDN-API** application shall not destroy the thread while **CAPI_WAIT_FOR_SIGNAL** is in the blocking state.

8.18.2.2 CAPI_GET_MANUFACTURER

Description

With this operation the application obtains the manufacturer identification of **COMMON-ISDN-API** (DLL). SzBuffer is a 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 APIENTRY CAPI_GET_MANUFACTURER (char * SzBuffer);
```

Parameter	Comment
SzBuffer	Pointer to a buffer of 64 bytes

8.18.2.3 CAPI_GET_VERSION

Description

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

Function call

```
DWORD APIENTRY CAPI_GET_VERSION (    DWORD * pCAPIMajor,
                                     DWORD * pCAPIMinor,
                                     DWORD * pManufacturerMajor,
                                     DWORD * pManufacturerMinor);
```

Parameter	Comment
pCAPIMajor	Pointer to a DWORD which will receive the **COMMON-ISDN-API** major version number: 2
pCAPIMinor	Pointer to a DWORD which will receive the **COMMON-ISDN-API** minor version number: 0
pManufacturerMajor	Pointer to a DWORD which will receive the manufacturer-specific major number
pManufacturerMinor	Pointer to a DWORD which will receive the manufacturer-specific minor number

Return Value

Return	Comment
0x0000	No error, version numbers have been copied.

8.18.2.4 CAPI_GET_SERIAL_NUMBER

Description

With this operation the application obtains the (optional) serial number of **COMMON-ISDN-API**. SzBuffer on call is a 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 of up to seven digits, can be read from the buffer after the function has returned.

Function call

```
DWORD APIENTRY CAPI_GET_SERIAL_NUMBER (char * SzBuffer);
```

Parameter	Comment
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.

8.18.2.5 CAPI_GET_PROFILE

Description

The application uses this function to get the capabilities from **COMMON-ISDN-API**. *SzBuffer* contains a pointer to a data area of 64 bytes. In this buffer **COMMON-ISDN-API** copies information about implemented features, the number of controllers and supported protocols. *CtrlNr* contains the number of the controller (bit 0..6) for which this information is requested. The profile structure retrieved is described at the beginning of Chapter 8.

```
DWORD APIENTRY CAPI_GET_PROFILE (    PVOID SzBuffer,
                                     DWORD CtrlNr );
```

Parameter	Comment
SzBuffer	Pointer to a buffer of 64 bytes
CtrlNr	Number of Controller. If 0, only the number of installed controllers is returned to the application.

Return Value

Return	Comment
0x0000	No error
<> 0	Coded as described in parameter *Info*, class 0x11xx

Note

This function may be extended, so the application must ignore unknown bits in the profile structure. **COMMON-ISDN-API** will set every reserved field to 0.

8.18.2.6 CAPI_INSTALLED

Description

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

Function call

```
DWORD APIENTRY CAPI_INSTALLED (VOID)
```

Return Value

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

8.19 Windows XP 64bit (Application Level)

In the operating system Windows XP 64bit, the **COMMON-ISDN-API** services are provided via a DLL (Dynamic Link Library) named "**CAPI2064.DLL**". It is a 64-bit DLL exporting 64-bit APIENTRY type functions. **Windows-based applications** (64-bit) can use the DLL mechanism as described in Chapter **8: Specifications for Commercial Operating Systems**, Subclause **8.18: Windows XP 32bit (Application Level)**, without modification.

Note: An application shall send and receive data through the 64bit DLL by using the 64bit pointer in the CAPI messages DATA_B3_REQ/DATA_B3_IND.

8.20 Windows XP (Device Driver Level)

For kernel-mode applications, the **COMMON-ISDN-API** 2.0 must be implemented as kernel-mode device driver. The interface to such a kernel-mode device driver in Windows 2000 is based on I/O request packets (IRPs), which can be sent to the driver by either kernel-mode or user-mode applications.

A CAPI20 device driver creates at least one CAPI20 device object which can be addressed by an application. The ***Flags*** field of each device object must be ORed with DO_DIRECT_IO after creation. Each device object is given a name for identification. The name of a CAPI20 device object is \Device\CAPI20*x*, where *x* is a configured decimal ordinal number. The CAPI20 device object name can be used by kernel-mode applications to send IRPs to the corresponding CAPI20 device driver.

A CAPI20 device driver may support multiple controllers. The implementation is free to create a single device object for all supported controllers or a separate device object for each supported controller. Controller numbers are assigned for each CAPI20 device object starting with 1.

In order to be accessible to user-mode applications, a CAPI20 device driver creates a symbolic link object for each CAPI20 device object. The name of the symbolic link object is \DosDevices\CAPI20*x*, where *x* is the same ordinal number used in the device object name. This allows user-mode applications to access the driver's **COMMON-ISDN-API** services by using the name \\.\CAPI20*x* in a Win32 *CreateFile()* operation.

To ensure the correct loading order of a CAPI20 driver, the driver must be assigned to the group "CAPI20". This is achieved by adding the REG_SZ value entry "Group" to the driver's service subkey in the registry:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\<CAPI-Driver-Service>\

DisplayName: REG_SZ: CAPI20 Driver ...
ErrorControl: REG_DWORD: ...
Group: REG_SZ: CAPI20
ImagePath: REG_SZ: ...
Start: REG_DWORD: ...
Type: REG_DWORD: ...
```

The driver installation must ensure that the group CAPI20 is listed in ServiceGroupOrder immediately before the group NDIS.

To permit unambiguous configuration of all CAPI20 device drivers, a new common subkey is created in the Windows 2000 registry. This subkey is named CAPI20 and contains a subkey *x* for each CAPI20*x* device object created by CAPI20 device drivers. The CAPI20 subkey must be queried during the installation of a new CAPI20 device driver. If the CAPI20 subkey does not yet exist, the installation procedure must create it. For each device object created by the new driver, a new subkey *x* is created with the lowest possible ordinal number: The ordinal number for the first CAPI20 device object is 1. Thus the first CAPI20 device driver installed uses the name \Device\CAPI201 for its first device object and the name\Device\CAPI202 for its second device object (if any), etc. The ordinal numbers claimed by the new driver must be noted in the driver's private configuration data. When the driver is removed from the system, the de-installation procedure must also remove the corresponding subkeys under the CAPI20 subkey.

```
HKEY_LOCAL_MACHINE\SOFTWARE\CAPI20\
Contents:
1\
        NumberOfControllers: REG_DWORD: <Number of Controllers supported >
        Manufacturer: REG_SZ: <Manufacturer Name>
        DeviceName: REG_SZ: CAPI201
```

```
          /* For each supported controller a  controller subkey is created: */
          1\
                    Channels: REG_DWORD: <Number of B-channels supported by this controller>
          2\
          etc.
2\ ...
```

Every driver that conforms to CAPI20 must be designed to work in a chain of layered drivers. Thus the driver must not use any operation which is only legal for a highest-level driver.

Every driver that conforms to CAPI20 must be designed to be unloadable, i.e. the driver must set the entry point of its Unload routine in the DriverObject passed to its DriverEntry routine. The Unload routine must release all previously allocated kernel and hardware resources in order to permit an new initialization of the driver at a later time.

Every driver that conforms to CAPI20 must handle the cancellation of IRPs.

Every driver that conforms to CAPI20 must handle the following major function codes:

```
    IRP_MJ_CREATE
    IRP_MJ_CLEANUP
    IRP_MJ_CLOSE
    IRP_MJ_READ
    IRP_MJ_WRITE
    IRP_MJ_SHUTDOWN
    IRP_MJ_DEVICE_CONTROL
    IRP_MJ_INTERNAL_DEVICE_CONTROL
```

To receive shutdown notification from the system shutdown process in a highest-level driver, the driver must call IoShutdownNotification() in its DriverEntry routine, but must ignore any error returned by this call.

Three types of IRP_MJ_xxx functions are used by a user-mode application to communicate with the **COMMON-ISDN-API** device: IRP_MJ_DEVICE_CONTROL, IRP_MJ_READ and IRP_MJ_WRITE.

The IRP_MJ_INTERNAL_DEVICE_CONTROL function is reserved exclusively for use by kernel-mode applications, i.e. for inter-device driver communication.

IRP_MJ_DEVICE_CONTROL is used for all CAPI20 functions except CAPI_GET_MESSAGE and CAPI_PUT_MESSAGE.

The CAPI_GET_MESSAGE and CAPI_PUT_MESSAGE functions use IRP_MJ_READ/WRITE (user-mode and kernel-mode applications) or IRP_MJ_INTERNAL_DEVICE_CONTROL (kernel-mode applications only).

The following DEVICE_CONTROL and INTERNAL_DEVICE_CONTROL codes are defined for the **COMMON-ISDN-API** functions:

```
/*
*         the common device type code for driver conforming to CAPI20
*/
#define FILE_DEVICE_CAPI20 0x8001

/*
*         DEVICE_CONTROL codes for user AND kernel-mode applications
```

```
*/
#define CAPI20_CTL_BASE                     0x800
#define CAPI20_CTL_REGISTER                 (CAPI20_CTL_BASE+0x0001)
#define CAPI20_CTL_RELEASE                  (CAPI20_CTL_BASE+0x0002)
#define CAPI20_CTL_GET_MANUFACTURER         (CAPI20_CTL_BASE+0x0005)
#define CAPI20_CTL_GET_VERSION              (CAPI20_CTL_BASE+0x0006)
#define CAPI20_CTL_GET_SERIAL               (CAPI20_CTL_BASE+0x0007)
#define CAPI20_CTL_GET_PROFILE              (CAPI20_CTL_BASE+0x0008)

/*
*       INTERNAL_DEVICE_CONTROL codes for kernel-mode applications only
*/
#define CAPI20_CTL_PUT_MESSAGE              (CAPI20_CTL_BASE+0x0003)
#define CAPI20_CTL_GET_MESSAGE              (CAPI20_CTL_BASE+0x0004)

/*
*       The wrapped control codes as required by the system
*/
#define CAPI20_CTL_CODE(function,method) \
        CTL_CODE(FILE_DEVICE_CAPI20,function,method,FILE_ANY_ACCESS)

#define IOCTL_CAPI_REGISTER \
        CAPI20_CTL_CODE(CAPI20_CTL_REGISTER, METHOD_BUFFERED)

#define IOCTL_CAPI_RELEASE \
        CAPI20_CTL_CODE(CAPI20_CTL_RELEASE, METHOD_BUFFERED)

#define IOCTL_CAPI_GET_MANUFACTURER\
        CAPI20_CTL_CODE(CAPI20_CTL_GET_MANUFACTURER, METHOD_BUFFERED)

#define IOCTL_CAPI_GET_VERSION \
        CAPI20_CTL_CODE(CAPI20_CTL_GET_VERSION, METHOD_BUFFERED)

#define IOCTL_CAPI_GET_SERIAL \
        CAPI20_CTL_CODE(CAPI20_CTL_GET_SERIAL, METHOD_BUFFERED)

#define IOCTL_CAPI_GET_PROFILE \
        CAPI20_CTL_CODE(CAPI20_CTL_GET_PROFILE, METHOD_BUFFERED)

#define IOCTL_CAPI_MANUFACTURER \
        CAPI20_CTL_CODE(CAPI20_CTL_MANUFACTURER, METHOD_BUFFERED)

#define IOCTL_CAPI_PUT_MESSAGE \
        CAPI20_CTL_CODE(CAPI20_CTL_PUT_MESSAGE, METHOD_BUFFERED)

#define IOCTL_CAPI_GET_MESSAGE \
        CAPI20_CTL_CODE(CAPI20_CTL_GET_MESSAGE, METHOD_BUFFERED)
```

To transfer CAPI20-specific return values from the driver to kernel or user-mode applications, the status code of the IRP is set accordingly. Because only some IRP status codes are mapped directly to Win32 error codes (the return codes of DeviceIoControl(), ReadFile(), WriteFile()), the following status code representation for CAPI20 errors (*Info* values) must be used:

Info	Windows 2000 Status code	Win32 Error Code
0x1001	STATUS_TOO_MANY_SESSIONS	ERROR_TOO_MANY_SESSIONS
0x1002	STATUS_INVALID_PARAMETER	ERROR_INVALID_PARAMETER
0x1003	N.A.	
0x1004	STATUS_BUFFER_TOO_SMALL	ERROR_INSUFFICIENT_BUFFER
0x1005	STATUS_NOT_SUPPORTED	ERROR_NOT_SUPPORTED
0x1006	N.A.	

0x1007	STATUS_NETWORK_BUSY	ERROR_NETWORK_BUSY
0x1008	STATUS_INSUFFICIENT_RESOURCES	ERROR_NOT_ENOUGH_MEMORY
0x1009	N.A.	
0x100A	STATUS_SERVER_DISABLED	ERROR_SERVER_DISABLED
0x100B	STATUS_SERVER_NOT_DISABLED	ERROR_SERVER_NOT_DISABLED
0x1101	STATUS_INVALID_HANDLE	ERROR_INVALID_HANDLE
0x1102	STAUS_ILLEGAL_FUNCTION	ERROR_INVALID_FUNCTION
0x1103	STATUS_TOO_MANY_COMMANDS	ERROR_TOO_MANY_CMDS
0x1104	N.A.	
0x1105	STATUS_DATA_OVERRUN	ERROR_IO_DEVICE
0x1106	STATUS_INVALID_PARAMETER	STATUS_INVALID_PARAMETER
0x1107	STATUS_DEVICE_BUSY	ERROR_BUSY
0x1108	STATUS_INSUFFICIENT_RESOURCES	ERROR_NOT_ENOUGH_MEMORY
0x1109	N.A	
0x110A	STATUS_SERVER_DISABLED	ERROR_SERVER_DISABLED
0x110B	STATUS_SERVER_NOT_DISABLED	ERROR_SERVER_NOT_DISABLED

In Windows 2000, all communication between a device object and an application is associated with a file object. For this reason, a file object pointer (or “file handle”) is used instead of the application ID to link a CAPI20 device object with a user-mode or kernel-mode application. Any application IDs contained in **COMMON-ISDN-API** messages are therefore ignored.

In the following, the interface between the application and the **COMMON-ISDN-API** device driver is described by means of Win32 functions. These functions are available for user-mode applications only. The equivalent kernel-mode functions can be found in the Windows 2000 documentation.

8.20.1 Message Operations

8.20.1.1 CAPI_REGISTER

Description

This is the function the application uses to announce its presence to **COMMON-ISDN-API**. The application describes its needs by passing the four parameters *MessageBufferSize*, *maxLogicalConnection*, *maxBDataBlocks* and *maxBDataLen*.

For a typical application, the amount of memory required should be calculated by the following formula:

MessageBufferSize = 1024 + (1024 * maxLogicalConnection)

The parameter *maxLogicalConnection* specifies the maximum number of logical connections this application can maintain concurrently. Any attempt by the application to exceed the logical connection count by accepting or initiating additional connections will result in a connection set-up failure and an error indication from **COMMON-ISDN-API**.

The parameter *maxBDataBlocks* specifies the maximum number of received data blocks that can be reported to the application simultaneously for each logical connection. The number of simultaneously available data blocks has a decisive effect on the data throughput in the system and should be between 2 and 7. At least two data blocks must be specified.

The parameter *maxBDataLen* specifies the maximum size of the application data block to be transmitted and received. Selection of a protocol that requires larger data units, or attempts to transmit or receive larger data units, will result in an error indication from **COMMON-ISDN-API**. The default value for the protocol ISO 7776 (X.75) is 128 octets. **COMMON-ISDN-API** is able to support at least up to 2048 octets.

CAPI_REGISTER	**CAPI_CTL_REGISTER**

Implementation

To perform the CAPI_REGISTER operation, the application must first obtain a handle to the **COMMON-ISDN-API** device using the Win32 CreateFile function, then send a CAPI_CTL_REGISTER to the **COMMON-ISDN-API** device. CAPI_REGISTER passes the following data structure to the driver:

```
struct capi_register_params {
     WORD MessageBufferSize,
     WORD maxLogicalConnection,
     WORD maxBDataBlocks,
     WORD maxBDataLen
};
```

Only one CAPI_CTL_REGISTER may be sent with a given handle before a CAPI_CTL_RELEASE is sent. If an application program wants to register as more than one **COMMON-ISDN-API** application, it must obtain several handles using CreateFile and send one CAPI_CTL_REGISTER with each handle.

Example:

```
capi_handle = CreateFile( "\\\\.\\CAPI201",
                    GENERIC_READ | GENERIC_WRITE,
                    0,
                    NULL,
                    OPEN_EXISTING,
                    FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,
                    NULL );

r.MessageBufferSize = MessageBufferSize;
r.maxLogicalConnection = maxLogicalConnection;
r.maxBDataBlocks = maxBDataBlocks;
r.maxBDataLen = maxBDataLen;

ret = DeviceIoControl( capi_handle,
                    CAPI_CTL_REGISTER,
                    &r,
                    sizeof(struct capi_register_params),
                    NULL,
                    0,
                    &ret_bytes,
                    NULL );
```

8.20.1.2 CAPI_RELEASE

Description

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

CAPI_RELEASE	**CAPI_CTL_RELEASE**

Implementation

A CAPI_RELEASE can be performed in one of two ways. If the same handle is to be used again, a CAPI_CTL_RELEASE must be sent. If the handle is no longer needed, the CAPI20 device may simply be closed using CloseHandle.

Example:

```
ret = DeviceIoControl( capi_handle,
                CAPI_CTL_RELEASE,
                NULL,
                0,
                NULL,
                0,
                &ret_bytes,
                NULL );
```

or

```
CloseHandle(capi_handle);
```

8.20.1.3 CAPI_PUT_MESSAGE

Description

With this operation the application transfers a message to **COMMON-ISDN-API**. The application is identified by a file handle.

CAPI_PUT_MESSAGE	**WriteFile()/CAPI_CTL_PUT_MESSAGE**

Implementation

The CAPI_PUT_MESSAGE function can be performed using either a WriteFile() operation or an INTERNAL_DEVICE_CONTROL IRP. The INTERNAL_DEVICE_CONTROL method is available to kernel-mode applications only.

1. WriteFile() operation

In the WriteFile() operation, one data buffer is sent to the CAPI20 device driver. This buffer must contain the message *and*, in the case of a DATA_B3_REQ message, the associated data. The data (if applicable) must be placed in the buffer immediately following the message.

```
ret = WriteFile( capi_handle,
                 ( PVOID )msg, /* buffer for message + data */
                 msg_length,      /* length of message + data */
                 &ret_bytes,
                &o_write );
```

The WriteFile() operation returns immediately, without waiting for any network event (in normal CAPI_PUT_MESSAGE operation).

When the WriteFile() call returns control to the application, the message buffer can be re-used.

2. INTERNAL_DEVICE_CONTROL

Kernel-mode applications may use an INTERNAL_DEVICE_CONTROL IRP with the IO_CONTROL code CAPI_CTL_PUT_MESSAGE for the CAPI_PUT_MESSAGE operation. With this IRP, a pointer to the following structure is passed to the CAPI20 device driver in Parameters.DeviceControl.Type3InputBuffer:

```
struct {
    PVOID message;
    PVOID data;
    };
```

The buffer passed in the message field can be re-used by the application as soon as the INTERNAL_DEVICE_CONTROL IRP is completed. The buffer passed in the data field can be re-used by the application as soon as the corresponding DATA_B3_CONF message is received.

8.20.1.4 CAPI_GET_MESSAGE

Description

With this operation the application retrieves a message from **COMMON-ISDN-API**. The application retrieves each message associated with the specified file handle which was used in the **CAPI_REGISTER** operation.

CAPI_GET_MESSAGE	**ReadFile()/CAPI_CTL_GET_MESSAGE**

Implementation

The CAPI_GET_MESSAGE operation can be performed either by calling ReadFile() or by using an INTERNAL_DEVICE_CONTROL IRP. The INTERNAL_DEVICE_CONTROL method is available to kernel-mode applications only.

1. ReadFile() operation

In the ReadFile() operation, one data buffer is received from the CAPI20 device driver. This buffer contains the message *and*, in the case of a DATA_B3_IND message, the associated data. The data (if applicable) is located in the buffer immediately following the message.

```
ret = ReadFile( capi_handle,
          buffer,
          buffer_size,
          &ret_bytes,
          &o_read );
```

The ReadFile() operation returns as soon as a **COMMON-ISDN-API** message is available.

The size of the buffer provided by the application should be at least MessageBufferSize + 512. If the buffer provided by the application is too small to hold the message and the data, an error is returned and the excess data is lost.

2. INTERNAL_DEVICE_CONTROL

Kernel-mode applications may use an INTERNAL_DEVICE_CONTROL IRP with the IO_CONTROL code CAPI_CTL_GET_MESSAGE for the CAPI_GET_-MESSAGE operation. With this IRP, a pointer to the following structure is passed to the CAPI20 device in Parameters.DeviceControl.Type3InputBuffer:

```
struct {
    PVOID message;
    PVOID data;
};
```

The CAPI20 device driver fills in the fields of this structure. When the INTERNAL_DEVICE_CONTROL is completed, the *message* field contains a pointer to the **COMMON-ISDN-API** message and, if the message is a DATA_B3_IND, the *data* field contains a pointer to the associated data buffer.

The message buffer may be re-used by the CAPI20 driver as soon as the application sends the next CAPI_CTL_GET_MESSAGE.

The data buffer may be re-used by the CAPI20 driver as soon as the application sends a corresponding DATA_B3_RESP message.

8.20.1.5 CAPI_SET_SIGNAL

There is no CAPI_SET_SIGNAL function. The asynchronous signaling of a received message is implicit in the completion of the corresponding message retrieval operation, whether ReadFile() or INTERNAL_DEVICE_CONTROL.

8.20.2 Other Functions

8.20.2.1 CAPI_GET_MANUFACTURER

Description

With this operation the application obtains the **COMMON-ISDN-API** manufacturer identification. The parameter Controller (dword) contains the number of the controller (bit 0..6) for which this information is requested. The application provides a buffer of at least 64 bytes. **COMMON-ISDN-API** copies the identification, coded as a zero-terminated ASCII string, to this buffer.

CAPI_GET_MANUFACTURER	**CAPI_CTL_GET_MANUFACTURER**

Implementation

With this IO_CONTROL the manufacturer identification is read from the Common ISDN API driver. A buffer of 64 bytes has to be provided by the application. The manufacturer identification is returned as zero terminated ASCII string. If the size of the incoming buffer of the io_control operation is larger or equal to sizeof (dword) the buffer is interpreted as the parameter *controller*.

8.20.2.2 CAPI_GET_VERSION

Description

With this function the application obtains the version of **COMMON-ISDN-API**, as well as an internal revision number. The parameter Controller (dword) contains the number of the controller (bit 0..6) for which this information is requested.

CAPI_GET_VERSION	**CAPI_CTL_GET_VERSION**

Implementation

The version of the **COMMON-ISDN-API** is read using this IO_CONTROL. If the size of the incoming buffer of the io_control operation is larger or equal to sizeof (dword) the buffer is interpreted as the parameter *controller*. The application must provide a buffer with the following structure:

```
struct capi_version_params {
    word CAPIMajor;
    word CAPIMinor;
    word ManufacturerMajor;
    word ManufacturerMinor;
};
```

8.20.2.3 CAPI_GET_SERIAL_NUMBER

Description

With this operation the application obtains the (optional) serial number of **COMMON-ISDN-API**. The parameter Controller (dword) contains the number of the controller (bit 0..6) for which this information is requested. The application provides a buffer of 8 bytes. **COMMON-ISDN-API** copies the serial number string to this buffer. The serial number, a seven-digit number coded as a zero-terminated ASCII string, is present in this buffer after the function has returned.

CAPI_GET_SERIAL_NUMBER	**CAPI_CTL_GET_SERIAL_NUMBER**

Implementation

The **COMMON-ISDN-API** serial number is read from the driver using this IO_CONTROL. If the size of the incoming buffer of the io_control operation is larger or equal to sizeof (dword) the buffer is interpreted as the parameter *controller*. The application must provide a buffer of 8 bytes. The serial number is returned as a zero-terminated ASCII string.

8.20.2.4 CAPI_GET_PROFILE

Description

The application uses this function to determine the capabilities of **COMMON-ISDN-API**. The parameter Controller (dword) contains the number of the controller (bit 0..6) for which this information is requested. The profile structure retrieved is described at the beginning of Chapter 8.

CAPI_GET_PROFILE	**CAPI_CTL_GET_PROFILE**

Implementation

The **COMMON-ISDN-API** capabilities can be read from the driver using this IO_CONTROL. If the size of the incoming buffer of the io_control operation is larger or equal to sizeof (dword) the buffer is interpreted as the parameter *controller*. If the size of the incoming buffer is less than sizeof (dword) or the parameter *controller* is set to 0, the number of installed controller is returned.

INDEX (PART II)