



308-381
Issue 1

AT&T 3270 Emulator +

High Level Language Application
Program Interface (**HLLAPI**)

Programmer's Guide

**©1988 AT&T
All Rights Reserved
Printed in USA**

NOTICE

The information in this document is subject to change without notice. AT&T assumes no responsibility for any errors that may appear in this document.

DEC is a registered trademark of Digital Equipment Corporation.
IBM and SNA are registered trademarks of International Business
Machines Corporation.

Tektronix is a registered trademark of Tektronix, Inc.
Teletype and UNIX are registered trademarks of AT&T.

Table of Contents

Preface

About This Guide	vii
Operating Requirements	x
Related Documentation	xiii
Other AT&T 3270 Emulator+ Documentation	xiv

1

HLLAPI and API

Introduction	1-1
The High Level Language Application Program Interface (HLLAPI)	1-3
The HLLAPI Tutorial	1-7
The Application Program Interface (API)	1-8

2

HLLAPI Overview

AT&T 3270 Emulator+ HLLAPI Concepts	2-1
AT&T 3270 Emulator+ HLLAPI Functions	2-3
Using the AT&T 3270 Emulator+ HLLAPI	2-8
The Environment	2-12

3

HLLAPI Tutorial

Introduction	3-1
Using the AT&T 3270 Emulator+ HLLAPI Tutorial	3-2

4	Application Program Interface	
	Introduction	4-1
	The API Tutorial	4-4
	Application Development	4-8

	Manual Pages	
	Introduction	1

A	AT&T 3270 Emulator+ HLLAPI Examples	
	Example 1	A-1
	Example 2	A-3

B	AT&T 3270 Emulator+ HLLAPI Functions	
	AT&T 3270 Emulator+ HLLAPI Functions	B-1

C	The xhllapi.h File	
	The xhllapi.h File	C-1

D	File Transfer Messages	
	File Transfer Messages	D-1

E	API Program Check Error Codes	
	API Program Check Error Codes	E-1

F	API Communication Check Error Codes API Communication Check Error Codes	F-1
G	API User/System Error Codes API User/System Error Codes	G-1
H	API LUV_TRC Status Displays API LUV_TRC Mode Status Line Displays	H-1
I	API External Symbols API External Symbols	I-1
J	Glossary Glossary	J-1
X	Index Index	X-1

List of Figures

Figure 2-1: HLLAPI Return Codes	2-11
Figure 2-2: Sample LUTABLE	2-14

About This Guide

The AT&T 3270 Emulator+ provides interactive data communication between ASCII terminals at an AT&T 3B Computer and remote mainframe systems (IBM or other). It provides the following functional features:

- IBM 3270 emulation
- Support for both the Binary Synchronous Communications (BSC) and the Systems Network Architecture (SNA) Synchronous Data Link Control (SDLC) protocols
- The ability to run the AT&T SNA/RJE Emulator+ simultaneously over the same SDLC link to the remote host computer
- The 3270 High Level Language Application Program Interface (HLLAPI)
- The 3270 Application Program Interface (API)
- The AT&T 3270 Emulator+ ESCORT Interface

This guide provides the information that you will need to use the AT&T 3270 Emulator+ High Level Language Application Program Interface (HLLAPI) or the AT&T 3270 Emulator+ Application Program Interface (API).

Who Should Read This Guide

This guide is intended for AT&T 3B Computer users and programmers who want to write C language application programs that use the functions of the AT&T 3270 Emulator+ HLLAPI and of the AT&T 3270 Emulator+ API.

The AT&T 3270 Emulator+ HLLAPI is a library of C language functions that can be called from a C language application program to communicate a series of 3270 tasks to a remote host computer (IBM or other). The AT&T 3270 Emulator+ API is a low-level version of the AT&T 3270 Emulator+ HLLAPI that provides two modes of accessing a remote host: 3270 data stream mode and "raw" data stream mode.

In 3270 data stream mode, an application program using either HLLAPI or API functions to communicate with the remote system is functionally equivalent to the interactive user interface to the host through a 3278/9 terminal. The raw mode, only available in the API, provides a raw data transfer interface that is independent of any terminal functionality (i.e., no 3270 data stream encoding/decoding is done).

In general, the AT&T 3270 Emulator+ API is more difficult to learn and use than the AT&T 3270 Emulator+ HLLAPI. Therefore, the use of HLLAPI rather than API functions is recommended, except in those application programs that must access the remote host computer in raw data stream mode.

Organization of This Guide

The *HLLAPI Programmer's Guide* is organized as follows:

- The *Preface* discusses prerequisites for the understanding and use of the AT&T 3270 Emulator+ application programming facilities.
- Chapter 1, *HLLAPI and API*, presents an overview of the AT&T 3270 Emulator+ application programming facilities, HLLAPI and API.
- Chapter 2, *HLLAPI Overview*, introduces AT&T HLLAPI concepts, and discusses several important things that you should know before getting started.
- Chapter 3, *HLLAPI Tutorial*, explains the on-line tutorial program for the AT&T 3270 Emulator+ HLLAPI.
- Chapter 4, *Application Program Interface*, discusses application program development using the AT&T 3270 Emulator+ API.
- *Manual Pages* contains the manual pages for the HLLAPI and API function calls.
- Appendix A, *AT&T 3270 Emulator+ HLLAPI Examples*, provides examples of how you would code a C language application program using AT&T HLLAPI function calls.
- Appendix B, *AT&T 3270 Emulator+ HLLAPI Functions*, lists the supported and unsupported IBM HLLAPI functions and the AT&T 3270 Emulator+ HLLAPI extensions.

- Appendix C, *The xhllapi.h File*, shows the **xhllapi.h** header file that you must include in your application program.
- Appendix D, *File Transfer Messages*, explains the messages that you may see when transferring files between an AT&T 3B Computer and a remote host computer.
- Appendix E, *API Program Check Error Codes*, lists the error messages that you may get when an API system call fails because of errors in the data received from the remote host computer.
- Appendix F, *API Communication Check Error Codes*, lists the error messages that you may get when an API system call fails because of conditions detected at the local communications interface.
- Appendix G, *API User/System Error Codes*, lists the error messages that you may get when an API system call fails because of user application or system errors.
- Appendix H, *API LUV_TRC Status Displays*, explains the API trace display that appears in the status line.
- Appendix I, *API External Symbols*, lists the global symbols used by the API library.
- The *Glossary* defines important terms that are used in this guide.
- The *Index* lists the key subjects treated in this guide and gives page references for them.

Operating Requirements

AT&T 3B Computer

The AT&T 3270 Emulator+ HLLAPI and API are intended for use on an AT&T 3B Computer.

- The 3B2 Computer must be equipped with a minimum of two megabytes of memory and an Intelligent Serial Controller (ISC) board. The ISC board should be installed following the procedures in the *AT&T 3B2 Computer Intelligent Serial Controller Manual*. (A separate ISC board is required for each active communication line to a host.) Each board supports a single line of either BSC or SNA/SDLC protocol. A user can install multiple ISC boards in the computer with any desired mix of boards supporting BSC or SNA.
- The 3B5 and 3B15 Computers must be equipped with an Input/Output Accelerator (IOA) processor and a Synchronous Data Link (SDLI).
- The 3B4000 Computer requires an ACP with an associated ISC card.

The AT&T 3270 Emulator+ supports up to 9600 baud for BSC and up to 19.2 kilobaud for SNA/SDLC operation.

The 3B Computer connects to the host using the switched telephone network or a non-switched (leased or private) line and a synchronous modem. The modem must supply clock to the ISC board. The host connects to the communication line through a communications controller, data adapter unit, or transmission control unit that is attached to the line using a modem that is compatible with the modem used by the 3B Computer.

Terminal Requirements

The AT&T 3270 Emulator+ provides screen and keyboard customization functions to allow work with any ASCII terminal supported by the AT&T 3B Computer. In addition, pre-customized files are provided for the following terminals:

- AT&T 4410, 4418, 4425, 605, 610, 615, 620, and 630
- Teletype 5410, 5418, and 5425
- Hewlett-Packard 2621
- Lear-Siegler ADM-3a
- Televideo 910, 924
- DEC VT-100
- Tektronix 4105A

Printer Requirements

The printer emulation procedures require the following operating features:

- A RETURN is required to advance the print head to column 1 of the next line.
- A Line Feed (LF) is required to advance the print head position to the current column of the next line. On some printers you may use "stty" on the target device to execute a line feed.
- Backspace capability.
- Formfeed capability.

Software Requirements

The AT&T 3270 Emulator+ operates under UNIX System V Releases 2.0 and later. Inter-Process Communication (IPC) software and the Terminal Information Utilities package must also be part of the software on your 3B Computer.

You must write your HLLAPI application program in the C programming language. This guide assumes that you are familiar with the UNIX System and the C programming language, although relevant information is included where needed. If you need additional information, consult the following documents for the UNIX System V release running on your system:

- *AT&T UNIX System V User's Guide*
- *AT&T UNIX System V User's Reference Manual*
- *AT&T UNIX System V Programmer's Guide*
- *AT&T UNIX System V Programmer's Reference Manual*

Related Documentation

This guide assumes that you have a working knowledge of the IBM 3270 Personal Computer and the BSC and SNA/SDLC protocols. If you need additional information regarding these topics, you may refer to the following documents:

- *IBM 3270 Personal Computer Control Program User's Guide and Reference*
- *IBM 3270 Personal Computer Control Program Programming Guide*
- *IBM 3270 Information Display System*
- *3274 Control Unit Description and Programmer's Guide*
- *IBM Systems Network Architecture: Concepts and Products*
- *IBM Systems Network Architecture Technical Overview*
- *IBM Synchronous Data Link Control (SDLC) Concepts*
- *IBM General Information: Binary Synchronous Communications*

Other AT&T 3270 Emulator+ Documentation

For more information regarding the AT&T 3270 Emulator+ product, please refer to the following documents:

- *AT&T 3270 Emulator+ Product Overview*
- *AT&T 3270 Emulator+ User's Guide*
- *AT&T 3270 Emulator+ System Administrator's Guide*
- *AT&T 3270 Emulator+ 3B2 Release Notes*
- *AT&T 3270 Emulator+ 3B5/15/4000 Release Notes*

1 HLLAPI and API

Introduction 1-1

The High Level Language Application Program Interface (HLLAPI) 1-3

AT&T 3270 Emulator+ HLLAPI Sample Scenarios 1-3

- Analyzing, Extracting, and Using Remote Host Computer Messages 1-3
- Sending Keystrokes to a Remote Host Computer 1-4
- Distributed Processing: Intercepting Communications between the Host and the Terminal User 1-5
- Distributed Processing: Extracting Data from Host Sessions 1-5
- Distributed Processing: Transferring Files to and from a Remote Host Computer 1-6
- Automating Keystrokes 1-6
- Filtering Keystrokes 1-6

The HLLAPI Tutorial 1-7

The Application Program Interface (API) 1-8

Introduction

The AT&T 3270 Emulator+ includes two interfaces that allow the user's C language application programs to communicate a series of 3270 tasks to a remote host computer. These interfaces are the High Level Language Application Program Interface (HLLAPI), and the Application Program Interface (API). Both interfaces are libraries of C language functions that you can call from a C language application program to do a variety of functions. The AT&T 3270 Emulator+ HLLAPI is a high-level extension of the AT&T 3270 Emulator+ API that is easier to learn and use.

NOTE

Although the AT&T 3270 Emulator+ HLLAPI is an extension of the AT&T 3270 Emulator+ API, their use is mutually exclusive. You cannot make function calls to both interfaces from the same application program.

Both inexperienced and experienced users can increase their productivity by using either interface in their application programs. Both interfaces can:

- save users from writing complicated application programs to produce complex screens and commands
- automate logon sequences
- simplify complex 3270 tasks, such as copying and file transfers
- simplify existing remote host computer applications
- monitor 3270 tasks, such as console operation, response time control, and availability, without human intervention
- create remote host and workstation processing applications that divide the functions of an application between the host and the AT&T 3B Computer

The AT&T 3270 Emulator+ HLLAPI provides the following additional advantages:

- Because it is an extension to the AT&T Emulator+ API, HLLAPI allows less experienced programmers to gain access to advanced AT&T 3270 Emulator+ API functions.

- Application programs that use AT&T 3270 Emulator+ HLLAPI functions are often shorter and easier to read and debug than those using API functions.
- The nature of the AT&T Emulator+ HLLAPI and the C programming language make structured programming techniques easy to implement.
- Application programs that use AT&T 3270 Emulator+ HLLAPI functions are comparatively easy to maintain.

The High Level Language Application Program Interface (HLLAPI)

The AT&T 3270 Emulator+ HLLAPI is largely compatible with the IBM 3270 Personal Computer HLLAPI and provides extensions not available in the IBM product. The supported and unsupported IBM HLLAPI functions and the AT&T 3270 Emulator+ HLLAPI extensions are listed in Appendix B, *HLLAPI Functions*.

A C language application program that uses the AT&T 3270 Emulator+ HLLAPI is capable of performing 3270 tasks without human intervention. The next section illustrates some situations in which your application program can use HLLAPI functions.

AT&T 3270 Emulator+ HLLAPI Sample Scenarios

The scenarios presented in this section illustrate HLLAPI functions that your application program can perform in these areas:

- analyzing, extracting, and using remote host computer messages
- sending keystrokes to a remote host computer from an AT&T 3B Computer
- distributed processing: intercepting communications between the remote host computer and the terminal user
- distributed processing: extracting data from host sessions
- distributed processing: transferring files to and from a remote host computer
- automating keystrokes
- filtering keystrokes

Analyzing, Extracting, and Using Remote Host Computer Messages

A terminal operator trying to start a transaction with a remote host computer normally follows these steps:

1. start the transaction

2. wait for a response from the host
3. analyze the response to find out if it is the expected one
4. extract and use the data from the response

You can emulate these steps by using a series of AT&T Emulator+ HLLAPI functions. After determining the correct starting point for the host transaction, your application program can call the `h_search` (Search Presentation Space) HLLAPI function to determine which keyword messages or prompting messages are on the display screen.

Then your application program can use the `h_sendkey` (Send Key) HLLAPI function to key data into the remote host session and enter a host transaction. Your application program can also use the `h_copypss` (Copy Presentation Space to String) HLLAPI function (or any of several other Copy functions) to copy the desired data from the remote host into a specified data area.

Some host systems do not stay locked in XCLOCK or XSYSTEM mode until they respond; instead, they quickly unlock the keyboard and allow the operator to stack other requests. In these situations, the terminal operator depends on some other visual prompt to know that the data has returned. `h_search` (Search Presentation Space) allows your application program to search the presentation space while waiting. If no host event occurs after a reasonable timeout period, then your HLLAPI program can call a customized error message function.



Your application program must be revised for even minor changes in the display messages. Subtle changes in display message syntax can make your program call a customized error message function erroneously.

Sending Keystrokes to a Remote Host Computer

If you want to write application programs that send keystrokes to a remote host computer, you should be aware of several things. Some application environments may only require that the terminal operator key in a string followed by the ENTER key to issue a command. Other applications may involve more complex 3270 formatted screens in which the terminal operator can enter data into any one of several fields. You must understand the fields that the display needs to have filled in.

You must be aware of the field lengths and contents when you are sending keystrokes to a field using `h_sendkey` (Send Key). If you fill in the field completely and the next attribute byte is an "autoskip," the cursor will be moved to the next field. If you then enter a tab, you will skip to yet another field.

On the other hand, if your keystrokes do not fill the field completely, there may be data left from prior input. (You can clear this data using the Erase End of Field (EOF) key.)

Distributed Processing: Intercepting Communications between the Host and the Terminal User

There are a number of applications that provide a single end user interface, but perform their processing at two or more different locations. A HLLAPI application program can interact with host applications by intercepting the communications between the remote host computer and the terminal user. This can be done by having the application program ask to be notified each time the host presentation space is updated or every time the terminal user enters an AID key.

Your application program can then use the HLLAPI Copy functions to update fields or presentation spaces, or send keystrokes to the host using the `h_sendkey` (Send Key) function.

Distributed Processing: Extracting Data from Host Sessions

Your application can take the following steps to extract host data for local use:

1. Your program must call the `h_connect` (Connect Presentation Space) HLLAPI function to connect to the host presentation space containing the data that will be copied.
2. Your program can call the `h_copy` (Copy Presentation Space to String) HLLAPI function to obtain the desired data from the host presentation space.
3. Your program can then call the `h_connect` (Connect Presentation Space) HLLAPI function to connect to a different presentation space where the data will be entered.
4. Next, your program can use the `h_sendkey` (Send Key) HLLAPI function to route the data to that presentation space.

5. Finally, your program can use the `h_sendkey` (Send Key) HLLAPI function to send the keystrokes that will start the application.

You can also send data from your local application to the remote host computer.

Distributed Processing: Transferring Files to and from a Remote Host Computer

Your application program can use the `h_send` (Send File) and `h_rcv` (Receive File) HLLAPI functions to transfer data and files to and from the remote host computer. `h_send` and `h_rcv` are more efficient than the Copy functions when you are copying many screens of data.

Automating Keystrokes

Your application program may provide all of the keystrokes for another application or may mingle keystrokes to the target destination with those from the keyboard. In order to do this, your application must occasionally lock out other sources of keystroke input that may be destined for a target application or presentation space. This can be done by calling the `h_resv` (Reserve) HLLAPI function. Your program can later unlock the target application or presentation space by calling the `h_rel` (Release) function.

Filtering Keystrokes

Your application program can act as a filter by intercepting a keystroke coming from HLLAPI (either from the keyboard or a source application) targeted for another destination. Once intercepted, the keystroke may then be:

- ignored (i.e., deleted)
- redirected to another application
- validated
- converted from upper case to lower case, or from ASCII to EBCDIC

The HLLAPI Tutorial

The AT&T 3270 Emulator+ HLLAPI Tutorial is a menu-based full-screen interactive learning tool that allows you to invoke individual HLLAPI functions and see the results of executing these functions and the parameters returned by them without writing complex programs. You can also use the AT&T 3270 Emulator+ HLLAPI Tutorial as an informal testing tool.

The AT&T 3270 Emulator+ HLLAPI Tutorial organizes the HLLAPI functions into the following seven menus:

- Local Environment Functions Menu – includes the HLLAPI functions that perform basic operational tasks.
- Communications Functions Menu – includes the HLLAPI functions that support transactions with the remote host computer.
- Keyboard Functions Menu – includes the HLLAPI functions associated with handling terminal keystrokes between your application program and the remote host computer.
- File Transfer Functions Menu – includes the HLLAPI functions that allow an application program to transfer files between an AT&T 3B Computer and a remote host computer.
- Memory Management Menu – includes the HLLAPI functions that allow you to preallocate, free and query the blocks of storage used by the application program.
- Presentation Space Management Menu – includes the HLLAPI functions that support your application program by helping you manage the regions in storage where different screens are stored, copy data to and from these regions, and control how your screens will appear.
- Environment Functions Menu – includes the HLLAPI functions that provide access to the UNIX System.

The AT&T 3270 Emulator+ HLLAPI Tutorial allows access to the UNIX System shell from any of its menus.

The Application Program Interface (API)

The AT&T 3270 Emulator+ Application Program Interface (API) is a low-level version of the AT&T 3270 Emulator+ HLLAPI. It is also a library of C language functions that you can call from a C language application program to emulate the actions of a 3270 terminal operator. Just like the AT&T 3270 Emulator+ HLLAPI, the API allows your application program to communicate with remote host computers from an AT&T 3B Computer. However, the API provides two modes of accessing a remote host: 3270 data stream mode and "raw" data stream mode. In 3270 data stream mode, the application program, using either HLLAPI or API functions to communicate with the remote system, is functionally equivalent to the interactive user interface to the host through a 3278/9 terminal. The raw mode, only available in the API, provides a raw data transfer interface that is independent of any terminal functionality (i.e., no 3270 data stream encoding/decoding is done).

In general, the AT&T 3270 Emulator+ API is more difficult to learn and use than the AT&T 3270 Emulator+ HLLAPI. Therefore, the use of HLLAPI rather than API functions is recommended, except in those application programs that must access the remote host computer in raw data stream mode.

NOTE

A Tutorial is also provided with the AT&T 3270 Emulator+ API. (See "The API Tutorial" in chapter 4.)

2

HLLAPI Overview

AT&T 3270 Emulator+ HLLAPI Concepts	2-1
Host Session	2-1
Presentation Space	2-1
WS Ctrl	2-1
Operator Information Area (OIA)	2-2
Attention Identifier Keys	2-2
Combination Keys	2-2

AT&T 3270 Emulator+ HLLAPI Functions	2-3
Local Environment Functions	2-3
Communications Functions	2-4
Keyboard Functions	2-5
File Transfer Functions	2-5
Storage Manager Function	2-5
Presentation Space Functions	2-6
Environment Functions	2-7

Using the AT&T 3270 Emulator+ HLLAPI	2-8
Calling Parameters	2-8
Returned Parameters	2-9
Linking Your Application Program	2-11

The Environment

2-12

Environment Variables

2-12

The LUTABLE File

2-13

AT&T 3270 Emulator+ HLLAPI Concepts

This section presents basic concepts that you will have to understand to use the AT&T 3270 Emulator+ HLLAPI.

Host Session

The communications between your application program and the IBM host computer are called *host sessions*. A *host session* is the logical connection between a *presentation space* and an application program running on a remote host computer.

Presentation Space

A *presentation space* is a region in computer storage that can be displayed on your terminal screen.



The IBM 3270 Personal Computer can display several windows on the screen. The AT&T 3270 Emulator+ HLLAPI allows you to view one window at a time, and this window occupies the size of the entire screen.

The *current connected presentation space* is the active session to which you are connected.

WS Ctrl

WS Ctrl (Work Station Control) is treated as a session type by HLLAPI, even though it is really a "pseudo-session." As a pseudo-session, *WS Ctrl* can call functions against other sessions, such as copy blocks of text from one session to another. But there is no *presentation space* associated with the *WS Ctrl* session.

Operator Information Area (OIA)

The *Operator Information Area* (OIA) is the bottommost line of the screen that displays information about the status of your workstation and the remote host computer.

Attention Identifier Keys

The *Attention Identifier* (AID) keys are non-ASCII control keys: PF, PA, or Enter.

Combination Keys

Combination keys must be used with another key or keys to produce a desired function. Examples are the Control and Shift keys.

AT&T 3270 Emulator+ HLLAPI Functions

The AT&T 3270 Emulator+ HLLAPI functions are classified into seven groups, according to the services that they provide to an application program. There are:

- Local Environment Functions
- Communications Functions
- Keyboard Functions
- File Transfer Functions
- Storage Manager Function
- Presentation Space Functions
- Environment Functions

Each HLLAPI function is associated with a Symbolic Name, and a Function Number. The Symbolic Name of a function is the name that you use to refer to that function in a C application program. The Function Number of a function is the number used by the *IBM 3270 Personal Computer HLLAPI* to refer to that function.

The following sections describe the HLLAPI function groups. Each description includes the HLLAPI functions that are part of that group, along with their Function Numbers and Symbolic Names. This information is useful if you are familiar with the IBM product and want to write C language application programs using the AT&T 3270 Emulator+ HLLAPI.

Local Environment Functions

The AT&T HLLAPI Local Environment Functions perform basic operational tasks that support the interface between your HLLAPI application program and its copy of the HLLAPI library. These functions are summarized below.

Function Name	Function Number	Symbolic Name
Connect Presentation Space	1	H_CONNECT
Disconnect Presentation Space	2	H_DISC
Set Session Parameters	9	H_SETPARMS
Query Session	10	H_QSESS
Reserve	11	H_RESV
Release	12	H_REL
Work Station Control	16	H_WSCTRL
Query System	20	H_QSYS
Reset System	21	H_RESET
Query Session Status	22	H_QSTATUS
Connect and Interact with Presentation Space	113	H_CONNINT

Communications Functions

The AT&T HLLAPI Communications Functions represent the fundamental set of functions that support transactions with the remote host computer. These functions are:

Function Name	Function Number	Symbolic Name
Send Key	3	H_SENDKEY
Wait	4	H_WAIT
Pause	18	H_PAUSE
Start Host Notification	23	H_STRTHOST
Query Host Update	24	H_QHOST
Stop Host Notification	25	H_STOPHOST

Keyboard Functions

The AT&T HLLAPI Keyboard Functions are the set of function calls associated with handling terminal keystrokes between your application program and the host computer. These functions are:

Function Name	Function Number	Symbolic Name
Start Keystroke Intercept	50	H_STARTKEY
Get Key	51	H_GETKEY
Post Intercept Status	52	H_POSTINT
Stop Keystroke Intercept	53	H_STOPKEY

File Transfer Functions

The AT&T HLLAPI File Transfer Functions allow an HLLAPI application program to transfer files between an AT&T 3B Computer and an IBM host. The two File Transfer Functions are:

Function Name	Function Number	Symbolic Name
Send File	90	H_SEND
Receive File	91	H_RECV

Storage Manager Function

The Storage Manager Function preallocates blocks of storage for use by certain HLLAPI functions.

You can make the following subfunction calls to the Storage Manager Function:

- Get Storage

- Free Storage
- Query Free Storage
- Free All Storage

Function Name	Function Number	Symbolic Name
Storage Manager	17	H_STMAN

Presentation Space Functions

The AT&T HLLAPI Presentation Space Functions provide support for your application program by helping you manage presentation spaces, copy data to and from various regions of storage, and control how your screen will appear.

The Presentation Space Functions are summarized below.

Function Name	Function Number	Symbolic Name
Copy Presentation Space	5	H_COPY
Search Presentation Space	6	H_SEARCH
Query Cursor Location	7	H_QCUR
Copy Presentation Space to String	8	H_COPYPSS
Copy OIA	13	H_CPOIA
Query Field Attribute	14	H_QATTR
Copy String to Presentation Space	15	H_CPSTR
Search Field	30	H_SRCHFLD
Find Field Position	31	H_FNDPOS
Find Field Length	32	H_FNDLEN
Copy String to Field	33	H_CPSTRF
Copy Field to String	34	H_CPFIELD
Convert Position or RowCol	99	H_CONV
Change Cursor Position in Presentation Space	111	H_CHCUR
Write a Character in Presentation Space	112	H_WRCHAR

Environment Functions

The AT&T HLLAPI Environment Functions provide direct access to the UNIX System. Your application program can execute UNIX System commands or programs and place the resulting data in the program segment following the function call or pipe it to other processes. The two Environment Functions are shown below.

Function Name	Function Number	Symbolic Name
Invoke UNIX System Command or Program	92	H_INVOKE
UNIX Redirect	93	H_REDIR

Using the AT&T 3270 Emulator+ HLLAPI

You can use an AT&T 3270 Emulator+ HLLAPI function by coding the `hllapi` function in your application program with a set of data arguments referred to as *calling parameters*. After executing the function call, the `hllapi` function returns certain parameters that your application program might use in an error routine.

Both the calling parameters and the returned parameters vary depending on the HLLAPI function that you are trying to call. An overview of these parameters follows.

Calling Parameters

The AT&T 3270 Emulator+ HLLAPI functions are called by invoking `hllapi` in your application program, as follows:

```
#include <xhllapi.h>

int hllapi(func,data,length,position)
int *func;
char *data;
int *length;
int *position;
```

The data arguments passed to the `hllapi` function are:

- | | |
|------------------|---|
| *func | is a pointer to the desired function code, preferably in the form of a symbolic name for the desired function. This parameter is always required. |
| *data | This argument is used in different ways by different functions; in some functions it is a pointer to a character array, while in others it is a pointer to a structure. |
| *length | This argument usually points to the length of whatever the data parameter points to. |
| *position | is a pointer to a value associated with the emulated 3278/3279 Display Station screen position. This value can be: |

- 1 through 1920 for Model 2's, or
- 1 through 3564 for Model 5's

You must provide all parameters when making a function call. You can use a filler variable for those function calls that do not require all parameters.

The **xhllapi.h** file that you must include in your application program with **#include** defines the symbolic names for the AT&T HLLAPI functions and return codes, and the structure identifiers used by several HLLAPI functions. This file is listed in Appendix C, *The xhllapi.h File*. The **xhllapi.h** file does not define storage; you are responsible for allocating the storage buffers required by your application program.

Returned Parameters

The AT&T 3270 Emulator+ HLLAPI function calls return a return code from Figure 2-1. For example, if you made the following call in your application program

```
y = hllapi(func,data,length,position)
```

the variable **y** would be assigned one of the return code values shown in Figure 2-1. These values inform your application program of the success, failure or status of the function call. This information might be used by an error handling routine in your application program.

In addition, the AT&T 3270 Emulator+ HLLAPI function calls return requested information at the locations pointed to by the calling parameters, as shown below; not all four parameters will be changed on return for each function call.

*func	This parameter is not changed.
*data	This parameter returns different information, depending on the function; in some functions, it points to a string of characters, in others to a structure.

- *length** usually points to the length of whatever the *data* parameter points to, although a *position* can be returned in this field.
- *position** The function return code described above is also placed at the location pointed to by this parameter.

The manual pages for the HLLAPI function calls describe the return codes that apply to each individual function call, although the general meanings are consistent with the standard return codes shown below. This consistency allows you to use a common error-handling routine in your HLLAPI application program.

Return Code	Symbolic Name	Description
0	HE_SUCCESS	good return
1	HE_INVAL	you specified an invalid presentation space
2	HE_PARM	a parameter error was encountered, or an invalid function was specified (refer to the individual function for details)
3	HE_WSCTRL	WS Ctrl action has occurred
4	HE_BUSY	the target presentation was busy
5	HE_INHBT	the execution of this function was inhibited for some reason different than the one stated in return code 4
6	HE_LENGTH	a data error was caused by an invalid length parameter specification
7	HE_POS	you specified an invalid presentation space position
8	HE_PROC	a functional procedure error was encountered
9	HE_SYSERR	a system error was encountered
10	HE_FUNCT	function unavailable

Return Code	Symbolic Name	Description
11	HE_RSC	the resource that you requested is unavailable
21	HE_OIA	updated OIA
22	HE_PRES	updated presentation space
23	HE_BOTH	both of the above have been updated
24	HE_NOFIELD	no such field
25	HE_NOKEYS	requested keys are not available
26	HE_UPDATE	a host presentation space or OIA has been updated
301	HE_FNUM	invalid function number
302	HE_NOENT	file not found
305	HE_ACCESS	access denied
308	HE_MEM	insufficient memory
310	HE_ENV	invalid environment
311	HE_FORM	invalid format
8000	HE_DATA	only data portion has been updated
9998	HE_PSID	invalid presentation space
9999	HE_NOTPR	parameter was not 'p' or 'r'

Figure 2-1: HLLAPI Return Codes

Linking Your Application Program

Once you have written your application program, you must link it with the AT&T HLLAPI library. You can do this at compile time by using the command line:

```
cc -o user_application_program.c -lhllapi -lapi -lcurses -lgemusr
```

The Environment

Environment Variables

The AT&T 3270 Emulator+ HLLAPI uses the environment variables that are set and exported by the `snaenvset` or `bscenvset` shell scripts. In addition, the AT&T HLLAPI uses the following UNIX System environment variables and files:

KY3278	name of the keyboard customization file (object file)
LUPORT	contains the logical channel to be used for a particular session. If this environment variable is NULL, the LUPORT uses the next available logical channel.
LUTABLE	file name for LUTABLE
P3274	named pipe to the controller process (e.g., <code>sna</code> or <code>tm3274</code>)
RTMSG	path name for the location of the run-time message file
SC3278	name of the screen customization file (object file)
STSIZE	maximum storage size that you can allocate, using the Storage Manager Function. A practical maximum value is 64K, or the amount left over in the address space by the <code>hllapi</code> process, including your application program (your application program is linked with the <code>hllapi</code> library)
TE3279.MSG	name of the 3279 terminal message file
TM3279	terminal model, either 2 or 5

Refer to the *AT&T 3270 Emulator+ System Administrator's Guide* for more information on both the `snaenvset` and `bscenvset` scripts and the environment variables listed above.

The LUTABLE File

The **LUTABLE** file describes the logical units (LU's) that will be assigned to the controller type. **hllapi** will first look in your home directory to see if there is an **LUTABLE** file. If not, the file defined by the **LUTABLE** environment variable (above) will be used. You or the System Administrator for your installation can restrict access to the **LUTABLE** file in your home directory (see **chmod(1)** in the *UNIX System V User's Reference Manual* and **chmod(2)** in the *UNIX System V Programmer's Reference Manual*). This feature provides additional security, since it in turn restricts **hllapi** use.

Your application program can interact with up to 4 presentation spaces, although at any given time it can only be connected to one. Therefore, the **LUTABLE** contains a maximum of 4 entries, with a maximum of 6 fields in each entry. Each entry in the **LUTABLE** file contains the following fields, separated by blanks:

- The presentation space short name. You can use any capital letter from A through Z in the presentation space short name, but no numbers.
- The presentation space long name. You can use any combination of 8 characters, excluding white space characters (space and tab).
- LU number(s) or number range: the AT&T 3270 Emulator+ allows up to 32 LU's to be connected at any given time, numbered from 0 through 31. LU numbers are decimal digits separated by commas; LU number ranges are separated by dashes.
- The name of the controller pipe.
- The terminal model number, either 2 or 5.
- An extended-attribute-bytes indicator, either a 'y' or a blank.

NOTE

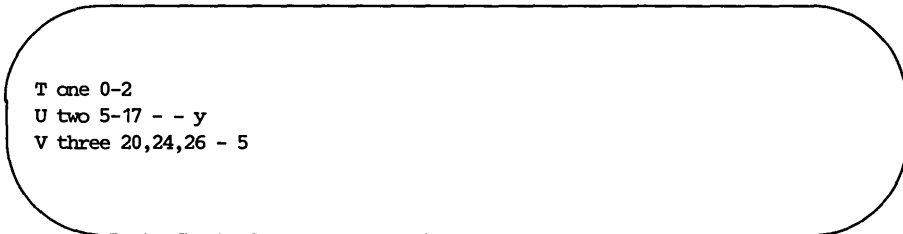
The LU number is equivalent to the host LU address minus 2, as defined in the LU-Macro Locator Field.

The Environment

Figure 2-2 shows a sample **LUTABLE** file. This file contains three entries that define three presentation spaces associated with the user application program. The first entry contains the presentation space short name "T," the presentation space long name "one," and the LU numbers 0 through 2. The name of the controller pipe, and the terminal model are not specified; therefore, the default values will be used. The default values are those that were assigned to the environment variables.

The second entry contains the presentation space short name "U," the presentation space long name "two," and the LU numbers 5 through 17. The two fields that follow contain a dash (-); this means that default values will be used for the name of the controller pipe and the terminal model. The last field contains a 'y,' which indicates that extended attributes will be used.

The last entry in Figure 2-2 is similar to the second entry, except that the terminal model is specified, the extended attributes field is left blank, and three non-consecutive LU numbers are specified.



```
T one 0-2
U two 5-17 - - y
V three 20,24,26 - 5
```

Figure 2-2: Sample LUTABLE

3

HLLAPI Tutorial

Introduction

3-1

Using the AT&T 3270 Emulator+ HLLAPI Tutorial

3-2

- Editing Mode

3-9

Environment Variables

3-11



Introduction

The AT&T 3270 Emulator+ HLLAPI Tutorial is a useful interactive learning tool that allows you to invoke individual HLLAPI functions, and see the results of executing these functions and the parameters returned by them without writing complex programs. You can also use the AT&T 3270 Emulator+ HLLAPI Tutorial as an informal testing tool.

The AT&T 3270 Emulator+ HLLAPI Tutorial runs through two separate programs: **hllapiprim**, and **hllapisec**. **hllapiprim** is the primary program that prompts you for the function name and the parameters required by the function that you want to execute. These parameters are passed to the secondary program, **hllapisec**, after have you entered the information requested by **hllapiprim**. **hllapisec** then shows the effect of executing the function call and the parameters returned by the function. The following section describes the actions that you should take to use the AT&T 3270 Emulator+ HLLAPI Tutorial.

Using the AT&T 3270 Emulator+ HLLAPI Tutorial

You can run the **hllapiprim** and **hllapisec** programs on the same terminal or on different terminals. When using one terminal, invoke the **hllapisec** program first as a background task, with the standard output and the standard error directed to files to avoid cluttering the screens.

NOTE

Your application program cannot establish a physical connect with the presentation space if **hllapisec** is running in the background

For example, typing in the command

```
hllapisec 1 > secout 2 > &1 &
```

redirects the standard output and the standard error to the file **secout**; you may use another file name if you like. Next, enter

```
hllapiprim
```

If you are using two terminals, the order in which you invoke the two AT&T 3270 Emulator+ HLLAPI Tutorial programs does not matter; on one terminal, enter

```
hllapiprim
```

on the other

```
hllapisec
```

See "The Environment" section in chapter 2 for information regarding environment variables that must be set before running the **hllapiprim** and **hllapisec** programs.

The **hllapiprim** program is a menu-based full-screen interactive program that organizes the functions supported by the AT&T 3270 Emulator+ HLLAPI into seven menus:

- Local Environment Functions Menu
- Communications Functions Menu

- Keyboard Functions Menu
- File Transfer Functions Menu
- Memory Management Menu
- Presentation Space Management Menu
- Environment Functions Menu

Once you invoke the **hllapiprim** program, the main menu will appear on the screen listing these menus, as follows:

HLLAPI Tutorial Main Menu

l - Local Environment Functions Menu
c - Communications Functions Menu
k - Keyboard Functions Menu
f - File Transfer Functions Menu
m - Memory Management Menu
p - Presentation Space Management Menu
e - Environment Functions Menu

q - Quit the HLLAPI Tutorial

Select a choice: _

The top portion of each menu screen lists the functions in the menu, and the bottom portion shows the menu-switching commands and other available response choices.

The region between the top and bottom portions of each screen is the user interaction area; you will be prompted to enter choices, and the results will be displayed in this area. You can respond to the "Select a choice:" prompt on each screen with a function number or with any of the valid commands listed at the bottom of the menu. Furthermore, if you respond to the "Select a choice:" prompt with a **!**, you will escape to the UNIX System shell program defined by the **SHELL** variable. If you respond with an **x**,

the latest parameters returned by a function are placed in a text file in hexadecimal format. The editor program defined by the **EDITOR** environment variable will then be invoked, and the text file will be displayed. Typing in an **x** at the "Select a choice:" prompt lets you see the non-displayable characters and save the results in files for later use.

For example, after you have run the AT&T 3270 Emulator+ HLLAPI function number 5 (Copy Presentation Space), the **x** command might produce the following display:

```
FUNCTION 5
DATA
0000 74686973 20697320 74686520 72657375      *this.is.the.resu*
0010 6c74206f 6620636f 70792070 72657365      *lt.of.copy.prese*
0020 6e746174 696f6e20 66756e63 74696f6e      *ntation.function*
0030 2063616c 6c                                     *call*

RETICODE
0000 34                                     *4*
```

The seven AT&T 3270 Emulator+ HLLAPI Tutorial menus that may be called from the HLLAPI Tutorial Main Menu are shown below.

Local Environment Functions Menu

- | | |
|-----------------------|------------------------------------|
| 1 - Connect PS | 16 - Work Station Control |
| 2 - Disconnect PS | 20 - Query System |
| 9 - Set Session Param | 21 - Reset System |
| 10 - Query Sessions | 22 - Query Session Status |
| 11 - Reserve | 113 - Connect and Interact with PS |
| 12 - Release | |

Select a choice: _

l - Loc Env Menu	f - File Trans Menu	e - Env Menu	x - hex return
c - Communi Menu	m - Memory Manag Menu	M - Main Menu	! - shell
k - Keyboard Menu	p - PS Manag Menu	q - quit	

Communications Functions Menu

- | | |
|--------------|------------------------------|
| 3 - Send Key | 23 - Start Host Notification |
| 4 - Wait | 24 - Query Host Update |
| 18 - Pause | 25 - Stop Host Notification |

Select a choice: _

l - Loc Env Menu	f - File Trans Menu	e - Env Menu	x - hex return
c - Communi Menu	m - Memory Manag Menu	M - Main Menu	! - shell
k - Keyboard Menu	p - PS Manag Menu	q - quit	

Keyboard Functions Menu

- 50 - Start Keyboard Intercept
- 51 - Get Key
- 52 - Post Intercept Status
- 53 - Stop Keystroke Intercept

Select a choice: _

l - Loc Env Menu	f - File Trans Menu	e - Env Menu	x - hex return
c - Communi Menu	m - Memory Manag Menu	M - Main Menu	! - shell
k - Keyboard Menu	p - PS Manag Menu	q - quit	

File Transfer Functions Menu

- 90 - Send File
- 91 - Receive File

Select a choice: _

l - Loc Env Menu	f - File Trans Menu	e - Env Menu	x - hex return
c - Communi Menu	m - Memory Manag Menu	M - Main Menu	! - shell
k - Keyboard Menu	p - PS Manag Menu	q - quit	

Memory Management Menu

17 - Storage Manager

Select a choice: _

l - Loc Env Menu f - File Trans Menu e - Env Menu x - hex return
c - Communi Menu m - Memory Manag Menu M - Main Menu ! - shell
k - Keyboard Menu p - PS Manag Menu q - quit

Presentation Space Management Menu

5 - Copy Presentation Space 31 - Find Field Position
6 - Search Presentation Space 32 - Find Field Length
7 - Query Cursor Location 33 - Copy String to Field
8 - Copy PS to String 34 - Copy Field to String
13 - Copy OIA 99 - Convert Position or RowCol
14 - Query Field Attribute 111 - Change Current PS Position
15 - Copy String to PS 112 - Write a Character in PS
30 - Search Field

Select a choice: _

l - Loc Env Menu f - File Trans Menu e - Env Menu x - hex return
c - Communi Menu m - Memory Manag Menu M - Main Menu ! - shell
k - Keyboard Menu p - PS Manag Menu q - quit

```
Environment Functions Menu

92 - Invoke UNIX Process
93 - UNIX Redirect

Select a choice: _

-----
l - Loc Env Menu   f - File Trans Menu   e - Env Menu   x - hex return
c - Communi Menu  m - Memory Manag Menu M - Main Menu  | - shell
k - Keyboard Menu p - PS Manag Menu   q - quit
```

To avoid switching back and forth between menus, you can execute any valid AT&T 3270 Emulator+ HLLAPI function from all menus, including the Main Menu; this is useful for experienced HLLAPI users, or when you must execute a sequence of HLLAPI calls.

Once you choose a function by entering a function number in response to the "Select a choice:" prompt, you will be asked to enter the calling parameters required by that function. For example,

Data<-

prompts you to enter the data string parameter,

Length<-

prompts you to enter the length parameter, and

Retcode<-

prompts you for the return code parameter.

Since not every function requires all parameters, you will only be asked to provide the parameters required by the function you have chosen. However, if the length parameter can be derived from the data string parameter, the AT&T 3270 Emulator+ HLLAPI Tutorial will automatically compute the length parameter and display it on the screen. You then have the option of changing the length parameter or pressing the return key to enter it. The parameters returned by the function will be displayed on the screen with

their labels once the function has executed.

The AT&T 3270 Emulator+ HLLAPI Tutorial will add a + character at the end of the returning parameters, if their content exceed the width of the physical screen; non-displayable characters will appear as spaces. Use the x command to view the entire contents of the returning parameters, and non-displayable characters.

Editing Mode

Whenever you are prompted for entries on any of the AT&T 3270 Emulator+ HLLAPI Tutorial menus, your terminal is placed in the tutorial editing mode. In this mode all characters typed, except the backspace, tilde, escape, and return characters, are immediately saved in an **hllapiprim** internal buffer. The normal UNIX System erase and kill characters (# and @) are not interpreted as such, and do not have to be preceded by a "\". The non-displayable characters (such as control characters) appear as spaces on the screen.

A backspace character moves the cursor backwards one position and erases the current character and all characters to its right. The return character ends the editing mode; the content of the buffer is taken as input to the current prompt, and the next prompt, if required, is displayed. In editing mode, you can not move the cursor beyond the rightmost position on the screen; a bell will sound if you try to.

When you enter the escape character from any position, the editor specified by the **EDITOR** variable will be invoked with the current buffer content. Use the escape character to enter characters beyond the rightmost screen position and to respond to menu prompts with the full-screen editor you prefer. This character is also useful if you must enter data for functions that require hundreds of characters or data that contains large numbers of non-displayable characters.

When you leave the editor, the file content, which you must write out if you made changes, will be transferred to the **hllapiprim** internal buffer and the original screen will be redisplayed with the new buffer contents. When the buffer content is read by the editor, newline characters will be appended to break it into lines for readability and to circumvent possible line-length limitations; when files are copied back to the buffer, these newline characters will be removed. If you have to enter a newline character into the buffer from the file, place it on a line by itself.

If the escape character is preceded by a tilde character, the buffer content will be converted into hexadecimal format before the editor is invoked. The hexadecimal format file content is similar to the result of executing the "x" command: each line shows the address, hexadecimal displays of 16 characters followed by their corresponding character displays.

Once inside the editor, the hexadecimal display portion can be edited using hex representations of characters. Note that when editing the hexadecimal file, the file format must be preserved in order for the file to be converted properly back to the internal buffer; i.e., hex codes must be in groups of 4 characters separated by space characters and each line should have six characters before the first hexadecimal code, although the address portion is never used when file content is transferred to the internal buffer. This tilde-escape editing makes it possible for a user to enter any character as part of a function parameter input, including the four editing control characters and non-ASCII characters with embedded null characters. For example, using the tilde-escape editing, the storage address returned by the HLLAPI function number 17 (H_STMAN - Storage Manager) can be entered as part of the data parameters for function number 23 (H_STRTHOST - Start Host Notification).

You can interrupt the editing mode with the **interrupt** signal or with the **quit** signal. The **interrupt** signal ends the current editing mode and moves the cursor to the "Select a choice:" prompt; all input parameters that you entered will be discarded, and the chosen function will not be executed. The **quit** signal ends the AT&T 3270 Emulator+ HLLAPI Tutorial program.

Environment Variables

The **hllapiprim** and **hllapisec** programs communicate through two named pipes that they create if needed. The location of these two named pipes is, by default, the **\$HOME** UNIX System directory; if the **HOME** variable is not defined or has no value, the current working directory will be used. The default file names for the two named pipes are **.hlltutp.1** and **.hlltutp.2**; however, you can set the environment variables **HLLTUTDIR** and **HLLTUTP** to override the default pipe names and locations. The AT&T 3270 Emulator+ HLLAPI Tutorial will add the suffixes **.1** and **.2** to the names defined by **HLLTUTP**; therefore, you must restrict **HLLTUTP** names to a length of 12 characters.

The AT&T 3270 Emulator+ HLLAPI Tutorial uses two additional environment variables; **EDITOR**, and **SHELL**. **EDITOR** determines the text editor program that the AT&T 3270 Emulator+ HLLAPI Tutorial will use (e.g., **ed** or **vi**); the default is **ed**. **SHELL** determines the shell program that will be used when you escape to the UNIX System while running the AT&T 3270 Emulator+ HLLAPI Tutorial; the default is **sh**.

You should also initialize other environment variables and/or files, as required by the AT&T 3270 Emulator+. (See the *AT&T 3270 Emulator+ System Administrator's Guide* for further information on this subject.)

4 Application Program Interface

Introduction	4-1
BSC Operation	4-2
SNA Operation	4-2
Configuration Files	4-3
Include Files	4-3

The API Tutorial	4-4
Running the API Tutorial Package	4-4
▪ Running with Two Terminals	4-5
▪ Running with One Terminal	4-7

Application Development	4-8
API Execution	4-8
Application Program Development	4-8
▪ Application Program Format	4-8
▪ Linking with the API Library	4-10
▪ The Application Program Environment	4-11
▪ Signals	4-11
▪ 3270 Data Stream Mode	4-11
▪ Raw Mode	4-12
▪ Multi-Session	4-12
▪ Host Interaction	4-13

Introduction

The API provides two modes of accessing a remote host: in 3270 data stream mode or in "raw" data stream mode. In 3270 data stream mode, the application program interface to the remote system is functionally equivalent to the interactive user interface to the host through a 3278/9 terminal. The raw mode provides a block data transfer interface, independent of any terminal functionality.

The following calls constitute the API interface:

- xlu2clos(3X)** power off the logical unit
- xlu2ctl(3X)** logical unit control functions
- xlu2func(3X)** perform special functions on logical unit
- xlu2gets(3X)** get a string from the LUD_3270 logical unit's screen buffer
- xlu2info(3X)** obtain 3278/9 status line and cursor position information
- xlu2init(3X)** initialize terminal function library
- xlu2intr(3X)** interrupt an API call
- xlu2open(3X)** power on the logical unit
- xlu2puts(3X)** put a string to the LUD_3270 logical unit's screen buffer
- xlu2read(3X)** read the next raw segment on the LUD_RAW/LUD_TRAW logical unit
- xlu2seek(3X)** position the cursor to a field in screen buffer for LUD_3270 logical unit
- xlu2writ(3X)** write a raw segment on the LUD_RAW/LUD_TRAW logical unit

where:

- logical unit** is an SNA or a BSC terminal port. Logical unit is used generically, and it applies to both SNA and BSC.
- LUD_3270** specifies that the application program is using a terminal interface to the remote system

- LUD_RAW** specifies that the application program is using a block data interface to the remote system; API does not examine or alter the data except for BSC control codes, which are stripped before the data are passed to an application. In the case of SNA, API presents the RU to the application
- LUD_TRAW** specifies that the application program is using a transparent block data interface to the remote system; API does not examine or alter the data except for BSC control codes, which are stripped before the data are passed to an application

BSC Operation

When operating in BSC mode, 3274 controller functions are performed by the BSC controller process, **TM3274**, and the communication board **b3274** program.

TM3274 performs the device interface functions of a 3274 cluster controller, including establishing internal communication with each device emulation process as it powers up, and validating and/or assigning device addresses to devices as they power up. **TM3274** interfaces with **b3274**, via the driver, to transfer data between the emulated device and communications line, and it also handles the transfer of screen contents during copy commands.

b3274 performs the protocol level functions of an IBM 3274-51C cluster controller operating in BSC mode, including data stream validation, terminal status maintenance, and reporting. It handles polling and device selection, and it also controls BSC timing functions, including timing of line activities and sync insertion.

SNA Operation

When operating in SNA mode, 3274 controller functions are performed by the SNA process, **SNA**, and the communication board **sdlc** program.

SNA emulates the SNA functions of an IBM 3274-51C cluster controller. It provides support for all protocols associated with the Path Control, Transmission Control (or Connection Point Manager), Session Control, Data Flow Control, and Presentation Services layers of SNA.

sdlc emulates the link-level SDLC protocols as implemented on the cluster controller.

Configuration Files

An API application uses the following configuration files:

- screen control customization object file
- keyboard mapping customization object file
- message object file.

In addition, the controller configuration object file (BSC or SNA) must be set up.

Include Files

Application programs are required to include the header file **xlu2io.h**, which contains all required definitions and includes all other required files. Other files of interest to developers are:

- xapi.h** includes most definitions
- ctrl.c** controls the components of the API function library (see "Linking with the API Library," below) to be linked.

The API Tutorial

The API Tutorial is intended to assist API programmers in gaining familiarity with API function calls. The tutorial allows you to enter API calls from a terminal and observe the results. The package operates interactively; a menu allows you to choose the call you want, and submenus allow you to specify options to the call.

The following files are contained in the tutorial package:

- primary** shell script
- secondary** shell script
- mast**
- slv**
- README** contains instructions
- show_it**

This section describes how to use the API Tutorial; installation of the package is not a prerequisite for developing or running API application programs. Refer to the appropriate *AT&T 3270 Emulator+ Release Notes* for installation instructions.

Running the API Tutorial Package

You can run the tutorial using one or two terminals. If one terminal is used, only part of the results of each API call can be displayed interactively following the execution of the call. The remainder of the output can be sent to a file you designate and can be examined after the tutorial program is terminated.

If two terminals are available, the **primary** script is run on one terminal and the **secondary** script on the other. If only one terminal is available, the **secondary** script should be run in the background with output directed to a file.

Before you run these programs you must have the 3274 controller process (SNA or BSC) running, and you should also have read the *AT&T 3270 Emulator+ Application Programmer's Guide* and have it available for reference. You must also create two named pipes in the directory that contains the tutorial. To create the pipes, you do the following:

-
- Step 1. Log in as root.
- Step 2. Enter the following sequence of commands:

```
cd the_directory_where_you_have_stored_the_Tutorial  
  
mknod pip1 p  
  
mknod pip2 p
```

Running with Two Terminals

If the package is to be run from two terminals, you must perform the following steps. (The two terminals are referred to as "T1" and "T2.")

- Step 1. Log in on T1 and T2.
- Step 2. On both T1 and T2, type
- ```
cd the_directory_where_these_files_are_stored
```
- Step 3. On T1, type
- ```
./secondary a b c .
```

where:

a is the screen file name
b is the keyboard file name
c is the name of the controller pipe

For example:

a could be **4410** (from SC.4410)
b could be **std** (from KY.std)
c could be **/tmp/p3274.4**

and you would enter:

```
./secondary 4410 std /tmp/p3274.4
```

Step 4. On T2, type

```
./primary
```

This causes the following primary menu to be displayed:

```
API SYSTEM CALL EXERCISER

0.    Exit from program

1.  xlu2open      2.  xlu2clos
3.  xlu2ctl      4.  xlu2gets
5.  xlu2puts     6.  xlu2seek
7.  xlu2info     8.  xlu2init
9.  xlu2func     10. xlu2write
11. xlu2read

Enter Option :
```

If you select any option from 1 through 11 a series of menu sub-screens will be displayed, each requesting you to enter a parameter associated with the call. After you have entered the last parameter, the output of the call (i.e. the return value of the call and the return value of certain parameters associated with the call) is displayed, generally followed by the primary menu.

Step 5. You may now request execution of API calls. Your first two calls must be **xlu2init** followed by **xlu2open**.

Step 6. The purpose of T1 is to display additional output from each API call as it is executed. To cause the API to display the output, you must use LUV_TRC or LUV_DSP as the *luwmod* parameter in the **xlu2open** call. Once the **xlu2open** call has been made, the screen on T1 will display the results of subsequent API calls.

You can also display this output by using the same values for the *luwmod* parameter in an **xlu2ctl** call.

- Step 7. You can now continue with other API calls, eventually closing each session you open with an **xlu2clos** call.
- Step 8. The tutorial continues to run even after all open sessions have been closed with calls to **xlu2clos**. It may be terminated by selecting option "0" in the primary menu. If you terminate the tutorial, any sessions still open from T2 will be closed.

Running with One Terminal

If the package is to be run from one terminal, both the **primary** and **secondary** scripts must be run from this terminal, and any output from an API call must be directed to a file. The following steps are required:

- Step 1. Type

```
./secondary a b c > logfile &
```

where *a*, *b*, and *c* are the same as in Step 3 above, and *logfile* is the file in which you want the output to be placed.

- Step 2. Type

```
./primary
```

to display the series of menus described in Step 4 above. Your first two calls must be **xlu2init** followed by **xlu2open**. You must use LUV_NTD for the *lvmmod* parameter in the **xlu2open** call.

- Step 3. You can now continue with other API calls as in Steps 7 and 8 above.

- Step 4. At the completion of the tutorial, you can enter

```
cat logfile
```

to obtain the output of the API calls you made.

Application Development

The API user does not interface with any of the protocol levels of SNA or BSC. The application program interface can be an emulated formatted screen, an unformatted screen, or a data block (raw mode). The general procedure for developing an API application is to write an application program using API function calls and then to link the compiled program with the API library.

API Execution

The associated SNA or BSC controller process must be running before you execute an API application. The application may be executed as a background or a foreground process. If you do not choose to view the screen, the terminal emulation application program may be initiated from the command line in the background, without monopolizing a terminal to execute the functions. However, if you prefer to view the application's interaction with the host as it occurs, the program requires a physical terminal and should be run in the foreground (see `xlu2ctl(3X)`).

Application Program Development

API programs are written in the C programming language. The following sections describe the development process in detail.

Application Program Format

An API user program takes the following general form:

```
#include <xlu2io.h>                /* API header file */
extern int errapi;                /* API error codes */
main (argc, argv)                /* user entry point */
int argc;
char *argv[];
{
    unsigned char luchan1;
    .....

    if ((xlu2init(... ) == -1)    /* initialize emulator */
        ... ; /* error */
```

continued

```

if ((xlu2open(&luchan1, ... ) == -1) /* power on the logical unit and
                                acquire logical unit channel */
    ... ; /* error */

```

```

.....

```

For 3270 data stream mode:

```

.....
if ((xlu2ctl(&luchan1, LUEVENT, KY_NMSG ... ) == -1) /* await next message */
    ... ; /* error */

if ((xlu2seek(&luchan1, ... ) == -1) /* position cursor to field for reading */
    ... ; /* error */

if ((xlu2gets(&luchan1, ... ) == -1) /* get a string from screen buffer */
    ... ; /* error */

```

user processing

```

.....
if ((xlu2seek(&luchan1, ... ) == -1) /*position cursor to field for writing*/
    ... ; /* error */
if ((xlu2puts(&luchan1, ... ) == -1) /* put a string to screen buffer */
    ... ; /* error */

```

```

.....
if ((xlu2ctl(&luchan1, LUEVENT, KY_ENTER ... ) == -1) /* transmit screen */
    ... ; /* error */

```

or, for raw mode:

```

.....
if ((xlu2ctl(&luchan1, LUEVENT, KY_NMSG ... ) == -1) /* await next message */
    ... ; /* error */
if ((xlu2read(&luchan1, ... ) == -1) /* read a data segment */
    ... ; /* error */

```

user processing

```

.....
if ((xlu2writ(&luchan1, ... ) == -1) /* write a data segment to host */
    ... ; /* error */

```

```

.....

```

continued

```
if ((xlu2clos(&luchan1) == -1); /* power off logical channel */
    ... ; /* error */
return
}
```

Linking with the API Library

When you link the API library with an application program, the resulting interface to the SNA or BSC controller is equivalent to the 3278/9 terminal emulation process. A user application consists of a process containing the user application program and the API function library.

You must compile the program with the specified API function library directory pathname. To save runtime memory, you may elect to exclude subsets of the API function library when linking the application object file(s). Library usage is determined by the emulation modes (and the associated functionality) required in the application program. The library subsets are:

- without 3270 data stream processing (RAW)
- without 3279 screen output generation (VNON)
- with all functionality included

`ctrl.c` controls the selection of the API function library components in generating one of the above subsets. The user specifies the functionality to be excluded by using `-D` in the command line followed by the name of the excluded subset. The following command compiles and links a sample user API program without 3270 data stream processing (RAW) and without 3279 screen output generation (VNON). `api` and `gemusr` contain API library runtime functions.

```
cc -o sample sample.c -DRAW -DVNON /usr/lib/ctrl.c
    -lapi -lcurses -lgemusr
```

The Application Program Environment

API controls the emulated terminal based on the physical attributes of standard input. Terminal emulator functions write the screen buffer to standard output. Therefore, when screen update is enabled, standard input and standard output may be redirected, but standard input must be a terminal device.

Signals

An API application program can receive any signal except SIGPWR, which is reserved. When a signal is caught during execution of an API application, the signal catching routine cannot call any other API functions (except `xlu2intr`) since the signal may have occurred during execution of an API function and API does not allow concurrent calls from the same process.

Signals do not automatically cause API function calls to be interrupted. `xlu2intr` is provided to allow a signal catching routine to interrupt execution of certain other (mainline) API functions.

If screen update is enabled and an incoming signal interrupts an API function call, and if the signal catching routine terminates the process, the physical terminal (not the emulated terminal) may be left in raw mode. If an API application signal catching routine terminates a process, it is recommended that the physical terminal be removed from raw mode (see ICANON in `termio(7)`).

3270 Data Stream Mode

The following must be considered when using the 3270 data stream mode for screen updating:

- Data are exchanged between the host and the logical unit through the screen buffer.
- The logical unit's screen buffer may be examined or altered, one field at a time, using `xlu2gets` or `xlu2puts`, respectively.
- In this mode, the logical unit has the functionality of a terminal emulation session.

Raw Mode

The following must be considered when using raw mode:

- Data are exchanged between the host and the logical unit in raw data block units.
- When data are received from the host for a raw mode logical unit, the data are acknowledged without any data stream analysis.
- Raw data blocks are retrieved with `xlu2read` and transmitted with `xlu2writ` API function calls.
- The application program must control the sequencing of the raw data blocks to form chains of blocks and/or commands.
- Outbound raw data from the host are received and queued by API internally; the queued data are provided to the user a segment at a time with each `xlu2read` call.
- The application program must issue an `xlu2read` call periodically in order not to overflow the internal queue of raw data from the host. `E_DOVFLO` is returned in `errapi` if overflow occurs on the logical unit while executing an API function call on a channel on which overflow has occurred. Otherwise, `E_DOVFLO` is indicated in the `err_cond` field of `XLU2IBUF` (see `xlu2info`) for the logical unit on which data were lost.

Multi-Session

An API application can establish up to four sessions with one or more hosts. This is done by making up to four `xlu2open` calls and using the returned value of `lu_chan` (the parameter that specifies the logical unit) in subsequent API function calls.

In general, the value of `lu_chan` returned by an API call will be the same as the called value. However, under certain circumstances, the called and returned value may differ. All API calls, except `xlu2init` and `xlu2open`, must be issued with an active `lu_chan` (i.e., one obtained from an `xlu2open` call). The call may return with an `lu_chan` different from the one used to issue the call under the following conditions:

- Only `xlu2read` and `xlu2ctl` with `LUEVENT` or `LUEVIMED` commands can return successfully with a different `lu_chan`.

- The `xlu2read` call returns with the `lu_chan` which has the oldest queued data.
- An `xlu2ctl` call, with an LUEVIMED command which takes an argument of `KY_NMSG` will, if any data are present, return with the `lu_chan` which corresponds to a session that has the oldest queued data.
- An `xlu2ctl` call, with an LUEVENT command which takes arguments of `KY_PEND` or `KY_WAIT`, returns with the `lu_chan` that first fulfills the request.
- An `xlu2ctl` call, with an LUEVENT command which takes arguments of `KY_ENTER`, `KY_CLEAR`, `KY_SYS_REQ`, `KY_PAn` or `KY_PFn` performs the requested function on the `lu_chan` input to the call but returns to the application program with the `lu_chan` that first achieves keyboard unlock.
- All other API calls will only return a different `lu_chan` on failure, and the failure must be an `E_TTY` or `E_CTRLIO` error on the other channel.
 - Before API will return a failure on a different `lu_chan`, it will first validate all the parameters in the original call and, if the call required interaction with the controller process, wait for the controller process to acknowledge receipt of the call information
 - An `E_TTY` or `E_CTRLIO` error on another channel does not necessarily indicate failure of the call on the input channel.

Host Interaction

The user application program uses API function calls to perform screen updates or host interactions. The API function terminates with a `return` statement. Communication with the controller and/or the host (by use of `xlu2open`, `xlu2ctl` with an LUEVENT command, `xlu2writ`, `xlu2read`, or `xlu2clos` functions) is conducted in a half-duplex manner, i.e., the API function that initiates the communication does not return control to the user application program until the controller/host completes processing. In the case of `xlu2open` and `xlu2ctl` with an LUEVENT command, "keyboard unlock" must also occur.

Keyboard unlock signifies that communication with the host is no longer in progress. This is indicated by the disappearance of the WAIT message displayed by the terminal emulation process, in the Operator Information Area, during communication. It does not reflect the keyboard's lock/unlock state due to an inhibit condition such as:

- an invalid operation (e.g., illegal function)
- field overflow
- bad key translation
- wrong screen location access
- non-numeric data entered into a numeric field
- keyboard disabled by the host

All the inhibit conditions listed above are cleared and true keyboard unlock is achieved before returning control to the user.

It is not ensured that the requested API function has executed successfully if it results in an inhibit condition or a 480 program check error (see Appendix E). For all conditions except the last (keyboard disabled), the API call will fail with an error indicating the cause of the inhibit. If the keyboard is disabled by the host, the call will not fail for this condition. In this case, the user must determine the occurrence of the inhibit by examining `inhb_lck` and `opia` in the `XLU2IBUF` structure for the session (see `xlu2info`).

For example, if the host disables the keyboard while an `xlu2ctl` call to send the ENTER key is being processed, the user program may have to reissue the same `xlu2ctl` call. Error messages for each API function call are listed on the manual page for that call and summarized in the Appendices.

All `xlu2ctl` calls (except `xlu2ctl` with an `LUEVIMED` command) that perform functions resulting in interaction with the host or the controller wait for "keyboard unlock" before returning to the application program.

The `xlu2ctl` call with an `LUEVENT` command, which takes arguments of `KY_NMSG`, `KY_PEND` or `KY_WAIT`, waits for its respective conditions to be satisfied before returning. Therefore, the application program must be sensitive to the state the `lu_chan` session is in before and after one of the above `xlu2ctl` requests is executed. `xlu2writ` does not wait for keyboard unlock, but it does wait for the controller to acknowledge the transmitted data.

Manual Pages

Introduction

1

Introduction

All function calls are based on this common format:

- The **NAME** part provides the name of the function call.
- The **SYNOPSIS** part summarizes the use of the function call. The following conventions apply:
 - **Boldface** strings are literals and are to be typed just as they appear
 - *Italic* strings usually represent substitutable argument prototypes and program names found elsewhere in the manual
 - Square brackets [] around an argument prototype indicate that the argument is optional. When an argument prototype is given as 'name' or 'file,' it always refers to a *file* name.
 - Ellipses ... are used to show that the previous argument prototype may be repeated
 - A final convention is used by the commands themselves. An argument beginning with a minus -, plus +, or equal sign = is often taken to be some sort of flag argument, even if it appears in a position where a file name could appear. Therefore, it is unwise to have files whose names begin with -, +, or =.
- The **DESCRIPTION** part explains the function call at hand, and the calling parameters.
- The **SEE ALSO** part gives you pointers to related information.
- The **NOTES** section points out important technical information and/or areas where you should exercise caution.
- The **RETURN CODES** section explains the returning parameters.

The HLLAPI and API function calls are organized alphabetically. All HLLAPI function calls begin with H_ prefix, while all API function calls begin with XLU2.

NAME

h_chcur – change cursor position in presentation space

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

h_chcur allows the application program to change the cursor position in the current connected presentation space.

Calling arguments:

- *func* points to the symbolic H_CHCUR
- *position* points to the destination cursor position in the current connected presentation space
- *data* and *length* are not applicable to *h_chcur*

SEE ALSO

h_qsys(3X).

RETURN VALUES

h_chcur returns one argument: a return code, defined in the *<xhllapi.h>* header file. The return code is placed at the location pointed to by the *position* calling argument, and is also returned as the function value for the *h_chcur* call.

The return code for *h_chcur* will have one of these values:

- HE_SUCCESS: *h_chcur* was successful
- HE_INVALID: an invalid presentation space was specified
- HE_POS: the specified presentation space position was invalid
- HE_SYSERR: a system error was encountered; call the *h_qsys* function to find out the reason for failure

NAME

`h_connect` – connect presentation space

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

`h_connect` establishes a connection between your HLLAPI application program and a specified presentation space. Your application program can only connect to one presentation space at a time, even though there are four sessions available to you.

If you issue two `h_connect` function calls in a row, an automatic disconnect takes place before the second call is issued, and the first call is nullified even if the second one fails.

You do not have to call `h_connect` before using these functions:

- `h_setparms` - set session parameters
- `h_qsess` - query session
- `h_wsctrl` - WS Ctrl
- `h_stman` - storage manager
- `h_qsys` - query system
- `h_reset` - reset system
- `h_qstatus` - query session status
- `h_qhost` - query host update
- `h_startkey` - start keystroke intercept
- `h_getkey` - get key
- `h_postint` - post intercept status
- `h_stopkey` - stop keystroke intercept
- `h_send` - send file
- `h_rcv` - receive file
- `h_invoke` - invoke UNIX system command or program
- `h_redir` - escape to the UNIX system
- `h_conv` - convert position or RowCol

Calling arguments:

- `func` points to the symbolic H_CONNECT.
- `data` points to the short name of the target presentation space. The presentation space short name is a capital letter from A through Z, and it is the

first field for the entry in the LUTABLE environment file that corresponds to the target presentation space.

- *length* and *position* are not applicable to *h_connect*.

SEE ALSO

h_setparms(3X), *h_qsys*(3X).

NOTES

The parameters CONPHYS, CONLOG, DISPLAY and NODISPLAY under *h_setparms* affect connect.

If you specified the CONPHYS option with *h_setparms*, *h_connect* establishes a physical connection with the requested presentation space and transfers control to the user at the terminal. If you specified the CONLOG option, *h_connect* establishes a logical connection and the HLLAPI application program will retain control.

DISPLAY allows the terminal user to view the results of your HLLAPI application program during an *h_connect* function call (if you specified the CONLOG option, above), while NODISPLAY does not.

RETURN VALUES

h_connect returns one argument: a return code, defined in the *<xhllapi.h>* header file. The return code is placed at the location pointed to by the *position* calling argument, and is also returned as the function value for the *h_connect* call.

The return code for *h_connect* will have one of these values:

HE_SUCC:	<i>h_connect</i> was successful, and the selected presentation space is unlocked and ready for input
HE_INVALID:	an invalid presentation space was specified
HE_BUSY:	<i>h_connect</i> was successful, but the presentation space is busy
HE_INHBT:	<i>h_connect</i> was successful, but the presentation space is locked, i.e., input is inhibited
HE_SYSERR:	a system error was encountered; call the <i>h_qsys</i> function to find out the reason for failure
HE_RSRC:	This resource is unavailable, and the requested presentation space is in use by another session; try again later

NAME

`h_connint` – connect and interact with presentation space

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

`h_connint` connects your HLLAPI application program to the specified presentation space in the same way that the `h_connect` would, and in addition, it transfers control to the user without application program involvement. The user can enter manual transactions interactively with the host session (e.g., a logon sequence), until an escape sequence is entered at the keyboard (ESC FD, unless customized in a different way). At this point, `h_connint` returns control to the application program, and the instruction following the `h_connint` call will be executed.

Calling arguments:

- `func` points to the symbolic H_CONNINT
- `data` points to the target presentation space short name. The presentation space short name is a capital letter from A through Z, and it is the first field for the entry in the LUTABLE environment file that corresponds to the target presentation space.
- `length` and `position` are not applicable to `h_connint`

SEE ALSO

`h_qsys(3X)`.

NOTES

`h_connint` is an AT&T 3270 Emulator+ HLLAPI extension; it is not part of the IBM 3270 Personal Computer HLLAPI specification.

RETURN VALUES

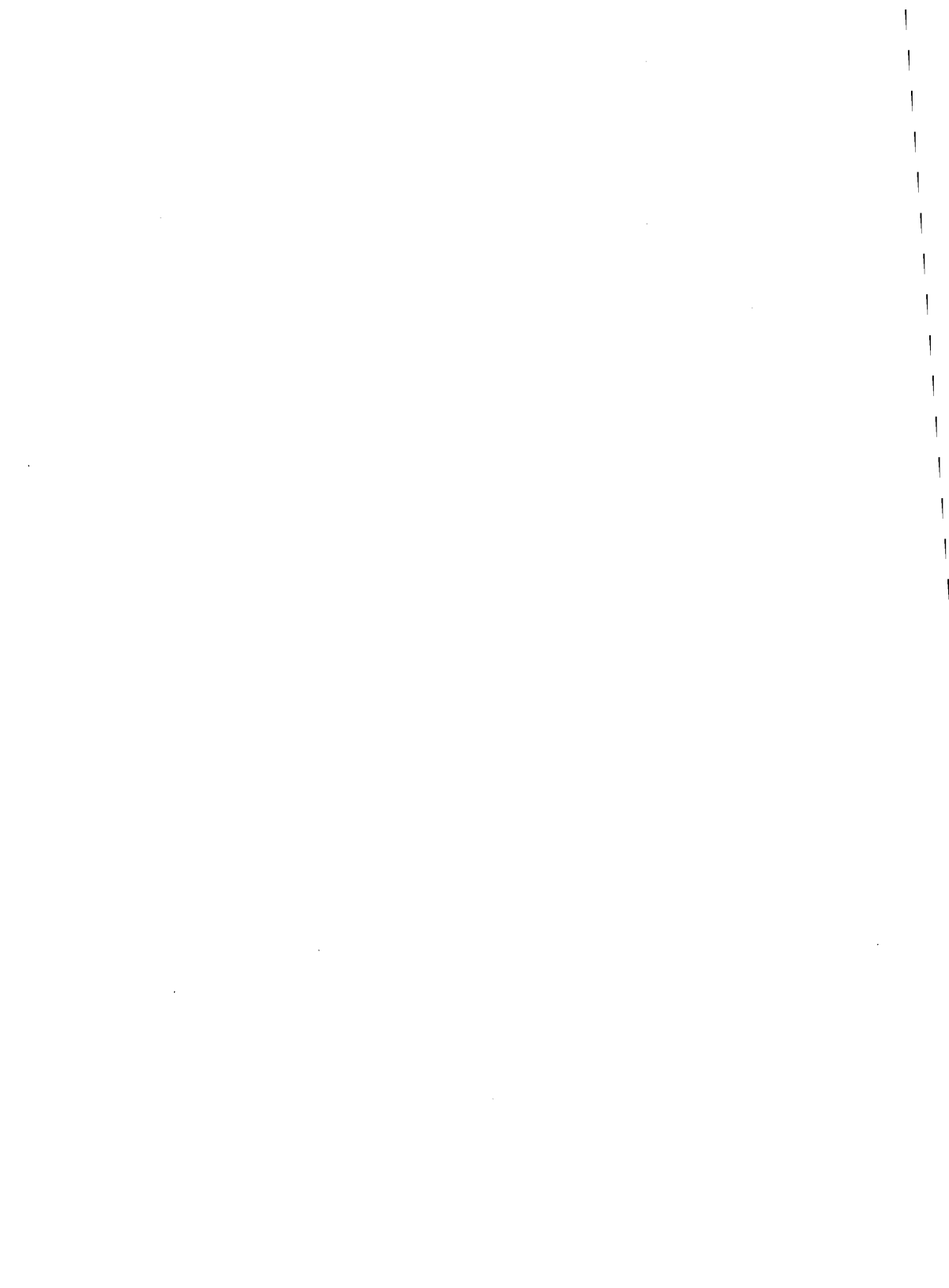
`h_connint` returns one argument: a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the `position` calling argument, and is also returned as the function value for the `h_connint` call.

The return code for `h_connint` will have one of these values:

HE_SUCCESS: `h_connint` was successful

HE_INVAL: an invalid presentation space was specified

HE_SYSERR: a system error was encountered; call the `h_qsys` function to find out the reason for failure



NAME

`h_conv` – convert position or row/column

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
```

```
int *func;
```

```
char *data;
```

```
int *length;
```

```
int *position;
```

DESCRIPTION

The `h_conv` function converts a presentation space positional value into the display row/column coordinates, or the display row/column coordinates into the presentation space positional value, without changing the cursor position. `h_conv` takes into account the model number for the emulated host display type, and the corresponding presentation space size, when it makes the conversion.

Calling arguments:

- `func` points to the symbolic H_CONV.
- `data` points to the presentation space short name and "P" for convert position, or to the presentation space short name and "R" for convert row/column. The presentation space short name is a capital letter from A through Z, and it is the first field for the entry in the LUT-ABLE environment file that corresponds to the target presentation space.
- `length` points to the row in the presentation space.
- `position` points to the column in the presentation space.

SEE ALSO

`h_qsys(3X)`.

RETURN VALUES

`h_conv` returns two arguments:

- the first argument is the row number, or "0" for incorrect input. This value is placed at the location pointed to by the `length` calling parameter.
- the second argument is a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the `position` calling argument, and is also returned as the function value for the `h_conv` call. If you have established a common error handling routine, be sure to take into account that the `return code` for `h_conv` is really a `status code`.

The return code for `h_conv` will have one of these values:

```
HE_SUCCESS:      incorrect input was provided
```

```
> HE_SUCCESS:    PS position or column
```

HE_PSID: an invalid presentation space ID was specified, or
the presentation space was never connected

HE_NOTPR: the second character in the data string is not "P" or
"R"

NAME

`h_copy` – copy presentation space

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

The `h_copy` function copies the contents of the current connected presentation space into a data area, specified in your HLLAPI application program.

The `h_copy` function translates the characters in a host presentation space into ASCII, and translates the attribute bytes and other characters not represented in ASCII into blanks. You can specify to print the original character values by using the ATTRB option under `h_setparms` (ATTRB - pass back all codes that do not have an ASCII equivalent, except EABs, as their original values; NOATTRB - change all unknown values to blanks).

Extended attribute bytes are not copied unless you have specified EAB with `h_setparms` (EAB: pass the presentation space data with extended attribute bytes; NOEAB: pass only data).

Calling arguments:

- `func` points to the symbolic H_COPY
- `data` points to the target area, which should be the size of the presentation space that you want to copy
- `length` and `position` are not applicable to `h_copy`

SEE ALSO

`h_copypps(3X)`, `h_qsys(3X)`, `h_setparms(3X)`.

NOTES

The target area for the copy must be twice the size of the presentation space if extended attribute bytes will be copied.

If you want to copy a portion of a presentation space only, use `h_copypps`.

RETURN VALUES

`h_copy` returns two arguments:

- the first argument is the target area that contains the copied presentation space. This area is placed at the location pointed to by the `data` calling parameter.
- the second argument is a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the `position` calling argument, and is also returned as the function value for the `h_copy` call.

The return code for `h_copy` will have one of these values:

- HE_SUCCESS: the presentation space contents were copied to the location specified by your application program; the target presentation space was active, and the keyboard was unlocked
- HE_INVALID: the presentation space was not connected, and the copy results are undefined
- HE_BUSY: the presentation space contents were copied, and the connected presentation space was waiting for host response
- HE_INHBT: the presentation space was copied, and the keyboard was locked
- HE_SYSERR: a system error was encountered; use *h_qsys* to find out the reason for failure

NAME

`h_copyyss` – copy presentation space to string

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

`h_copyyss` copies all or part of the current connected presentation space into a data area, previously defined in your HLLAPI application program.

The offset of the string into the presentation space is based on defining the upper left corner (row 1, column 1) as location 1, and the bottom right corner as the maximum screen size for the presentation space. The sum total of the offset plus the string length, cannot exceed the maximum screen size for the presentation space.

`h_copyyss` translates the character(s) in a host presentation space into ASCII, and the attribute bytes and other characters not represented in ASCII into blanks. If you do not want the attribute bytes translated to blanks, you could request that the original values be copied using the ATTRB option under `h_setparms`.

Extended attribute bytes are not copied unless you specified EAB under `h_setparms`.

Calling arguments:

- `func` points to the symbolic H_COPYPSS
- `data` points to the target data string
- `length` points to the length of `data`
- `position` points to the beginning offset of `data`

SEE ALSO

`h_qsys(3X)`, `h_setparms(3X)`.

NOTES

The sum value of the position offset + length cannot exceed the maximum size of the presentation space.

RETURN VALUES

`h_copyyss` returns one argument: a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the `position` calling argument, and is also returned as the function value for the `h_copyyss` call.

The return code for `h_copyyss` will have one of these values:

HE_SUCCESS: the presentation space contents were copied to the application program, the target presentation space was active, and the keyboard was unlocked

HE_INVALID:	your HLLAPI program was not connected to a presentation space, and the copy results are undefined
HE_PARM:	an error was made in specifying the string length
HE_BUSY:	the presentation space contents were copied, and the connected presentation space was waiting for host response
HE_INHBT:	the presentation space was copied, and the keyboard was locked
HE_POS:	the specified presentation space position is invalid
HE_SYSERR:	a system error was encountered; call the <i>h_qsys</i> function to find out the reason for failure

NAME

`h_cpfield` – copy field to string

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

The `h_cpfield` function transfers characters from a field within the current connected presentation space into a string. You can use `h_fndpos` to find the position of the source field, and `h_fndlen` to find its length.

You can use `h_cpfield` with both protected or unprotected fields in a field-formatted presentation space, i.e., a host presentation space.

The `h_cpfield` function ends when one of the following conditions are met:

- when it reaches the end of the source field
- when it exceeds the length of the target string
- when it reaches the end of the presentation space

Calling arguments:

- `func` points to the symbolic `H_CPFIELD`
- `data` points to the target data string
- `length` points to the length of the value pointed to by `data`
- `position` points to the position of the source field in the presentation space that will be copied

SEE ALSO

`h_fndlen(3X)`, `h_fndpos(3X)`, `h_qsys(3X)`.

NOTES

No forward wrapping will occur, but there will be backwards wrapping if the given position is located before the start of a field in the presentation space.

RETURN VALUES

`h_cpfield` returns one argument: a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the `position` calling argument, and is also returned as the function value for the `h_cpfield` call.

The return code for `h_cpfield` will have one of these values:

- `HE_SUCCESS`: `h_cpfield` was successful
- `HE_INVAL`: an invalid presentation space was specified; i.e., it was not connected, not configured, or it was labeled with an invalid name

HE_PARM: a parameter error was encountered

HE_LENGTH: the data to be copied and the target string were not the same size. The data may not have been truncated because the string length may have been larger than the field copied

HE_POS: the specified presentation space position is invalid

HE_SYSERR: a system error was encountered; call the *h_qsys* function to find out the reason for failure

NAME

`h_cpoia` – copy OIA

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
struct cpoia *data;
int *length;
int *position;
```

DESCRIPTION

`h_cpoia` returns the operator information area (OIA) data from the current connected presentation. The OIA data provides information about the status of your work station and the IBM host computer.

Calling arguments:

- `func` points to the symbolic H_CPOIA
- `data` points to a structure of type `cpoia`, where the information returned by `h_cpoia` will be placed. `cpoia` is defined in the `<xhllapi.h>` header file as follows:

```
typedef unsigned char uchar;
```

```
/* Copy OIA data string */
```

```
typedef struct {
```

```
char    cp_format;           /* OIA Format byte */
char    cp_image[80];       /* OIA Image Group */
```

```
/* OIA Indicator Group */
```

```
/* Group 1: On-line and screen ownership */
```

```
#define SETUP      0x80      /* Setup mode */
#define TEST       0x40      /* Test mode */
#define SSCPOWN    0x20      /* SSCP-LU session owns screen */
#define LUOWN      0x10      /* LU-LU session owns screen */
#define UNOWN      0x08      /* Online and not owned */
#define READY      0x04      /* Subsystem ready */
uchar    group_1;
```

```
/* Group 2: Character selection */
```

```
#define EXTEND     0x80;     /* Extended Select */
#define APL        0x40;
#define KANA       0x20;
#define ALPHA      0x10;
#define TEXT       0x08;
uchar    group_2;
```

```

/* Group 3: Shift state */

#define NUMERIC    0x80;        /* Numeric Shift */
#define SHIFT      0x40;        /* Upper Shift */
    uchar    group_3;

/* Group 4: PSS group 1 */
/* Group 5: Highlight group 1 */
/* Group 6: Color group 1 */

#define SELECT     0x80;        /* Operator Selectable */
#define INHERIT    0x40;        /* Field Inherit */
    uchar    group_4;
    uchar    group_5;
    uchar    group_6;

/* Group 7: Insert */

#define INSERT     0x80;        /* Insert mode */
    uchar    group_7;

/* Group 8: Input inhibited */

/* Group 8: Byte 1 */

#define CHECK      0x80;        /* Non-resettable machine check */
#define KEY        0x40;        /* Reserved for security key */
#define MACHINE    0x20;        /* Machine Check */
#define COMM       0x10;        /* Communications Check */
#define PROGRAM    0x08;        /* Program check */
#define RETRY      0x04;
#define NWORKING   0x02;        /* Device not working */
#define VBUSY      0x01;        /* Device very busy */

/* Group 8: Byte 2 */

#define BUSY       0x80;        /* Terminal busy */
#define WAIT       0x40;        /* Terminal wait */
#define SYMBOL     0x20;        /* Minus symbol */
#define FUNCTION   0x10;        /* Minus function */
#define TOOMUCH    0x08;        /* Too much entered */
#define NENOUGH    0x04;        /* Not enough entered */
#define WRONG      0x02;        /* Wrong number */
#define NUMBER     0x01;        /* Numeric field */

/* Group 8: Byte 3 */

#define UNAUTH     0x40;        /* Operator unauthorized */

```

```

#define UNAUTHM 0x20;          /* Operator unauthorized */
                                /* minus function */
#define IDEAD 0x10;           /* Invalid dead key combination */
#define WPLACE 0x08;         /* Wrong placed */

    /* Group 8: Byte 4 */

#define PENDING 0x80;        /* Message pending */
#define PARTITION 0x40;     /* Partition wait */
#define SYSTEM 0x20;        /* System wait */
#define MISMATCH 0x10;     /* Hardware mismatch */
#define NCONFIG 0x08;      /* Logical unit not configured */
                                /* at control unit */

    /* Group 8: Byte 5 */

#define AUTOKEY 0x80;       /* Autokey inhibit */
#define INPUT 0x40;        /* Application program has */
                                /* operator input inhibited */

uchar group_8[5];

    /* Group 9: PSS Group 2 */
    /* Group 10: Highlight Group 2 */
    /* Group 11: Color Group 2 */

#define SELECT 0x80;        /* Selected */
#define DISABLE 0x40;      /* Display disabled (Group 9 only) */
    uchar group_9;
    uchar group_10;
    uchar group_11;

    /* Group 12: Communications error reminder */

#define ERROR 0x80;        /* Communications error */
#define MONITOR 0x40;     /* Response time monitor */
    uchar group_12;

    /* Group 13: Printer status */

#define CUSTOM 0x80;       /* Printer code not customized */
#define MALFUNC 0x40;     /* Printer malfunction */
#define PRINTING 0x20;    /* Printer printing */
#define ASSIGN 0x10;     /* Assign printer */
#define WHAT 0x08;       /* What printer */
#define PRINTER 0x04;    /* Printer assignment */
    uchar group_13;

```

```

/* Group 14 & 15: Reserved */

uchar  group_14;
uchar  group_15;

/* Group 16: Autokey play/record status */

#define PLAY      0x80;
#define RECORD   0x40;
uchar  group_16;

/* Group 17: Autokey abort/pause state */

#define OVERFLOW 0x80;          /* Recording overflow */
#define PAUSE    0x40;
uchar  group_17;

/* Group 18: Enlarge state */

#define ENLARGE  0x80;          /* Window is enlarged */
uchar  group_18;
} cpoia;

```

- *length* points to the length of the location pointed to by *data*
- *position* is not applicable to *h_cpoia*

SEE ALSO

h_qsys(3X).

RETURN VALUES

h_cpoia returns two arguments:

- the first argument is the information described by the structure of type *cpoia*, placed at the location pointed to by the *data* calling parameter. The members of the *cpoia* structure describe the following:
 - cp_format:** OIA format byte for the 3270 PC
 - cp_image[80]:** OIA image in host code points, with no extended attribute types.
- the second argument is a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the *position* calling argument, and is also returned as the function value for the *h_cpoia* call.

The return code for *h_cpoia* will have one of these values:

HE_SUCCESS: the target presentation space is unlocked, and the OIA data was successfully returned

HE_INVALID: the target presentation space is not connected

HE_PARM: an error was made when specifying the string length, and the OIA data was not returned

HE_BUSY: the target presentation space is busy, but the OIA data was returned

HE_INHBT: the target presentation space is locked, but the OIA data was returned

HE_SYSERR: a system error was encountered; use *h_qsys* to find out the reason for failure

NAME

`h_cpstr` – copy string to presentation space

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
```

```
int *func;
```

```
char *data;
```

```
int *length;
```

```
int *position;
```

DESCRIPTION

`h_cpstr` copies an ASCII data string directly into the current connected presentation space, at the location specified by the *position* calling parameter. The function continues until the data reaches the named calling string length, or until the data reaches a non-autoskip attribute byte (an unprotected, non-numeric field that does not force the cursor into the next alphanumeric unprotected field). The physical location of the cursor will remain unchanged once the copy is complete.

The string length is assigned by either:

- an explicit length value (if you selected the default value STRLEN using `h_setparms`), or
- an ending string delimiter EOT=*n* (if you selected STREOT using `h_setparms`).

The input string should contain the appropriate extended attribute bytes following each ASCII character if extended attribute bytes were specified using the EAB parameter under `h_setparms`.

Calling arguments:

- *func* points to the symbolic H_CPSTR
- *data* points to the ASCII string to be copied into the presentation space
- *length* points to the length of the location pointed to by *data*
- *position* points to the position in the presentation space where the copy will begin, between 1 and the maximum screen size for the presentation space

SEE ALSO

`h_qcur(3X)`, `h_qsys(3X)`, `h_setparms(3X)`.

NOTES

If you want to place the string data at a specific cursor location, use `h_qcur` first to obtain the presentation space position of the cursor, and then place this value in the presentation space position calling parameter.

The string to be copied can be no larger than 1920 characters (3840 if you are copying extended attribute bytes also) for Model 2, and 3564 characters (7128 if you are copying extended attribute bytes) for Model 5.

RETURN VALUES

h_cpstr returns one argument: a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the *position* calling argument, and is also returned as the function value for the *h_cpstr* call.

The return code for *h_cpstr* will have one of these values:

- HE_SUCCESS: *h_cpstr* was successful
- HE_INHBT: the target presentation space is protected or inhibited, or illegal data was sent to target presentation space (such as a field attribute byte)
- HE_LENGTH: the copy was completed, but the data was truncated
- HE_POS: the specified presentation space position is invalid
- HE_SYSERR: a system error was encountered; call the *h_qsys* function to find out the reason for failure

NAME

`h_cpstrf` – copy string to field

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

The `h_cpstrf` functions transfers a string of characters into a target field in the current connected presentation space. This function can be used with both protected or unprotected fields in a field-formatted presentation space, i.e., a host presentation space.

The calling data string parameter specifies the string that will be transferred; there is no wrapping during the process, and it ends when one of these four conditions is encountered:

- an EOT is reached, if EOT mode was selected using `h_setparms`
- the length of the string is reached
- an end of field is encountered
- the end of the presentation space is reached

Calling arguments:

- `func` points to the symbolic `H_CPSTRF`
- `data` points to the string containing the data to be transferred to a target field within the last connected presentation space
- `length` points to the length of the location pointed to by `data`.
- `position` points to the position of the target copy field

SEE ALSO

`h_chcur(3X)`, `h_qsys(3X)`, `h_sendkey(3X)`, `h_setparms(3X)`.

NOTES

The combined `h_chcur` and `h_sendkey` function calls will return control to your HLLAPI application program sooner than if you use `h_cpstrf`; use this combination if response time is an important factor.

RETURN VALUES

`h_cpstrf` returns one argument: a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the `position` calling argument, and is also returned as the function value for the `h_cpstrf` call.

The return code for `h_cpstrf` will have one of these values:

HE_SUCCESS: `h_cpstrf` was successful

HE_INHBT: the target field was protected or inhibited, or illegal data was sent to the target field (such as a field attribute style)

HE_LENGTH: copy was completed, but data was truncated

HE_POS: the specified presentation space position is invalid

HE_SYSERR: a system error was encountered; call the *h_qsys* function to find out the reason for failure

NAME

h_disc – disconnect presentation space

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

h_disc drops the connection between your HLLAPI application program and the current connected presentation space.

Calling arguments:

- *func* points to the symbolic H_DISC
- *data*, *length*, and *position* are not applicable to *h_disc*

SEE ALSO

h_qsys(3X).

RETURN VALUES

h_disc returns one argument: a return code, defined in the *<xhllapi.h>* header file. The return code is placed at the location pointed to by the *position* calling argument, and is also returned as the function value for the *h_disc* call.

The return code for *h_disc* will have one of these values:

- HE_SUCCESS: *h_disc* was successful
- HE_INVALID: you are not currently connected to any presentation space
- HE_SYSERR: a system error was encountered; call *h_qsess* to find out the reason for failure

NAME

`h_fndlen` – find field length

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
```

```
int *func;
```

```
char *data;
```

```
int *length;
```

```
int *position;
```

DESCRIPTION

The `h_fndlen` functions returns the length of a target field in the current connected presentation space. This function can be used to find both protected or unprotected fields in a field-formatted presentation space, i.e., a host presentation space.

Calling arguments:

- *func* points to the symbolic H_FNDLEN
- *data* points to a two-character calling data string containing:

two blanks, "T " or "t ":	this field
"P " or "p ":	the previous protected or unprotected field
"N " or "n ":	the next protected or unprotected field
"NP" or "np":	the next protected field
"NU" or "nu":	the next unprotected field
"PP" or "pp":	the previous protected field
"PU" or "pu":	the previous unprotected field
- *length* is not applicable to `h_fndlen` (2 is implied)
- *position* points to the position within the presentation space where the `h_fndlen` function will start

SEE ALSO

`h_qsys(3X)`.

NOTES

`h_fndlen` returns the number of characters contained in the returned data string, including all characters from the beginning of the target field up to either the character preceding the next attribute byte, or the end of the presentation space.

RETURN VALUES

`h_fndlen` returns two arguments:

- the first argument is the length of the requested field, not including the attribute byte. This value is placed at the location pointed to by the *length* calling parameter.

- the second argument is a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the *position* calling argument, and is also returned as the function value for the *h_fndlen* call.

The return code for *h_fndlen* will have one of these values:

HE_SUCCESS: *h_fndlen* was successful

HE_INVALID: your programmed operator was not connected to the desired presentation space

HE_PARM: a parameter error was encountered

HE_POS: the specified presentation space position is invalid

HE_SYSERR: a system error was encountered; call the *h_qsys* function to find out the reason for failure

HE_NOFIELD: no such field was found

NAME

`h_fndpos` – find field position

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
```

```
int *func;
```

```
char *data;
```

```
int *length;
```

```
int *position;
```

DESCRIPTION

`h_fndpos` returns the beginning position of a target field in the current connected presentation space. This function can be used to find both protected or unprotected fields in a field-formatted host presentation space.

Calling arguments:

- `func` points to the symbolic H_FNDPOS
- `data` points to a two-character data string containing:

two blanks, "T ", or "t ":	this field
"P " or "p ":	the previous protected or unprotected field
"N " or "n ":	the next protected or unprotected field
"np""NP" or :	the next protected field
"nu""NU" or :	the next unprotected field
"pp""PP" or :	the previous protected field
"pu""PU" or :	the previous unprotected field
- `length` is not applicable to `h_fndpos` (2 is implied)
- `position` points to the position within the field, relative to the origin of the presentation space, at which `h_fndpos` will start

SEE ALSO

`h_qsys(3X)`.

RETURN VALUES

`h_fndpos` returns two arguments:

- the first argument is the the position of the requested field, relative to the origin of the presentation space; the origin of the presentation space is defined to be the first position after the attribute byte. This value is placed in the location pointed to by the `length` calling parameter. This value is placed at the location pointed to by the `length` calling parameter.
- the second argument is a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the `position` calling argument, and is also returned as the function value for the `h_fndpos` call.

The return code for *h_fndpos* will have one of these values:

- HE_SUCCESS: *h_fndpos* was successful
- HE_INVALID: your programmed operator was not connected to a valid presentation space
- HE_PARM: a parameter error was encountered
- HE_POS: the specified presentation space position is invalid
- HE_SYSERR: a system error was encountered; call the *h_qsys* function to find out the reason for failure

NAME

`h_getkey` – get key

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
struct get_key *data;
int *length;
int *position;
```

DESCRIPTION

Your HLLAPI application program can retrieve keystrokes from sessions specified by the `h_startkey` function call and process them by using `h_getkey`. Keystrokes entered by the terminal user arrive asynchronously, and are queued in the keystroke buffer that you provided in your application program with `h_startkey`.

You can use `h_sendkey` to the original keystrokes, and any others that your application program needs to pass to the target presentation space.

The CapsLock key has the same effect as the host shift lock key; i.e., pressing CapsLock will produce the upper case of all keys, not just of the alphabetic keys.

After the terminal user returns control to your HLLAPI application program, repeated calls to the `h_getkey` function will empty out the buffer of previously stored keystrokes.

Calling arguments:

- `func` points to the symbolic `H_GETKEY`
- `data` points to a structure of type `get_key`, defined in the `<xhllapi.h>` header file as shown below:

```
typedef struct {
    char gk_psid; /* Presentation Space ID */
    char gk_option; /* Option code character */
    char gk_buffer[4];
} get_key;
```

The members of this structure must all be specified in the calling function. They describe the following:

<code>gk_psid</code> :	presentation space short name
<code>gk_option</code> :	option code character position
<code>gk_buffer[4]</code> :	4 byte address of the buffer space that will be used internally, allocated with <code>h_stman</code> 's "Get Storage" option.

- `length` and `position` are not applicable to `h_getkey`

SEE ALSO

`h_qsys(3X)`, `h_sendkey(3X)`, `h_setparms(3X)`, `h_startkey(3X)`.

RETURN VALUES

`h_getkey` returns two arguments:

- the first argument is the structure of type `get_key` mentioned previously, describing the following information:
 - `gk_psid`: presentation space short name
 - `gk_option`: an option code: "A" for ASCII returned (the symbolic name is `GK_ASCII`); "M" for keystroke mnemonic (the symbolic name is `GK_MNEM`); or S for special shift (Alt/Ctrl) returned with other data (the symbolic name is `GK_SHIFT`)
 - `gk_buffer[4]`: the 4 bytes of buffer space that will be used internally for enqueueing and dequeuing keystrokes; bytes 3 and 4 contain ASCII plus `X'00'`, or `@` or `ESC = n` character plus keystroke mnemonic; bytes 5 and 6 may be similar to bytes 3 and 4, or may be set to 0

This information is placed at the location pointed to by the *data* calling argument.

- the second argument is a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the *position* calling argument, and is also returned as the function value for the `h_getkey` call.

The return code for `h_getkey` will have one of these values:

- `HE_SUCCESS`: `h_getkey` was successful
- `HE_INVALID`: an invalid presentation space was specified
- `HE_INHBT`: you specified the "AID only" option under the `h_startkey` function call, and non-AID keys are inhibited by this session type when HLLAPI tries to write invalid keys to the presentation space
- `HE_PROC`: no prior `h_startkey` was called for this presentation space
- `HE_SYSERR`: a system error was encountered, call the `h_qsys` function to find out the reason for failure
- `HE_NOKEYS`: the keystrokes requested are not available on the input queue

NAME

`h_invoke` – invoke UNIX System

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
```

```
int *func;
```

```
char *data;
```

```
int *length;
```

```
int *position;
```

DESCRIPTION

The `h_invoke` function call allows your application program (parent application) to invoke a UNIX System program or command(s). The calling data string parameter contains the entire UNIX System command line. Once the process is complete, control is returned to your application program and the resulting data, if any, is placed into the application program portion following the `h_invoke` function call.

Calling arguments:

- `func` points to the symbolic H_INVOKE
- `data` points to the string containing the command line for the UNIX command(s) or program
- `length` points to the length of the location pointed to by `data`
- `position` is not applicable to `h_invoke`

SEE ALSO

`h_qsys(3X)`.

RETURN VALUES

`h_invoke` returns one argument: a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the `position` calling argument, and is also returned as the function value for the `h_invoke` call.

The return code for `h_invoke` will have one of these values:

HE_FNUM:	an invalid function number was specified
HE_NOENT:	file not found
HE_ACCESS:	access denied
HE_MEM:	insufficient memory
HE_ENV:	invalid environment
HE_FORM:	invalid format

NAME

`h_pause` — pause

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

`h_pause` waits for a specified amount of time, and should be used in place of "timing loops" to wait for an event to occur. A host event may end a `h_pause`, if a prior call was made to `h_strthost`.

`h_pause` should not be used for:

- tasks with long durations
- timing out tasks until system response time is better (4:00am, for example), and then begin an event
- applications that require a high-resolution timer; use an alternate timing method, since the interval created by `h_pause` is approximate

When your application program calls `h_pause`, the length of the pause is affected by the FPAUSE and IPAUSE parameters in `h_setparms`. You must use `h_ghost` to obtain information on the type of update (presentation space and/or OIA) and the host, once a host event satisfies a pause, before the next `h_pause`. If your application program uses the IPAUSE option, the pending event will continue to satisfy a `h_pause` until `h_ghost` returns.

Calling arguments:

- `func` points to the symbolic H_PAUSE
- `length` points to the pause duration in half-second increments; a practical maximum value is 7200
- `data` and `position` are not applicable to `h_pause`

SEE ALSO

`h_ghost(3X)`, `h_qsys(3X)`, `h_setparms(3X)`.

RETURN VALUES

`h_pause` returns one argument: a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the `position` calling argument, and is also returned as the function value for the `h_pause` call.

The return code for `h_pause` will have one of these values:

- HE_SUCCESS: the pause duration has expired
- HE_SYSERR: a system error was encountered; use `h_qsys` to find out the reason for failure

HE_UPDATE: a host session presentation space or OIA has been updated; use *h_ghost* for more information

NAME

`h_postint` – post intercept status

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
struct post_intercept *data;
int *length;
int *position;
```

DESCRIPTION

`h_postint` informs the 3270 PC Control Program that a keystroke obtained using `h_getkey` was accepted.

Your HLLAPI application program does not run at the same time that the terminal user enters keystrokes, and therefore, this function cannot reject any of the keystrokes before they go to the host. However, you can issue the `h_postint` function call, and the return code will be set to `HE_SUCCESS` if no error conditions are present.

Calling arguments:

- `func` points to the symbolic `H_POSTINT`.
- `data` points to a structure of type `post_intercept`, defined in the `<xhllapi.h>` header file as follows:

```
typedef struct {
    char pi_psid; /* Presentation Space ID */
    char pi_option; /* Option code */
} post_intercept;
```

The members of `post_intercept` describe the following information:

`pi_psid`: short name of the presentation space
`pi_option`: "A" for accepted keystroke (the symbolic name is `PI_ACCEPT`), or a "R" for rejected keystroke (the symbolic name is `PI_REJECT`)

- `length` and `position` are not applicable to `h_postint`

SEE ALSO

`h_getkey(3X)`, `h_qsys(3X)`.

RETURN VALUES

`h_postint` returns one argument: a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the `position` calling argument, and is also returned as the function value for the `h_postint` call.

The return code for `h_postint` will have one of these values:

`HE_SUCCESS`: `h_postint` was successful

- HE_INVALID: an invalid presentation space was specified, one that was not connected, configured, or that was labeled with an invalid name
- HE_PARM: an invalid session option was specified
- HE_PROC: no prior *h_startkey* was called for this presentation space ID
- HE_SYSERR: a system error was encountered; use *h_qsys* to find out the reason for failure

NAME

`h_qattr` – query field attribute

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

`h_qattr` returns the attribute byte of the field containing the presentation space position in the current connected presentation space, ignoring extended attribute bytes (EABs).

Calling arguments:

- `func` points to the symbolic H_QATTR
- `data` and `length` are not applicable to `h_qattr`
- `position` points to the position in the current connected presentation space (1 through 1920)

SEE ALSO

`h_qsys(3X)`.

NOTES

Attribute byte values must be greater than X'CO'.

RETURN VALUES

`h_qattr` returns two arguments:

- the first argument is the attribute value of the field containing the presentation space position in the current connected presentation space, or 0 if the screen is unformatted. This argument is placed at the location pointed to by the `length` calling parameter.
- the second argument is a returned code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the `position` calling argument, and is also returned as the function value for the `h_qattr` call.

The return code for `h_qattr` will have one of these values:

- HE_SUCCESS: `h_qattr` was successful
- HE_INVALID: your programmed operator was not connected to a host
- HE_POS: the specified presentation space position is invalid
- HE_SYSERR: a system error was encountered; call the `h_qsys` function to find out the reason for failure



NAME

`h_qcur` – query cursor location

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

The `h_qcur` function shows the position of the cursor in the current connected presentation space. The position of the cursor will be returned as follows:

- if you specified OLDRET with `h_setparms`, the cursor position will be placed at the location pointed to by the `position` calling argument
- if you specified NEWRET with `h_setparms`, the cursor position will be placed at the location pointed to by the `length` calling argument

You must be connected to the desired presentation space before you can call `h_qcur`.

Calling arguments:

- `func` points to the symbolic H_QCUR
- `data`, `length`, and `position` are not applicable to `h_qcur`. However, you must pass pointers to these locations as arguments to `h_qcur`.

SEE ALSO

`h_qsys(3X)`, `h_setparms(3X)`.

RETURN VALUES

`h_qcur` returns two arguments:

- the first argument is the cursor position. If you specified OLDRET with `h_setparms`, this argument is placed at the location pointed to by the `position` calling argument. If you specified NEWRET with `h_setparms`, the cursor position is placed at the location pointed to by the `length` calling argument.
- the second argument is a return code, defined in the `<xhllapi.h>` header file. If you specified OLDRET with `h_setparms`, the return code is returned as the function value for the `h_qcur` call. If you specified NEWRET with `h_setparms`, the return code is placed at the location pointed to by the `position` calling argument, and is also returned as the function value for the `h_qcur` call.

The return code for `h_qcur` will have one of these values:

```
HE_SUCCESS:  h_qcur was successful
HE_INVAL:    the desired presentation space was not connected
HE_SYSERR:   a system error was encountered; use h_qsys to find
              out the reason for failure
```


NAME

`h_ghost` – query host update

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
```

```
int *func;
```

```
char *data;
```

```
int *length;
```

```
int *position;
```

DESCRIPTION

`h_ghost` lets the programmed operator determine if the host has updated the presentation space and/or the OIA, with real data sent from the host, since the last time this request was made. The target presentation space must be specified in the data string, but you don't need to be connected to a given host presentation space to check for updates. Your application program must call `h_strthost` before using `h_ghost`.

Calling arguments:

- `func` points to the symbolic H_QHOST.
- `data` points to the target presentation space short name. The presentation space short name is a capital letter from A through Z, and it is the first field for the entry in the LUTABLE environment file that corresponds to the target presentation space.
- `length` and `position` are not applicable to `h_ghost`

SEE ALSO

`h_qsys(3X)`, `h_strthost(3X)`.

RETURN VALUES

`h_ghost` returns one argument: a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the `position` calling argument, and is also returned as the function value for the `h_ghost` call.

The return code for `h_ghost` will have one of these values:

- HE_SUCCESS: no updates have been made since the last call
- HE_INVALID: an invalid presentation space was specified, one that was labeled with an invalid name
- HE_PROC: no prior `h_strthost` function was called for this presentation space ID
- HE_SYSERR: a system error was encountered; use `h_qsys` to find out the reason for failure
- HE_OIA: the OIA was updated
- HE_PRES: the presentation space was updated
- HE_BOTH: both OIA and presentation space were updated

NAME

`h_qsess` – query sessions

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
struct q_sessions_data *data;
int *length;
int *position;
```

DESCRIPTION

`h_qsess` returns the session short name, long name, type, and the presentation space size for as many presentation spaces as the size of the `data` calling argument can accommodate. This size is specified by the `length` calling argument.

Calling arguments:

- `func` points to the symbolic `H_QSESS`.
- `data` points to a structure of type `q_session_data`. This structure is defined in the `<xhllapi.h` header file as follows:

```
/* Query Sessions data string */

typedef struct {
    char qe_psid;                /* Short name */
                                /* of session */
    char qe_lname[LONG_NAME];   /* Long name */
                                /* of session */
    char qe_stype;              /* Session Type */
    short qe_size;              /* PS Size */
} q_sessions_data;
```

- `length` points to the length of the location pointed to by `data`.
- `position` is not applicable to `h_qsess`.

SEE ALSO

`h_qsys(3X)`.

RETURN VALUES

`h_qsess` returns two arguments:

- the first argument points to a data string structure of type `q_session_data` that contains the session short name, long name, type (host) and the presentation space size. This information is placed at the location pointed to by the `data` calling parameter.
- the second argument is a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the `position` calling argument, and is also returned as the function value for the `h_qsess` call.

The return code for *h_qsess* will have one of these values:

- HE_SUCCESS: *h_qsess* was successful
- HE_PARM: an improper string size was specified; the string is too small, and the ability to verify the string size is language dependent (as with other functions)
- HE_SYSERR: a system error was encountered

NAME

`h_qstatus` – query session status

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
struct q_status_data *data;
int *length;
int *position;
```

DESCRIPTION

`h_qstatus` provides your application program with session-specific information. The you need to specify the session short name in the calling `data` parameter, and the requested session information will be returned in the structure pointed to by `data`.

The session information provided is the session long name, the session type ('H', for HOST only), the session characteristics, whether the session has base attributes or extended attributes, whether the session supports programmed symbols, and if your application program is a well-behaved one (through PIF status information).

Calling arguments:

- `func` points to the symbolic `H_QSTATUS`.
- `data` points to a structure of type `q_status_data`, defined in the `<xhllapi.h>` header file as follows:

```
/* Query Session Status data string */

typedef struct {
    char qt_psid;                /* Short name */
    char qt_lname[LONG_NAME];  /* Long name */
    char qt_stype;              /* Session type */
    char qt_schars;            /* Session */
                                /* Characteristics */
    short qt_rows;             /* Rows in presen- */
                                /* tation space */
    short qt_cols;             /* Columns in PS */
    char qt_pifstat[2];        /* PIF Status */
    char qt_reserved;          /* Reserved */
} q_status_data;
```

The members of the `q_status_data` structure describe the following information:

<code>qt_psid:</code>	presentation space short name
<code>qt_lname:</code>	presentation space long name
<code>qt_stype:</code>	session type ('H' only)

qt_schars:	sessions characteristics byte:
	SC_EAB: the session has extended attributes
	SC_PSS: the session supports programmed symbols
qt_rows:	rows in the presentation space
qt_cols:	columns in the presentation space
qt_pifstat:	PIF status
qt_reserved:	reserved

You must specify the **qt_psid** and **qt_lname** members of this structure. The remaining members are returned by the *h_qstatus* function call.

- *length* points to the length of the location pointed to by *data*.
- *position* is not applicable to *h_qstatus*.

SEE ALSO

h_qsys(3X).

RETURN VALUES

h_qstatus returns two arguments:

- the first argument points to the structure of type **q_status_data** mentioned above, and is placed at the location pointed to by the *data* calling parameter.
- the second argument is a return code, defined in the *<xhllapi.h>* header file. The return code is placed at the location pointed to by the *position* calling argument, and is also returned as the function value for the *h_qstatus* call.

The return code for *h_qstatus* will have one of these values:

HE_SUCCESS:	<i>h_qstatus</i> was successful
HE_INVALID:	the requested session was invalid
HE_SYSERR:	a system error was encountered; call the <i>h_qsys</i> function to find out the reason for failure

NAME

`h_qsys` – query system

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
struct q_system_data *data;
int *length;
int *position;
```

DESCRIPTION

Your HLLAPI application program can use `h_qsys` to determine the level of 3270 PC Control Program support, and other system related values. `h_qsys` returns a string with the appropriate system data; the AT&T Tier 4 Support group can use this information to help you determine the problem, after you received an `HE_SYSERR` return code (a system error was encountered).

You should include a return code check in your HLLAPI application program, as a prerequisite for continuing the program. If the return code is `HE_SYSERR` (system error), your application program should call a subroutine that calls `h_qsys`, and extracts extended error code information to help the AT&T Tier 4 Support group determine the cause of the system error.

Calling arguments:

- `func` points to the symbolic `H_QSYS`.
- `data` points to a structure of type `q_system_data`, defined in the `<xhllapi.h>` header file as follows:

```
/* Query System data string */

typedef struct {
    char sy_vernum;           /* HLLAPI Version Number */
    char sy_levnum[2];       /* HLLAPI Level Number */
    char sy_date[6];         /* HLLAPI Date (MMDDYY) */
    char sy_limver;          /* LIM Version */
    char sy_limlev[2];       /* LIM Level */
    char sy_hwbase;          /* Hardware Base */
    char sy_cptype;          /* Control Program Type */
    char sy_cplevel;         /* Control Program Level */
    char sy_resv1;           /* Reserved */
    char sy_resv2[2];        /* Reserved */
    char sy_psid;            /* Session Short Name */
    char sy_exterr1[4];      /* Extended Error Code 1 */
    char sy_exterr2[4];      /* Extended Error Code 2 */
    char sy_resv[8];         /* Reserved */
} q_system_data;
```

The members of `q_system+data` describe the following information:

sy_vern num:	AT&T 3270 Emulator+ HLLAPI version number
sy_lev num[2]:	AT&T 3270 Emulator+ HLLAPI level number
sy_date [6]:	AT&T 3270 Emulator+ HLLAPI date (month, date, and year for service purposes only)
sy_lim ver:	LIM version number
sy_lim lev[2]:	LIM level number
sy_hw base:	hardware base
sy_cp type:	3270 PC Control Program type
sy_cp level:	3270 PC Control Program level
sy_res v1:	reserved
sy_res v2[2]:	reserved
sy_p sid:	presentation space short name
sy_ext err1[4]:	Extended Error Code 1; this is a printable ASCII string representing a hex word giving the HLLAPI component ID and system error number for that function
sy_ext err2[4]:	Extended Error Code 2; this is a printable ASCII string representing a fault symptom code for the last internal system error
sy_res v[8]:	reserved

You do not need to provide any of the above information for *data*.

- *length* and *position* are not applicable to *h_qsys*.

RETURN VALUES

h_sys returns two arguments:

- the first argument points to the structure of type **q_system_data** described above, and is placed at the location pointed by the *data* calling parameter.
- the second argument is a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the *position* calling argument, and is also returned as the function value for the *h_qsysf1* call.

The return code for *h_qsys* will have one of these values:

HE_SUCCESS :	<i>h_qsys</i> was successful; data string was returned
HE_PARM :	improper string size (string too small), and the ability to verify the string size is language dependent
HE_SYSERR :	a system error was encountered

NAME

`h_recv` — receive file

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

`h_recv` requests that a file be sent from a host session to the AT&T 3B computer where HLLAPI is running.

To receive a file from a host session, you must be logged on to TSO or CMS, and have received the ready message on your screen. For TSO, the ready message is "READY", and for CMS it is "R;".

The TIMEOUT parameter under `h_setparms` is used by this function.

Calling arguments for TSO:

- `func` points to the symbolic H_RECV
- `data` points to the file transfer command string. This command string has the following format:

```
file_name h:data_set_name(member_name)/password ASCII CRLF
APPEND
```

file_name is the name of the file containing the received data.

h is the short name of the host presentation space that the transfer will take place on. It need not be present if only one host session is defined in the LUTABLE. If it is present, it must be followed by a colon (:).

data_set_name is the name of the host data set that the file you want to transfer is on. If it is a fully qualified name, enclose it in single quotes.

member_name is the member of a Partitioned Data Set (PDS) you wish to receive. If the member name is present, it must be enclosed in parenthesis and *data_set_name* must be the name of the PDS. If the *member_name* is also part of a fully qualified *data_set_name* both names must be enclosed in single quotes (' ').

password is the password required to access the host data set. If present, it must be preceded by a slash (/).

ASCII CRLF indicate to the host ASCII/EBCDIC translation

will take place during the transfer and also that logical records are separated by the two characters CR and LF. Since UNIX stores files with records that end in NL, CRLF will be automatically translated to NL. These parameters should be present if you are receiving text files from the host; they should appear separated by a blank, and in the order shown. Binary files should not be transferred using this parameter.

APPEND is an optional parameter that specifies that received data is to be added to the end of the local file. If the parameter is not present, a new file will be created or an existing file of the same name will be overwritten.

- *length* points to the length of the location pointed to by *data*
- *position* is not applicable to *h_recv*

Calling arguments for CMS:

The calling arguments for CMS are the same as for TSO, except for the command string pointed to by the *data* argument. This command string has the following format:

file_name *h:fn ft fm* (ASCII CRLF APPEND)

file_name is the name of the file containing the received data.

h is the short name of the host presentation space that the transfer will take place on. It need not be present if only one host session is defined in the LUTABLE. If it is present, it must be followed by a colon (:).

fn ft fm is the file name, file type, and file mode of the receive file on CMS. The file name is required, while the file type and file mode are optional.

ASCII CRLF indicate to the host ASCII EBCDIC translation will take place during the transfer and also that logical records are separated by the two characters CR and LF. Since UNIX stores files with records that end in NL, CRLF will be automatically translated to NL. This parameter should be present if you are receiving text files from the host; binary files should not be transferred using this parameter.

APPEND is an optional parameter that specifies that received data is to be added to the end of the local file. If the parameter is not present, a new file will be created or an existing file of the same name will be overwritten.

SEE ALSO

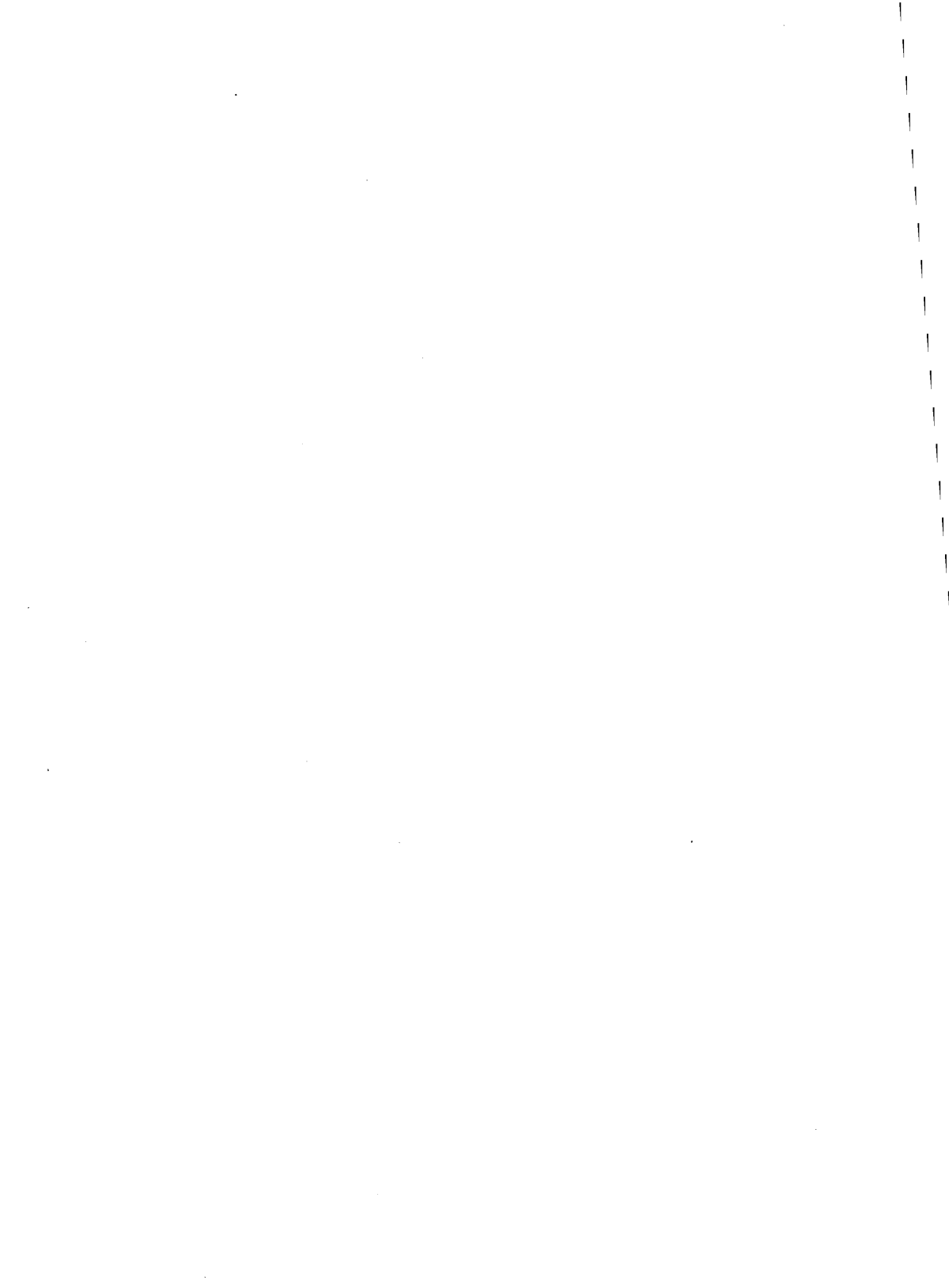
h_qsys(3X), *h_setparms*(3X).

RETURN VALUES

h_recv returns one argument: a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the *position*

calling argument, and is also returned as the function value for the *h_recv* call. The return code for *h_recv* will have one of these values:

HE_PARM:	parameter error - you have specified a data string length that is too long, and the file transfer was unsuccessful
HE_XCOMPL:	file transfer complete
HE_XCOMPLSEG:	file transfer complete with records segmented
HE_SYSERR:	a system error was encountered; call the <i>h_gsys</i> function to find out the reason for failure
HE_FNUM:	invalid function number
HE_NOENT:	file not found
HE_ACCESS:	access denied
HE_MEM:	insufficient memory
HE_ENV:	invalid environment
HE_FORM:	invalid format



NAME

`h_redir` – UNIX System redirect

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

`h_redir` allows the parent application program to do anything that you could do from the shell prompt '\$'. Once this process is complete, control is returned to the parent application.

Calling arguments:

- `func` points to the symbolic H_REDIRE
- `data` points to the string containing the UNIX System command line
- `length` points to the length of the data string parameter
- `position` is not applicable to `h_redir`

SEE ALSO

`h_qsys(3X)`.

RETURN VALUES

`h_redir` returns one argument: a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the `position` calling argument, and is also returned as the function value for the `h_redir` call.

The return code for `h_redir` will have one of these values:

HE_FNUM:	an invalid function number was specified
HE_NOENT:	file not found
HE_ACCESS:	access denied
HE_MEM:	insufficient memory
HE_ENV:	invalid environment
HE_FORM:	invalid format

NAME

h_rel – release

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

h_rel unlocks a presentation space that was reserved using *h_resv*; the target presentation space is the current connected presentation space.

Calling arguments:

- *func* points to the symbolic H_REL
- *data*, *length*, and *position* are not applicable to *h_rel*

SEE ALSO

h_resv(3X), *h_qsys*(3X).

NOTES

A presentation space can be controlled, in a mutually exclusive way, by either the terminal user or the HLLAPI application program. Therefore, *h_rel* has no real functionality in the AT&T 3270 Emulator+ HLLAPI implementation, and it has been included for compatibility with the IBM 3270 HLLAPI product.

RETURN VALUES

h_rel returns one argument: a return code, defined in the *<xhllapi.h>* header file. The return code is placed at the location pointed to by the *position* calling argument, and is also returned as the function value for the *h_rel* call.

The return code for *h_rel* will have one of these values:

- HE_SUCCESS: *h_rel* was successful
- HE_INVALID: you HLLAPI application program is not connected to a valid presentation space
- HE_SYSERR: a system error was encountered; call the *h_qsys* function to find out the reason for failure

NAME

h_reset – reset system

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

h_reset reinitializes the HLLAPI library, and resets the session parameter options to their defaults. This function releases any reserved sessions, and disconnects all connected presentation spaces.

You can use *h_reset* during initialization, or at the end of the program to reset the system to a known initial condition.

Calling arguments:

- *func* points to the symbolic H_RESET
- *data*, *length*, and *position* are not applicable to *h_reset*

SEE ALSO

h_qsys(3X), *h_setparms*(3X), *h_stman*(3X).

NOTES

h_reset does not free up blocks of storage that were allocated with *h_stman*; use the "Free All Storage" option under *h_stman* for this purpose.

RETURN VALUES

hllapi returns one argument: a return code, defined in the *<xhllapi.h>* header file. The return code is placed at the location pointed to by the *position* calling argument, and is also returned as the function value for the *h_reset* call.

The return code for *h_reset* will have one of these values:

HE_SUCCESS: *h_reset* was successful

HE_SYSERR: a system error was encountered; call the *h_qsys* function to find out the reason for failure



NAME

`h_resv` – reserve

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

`h_resv` locks the current connected presentation space to prevent input from the terminal operator. You may need to prevent the user from gaining access to the host session where your HLLAPI application program is sending transactions, until the application program is done.

Calling arguments:

- `func` points to the symbolic H_RESV
- `data`, `length`, and `position` are not applicable to `h_resv`

SEE ALSO

`h_qsys(3X)`.

NOTES

A presentation space can be controlled, in a mutually exclusive way, by either the terminal user or the HLLAPI application program. Therefore, `h_resv` has no real functionality in the AT&T 3270 Emulator+ HLLAPI implementation, and it has been included for compatibility with the IBM 3270 HLLAPI product.

RETURN VALUES

`h_resv` returns one argument: a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the `position` calling argument, and is also returned as the function value for the `h_resv` call.

The return code for `h_resv` will have one of these values:

- | | |
|-------------|--|
| HE_INVALID: | your HLLAPI program is not connected to a valid presentation space |
| HE_INHBT: | <code>h_resv</code> failed because the presentation space was inhibited |
| HE_SYSERR: | a system error was encountered; <code>h_qsys</code> to find out the reason for failure |

NAME

`h_search` – search presentation space

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

The `h_search` function allows your HLLAPI program to search for a particular string within the current connected presentation space. `h_search` normally checks the entire presentation space.

`h_search` scans the presentation space for the specified string, and if you specified OLDRET:

- the return code is set to the beginning location of the string in the presentation space, if the string is found. This location is based on the layout where the upper left corner is location 1 (row 1, column 1), and the lower right corner is:
 - 1920 for Model 2s
 - 3564 for Model 5s
- if the string is not found, the return code is set to 0.

If you specified NEWRET with `h_setparms`:

- the returning length is set to the beginning location of the string in the presentation space, if the string is found. This location follows the same layout as described above for OLDRET
- the returning length is set to 0 if the string is not found

You can use the `h_search` function to determine when a specific 3270 presentation space is available. `h_search` allows you to check for specific prompt messages before continuing, if your programmed operator is expecting a specific prompt or message before sending data. Your program can then call the `h_pause` or `h_ghost` function calls and continue to call `h_search` until it receives a non-zero return code.

The parameters under `h_setparms` that relate directly to search functions are listed below; an asterisk (*) means that this is the default option:

- SRCHALL *: the search function will scan the entire presentation space.
- SRCHFROM: the search function will start from the specified beginning position.
- SRCHFRWD *: the search function will take place in an ascending direction.

SRCHBKWD: the search function will take place in a descending direction. If the first character of the requested string starts within the specified search bounds, the search will be satisfied.

If you are looking for a string that may occur multiple times within the presentation space, you can use *h_setparms* to specify SRCHFROM. Once you specify a search starting position, the function looks for the named string from that position through the end of the presentation space.

Calling arguments:

- *func* points to the symbolic H_SEARCH
- *data* points to the string to be searched
- *length* points to the length of *data*
- *position* points to the address where the search will begin (not used with the SRCHALL option under *h_setparms*)

SEE ALSO

h_qsys(3X), *h_setparms*(3X).

NOTES

You can also use *h_setparms* to specify SRCHBKWD. In this mode, the search function locates the last time the string occurs.

RETURN VALUES

h_search returns two arguments:

- the first argument is the starting position of the searched string, and it is placed at the position pointed to by the *length* calling parameter. If you specified NEWRET with *h_setparms*, then the value for this argument will be:
 - length* = 0: the string was not found
 - length* > 0: the string was found at the presentation space position, shown in *length*
- the second argument is a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the *position* calling argument, and is also returned as the function value for the *h_search* call.

If you specified OLDRET with *h_setparms*, then the return code will be

- = 0: meaning that the string was not found
- > 0: meaning that the string was found at the presentation space position shown in *position*

If you specified NEWRET with *h_setparms*, and *length* > 0, then the return code will be:

HE_SUCCESS: *h_search* was successful

HE_INVALID: the specified presentation space was not connected
HE_PARM: an error was made in specifying parameters
HE_POS: the specified presentation space position is invalid
HE_SYSERR: a system error was encountered; call the *h_qsys*
function to find out the reason for failure
HE_NOFIELD: the search string was not found



NAME

`h_send` – send file

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

`h_send` is used to send a file from the AT&T 3B computer, where HLLAPI is running, to a TSO or CMS host session.

To send a file to the host session, you must be logged on to TSO or CMS, and have received the ready message on your screen. For TSO, the ready message is "READY", and for CMS it is "R;".

The TIMEOUT parameter under `h_setparms` is used by this function.

Calling arguments for TSO:

- `func` points to the symbolic H_SEND
- `data` points to the command line for the file that you want to send to the host. This command line must contain the following information:

```
file_name h:data_set_name(member_name)/password ASCII CRLF
APPEND LRECL(n) BLKSIZE(n) RECFM(x) SPACE(n1,n2) units
```

file_name is the name of the file that contains the data to be transmitted.

h is the short name of the host presentation space that the transfer will take place on. It need not be present if only one host session is defined in the LUTABLE. If it is present, it must be followed by a colon.

data_set_name is the host data set you wish the transferred file to reside. If it is a fully qualified name, enclose it in single quotes.

member_name is the member of a Partitioned Data Set (PDS) you wish to send to. If *member_name* is present, it must be enclosed in parenthesis and *data_set_name* must be the name of the PDS. If the *member_name* is also part of a fully qualified *data_set_name*, both names must be enclosed in single quotes.

password is the password required to access the host data set. If present, it must be preceded by a slash (/).

ASCII CRLF indicate to the host ASCII/EBCDIC translation will take place during the transfer and also that logical records are

separated by the two characters CR and LF. Since UNIX stores files with records that end in NL, NL will be automatically translated to CRLF. These parameters should be present if you are receiving text files from the host; they should appear separated by a blank, and in the order shown. Binary files should not be transferred using this parameter.

APPEND is an optional parameter that specifies that transmitted data is to be added to the end of the host file. If the parameter is not present, a new file will be created or an existing file of the same name will be overwritten.

LRECL(n) if present indicates the record length of the TSO file. *n* may be from 1 to 132, with a default of 80 if this parameter is not present. This parameter may not be included if the member parameter is specified.

BLKSIZE(n) is an optional parameter indicating the block size of the new TSO data set. If the member parameter is present this parameter may not be used. The default for this parameter is the same as the logical record length (LRECL).

RECFM(x) is optional indicating the record format of the new TSO data set. *x* may be **v**, **f** or **u** for variable, fixed or undefined. If the CRLF parameter was specified and this parameter was not, the data set will be created for variable length records otherwise the records are assumed to be fixed length. This parameter should not be included if the member parameter was specified.

SPACE(n1,n2) units is an optional parameter indicating the amount of space to be allocated for the new data set. *n1* indicates the size of the initial allocation and *n2* the amount of all subsequent additions to the data set. *units* may be anyone of **BLOCKS**, **TRACKS** or **CYLINDERS**. If this parameter is omitted, a new data set will only be allocated one block. If the member parameter was used, this parameter should not be present.

- *length* points to the length of the string pointed to by *data*
- *position* is not applicable to *h_send*

Calling arguments for CMS:

The calling arguments for CMS are the same as for TSO, except for the command string pointed to by the *data* argument. This command string has the following format:

file_name *h:fn ft fm* (**ASCII CRLF APPEND LRECL(n) RECFM(x)**)

file_name is the name of the file that contains the data to be transmitted.

h is the short name of the host presentation space that the transfer will

take place on. It need not be present if only one host session is defined in the LUTABLE. If it is present, it must be followed by a colon (:).

fn ft fm is the file name, file type, and file mode of the target file on CMS. The file name and file type is required. The file mode is optional and will default to your A disk.

ASCII CRLF indicate to the host ASCII/EBCDIC translation will take place during the transfer and also that logical records are separated by the two characters CR and LF. Since UNIX stores files with records that end in NL, NL will be automatically translated to CRLF. These parameters should be present if you are receiving text files from the host; they should appear separated by a blank, and in the order shown. Binary files should not be transferred using this parameter.

APPEND is an optional parameter that specifies that transmitted data is to be added to the end of the host file. If the parameter is not present, a new file will be created or an existing file of the same name will be overwritten.

LRECL(*n*) if present indicates the record length of the CMS file. *n* may be from 1 to 132, with a default of 80 if this parameter is not present.

RECFM(*x*) is optional indicating the record format of the new CMS data set. *x* may be *v* or *f* for variable, fixed. If the CRLF parameter was specified and this parameter was not the data set will be created for variable length records otherwise the records are assumed to be fixed length.

SEE ALSO

`h_qsys(3X)`, `h_setparms(3X)`.

RETURN VALUES

`h_send` returns one argument: a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the *position* calling argument, and is also returned as the function value for the `h_send` call.

The return code for `h_send` will have one of these values:

HE_PARM:	parameter error - you have specified a data string length that is too long, and the file transfer was unsuccessful
HE_XCOMPL:	file transfer complete
HE_XCOMPLSEG:	file transfer complete with records segmented
HE_SYSERR:	a system error was encountered; call the <code>h_qsys</code> function to find out the reason for failure
HE_FNUM:	invalid function number
HE_NOENT:	file not found

H_SEND(3X)

(AT&T 3270 Emulator+ HLLAPI)

H_SEND(3X)

HE_ACCESS:	access denied
HE_MEM:	insufficient memory
HE_ENV:	invalid environment
HE_FORM:	invalid format

NAME

`h_sendkey` – send key

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
```

```
int *func;
```

```
char *data;
```

```
int *length;
```

```
int *position;
```

DESCRIPTION

`h_sendkey` sends a string of keystrokes to the current connected presentation space. You must use `h_connect` in your application program before sending keystrokes.

You define the keystrokes that will be sent in the calling data parameter, and they will appear to the target session as though they were entered by the terminal operator. You cannot send keystrokes to a session whose keyboard is locked (e.g., when input is inhibited), and your application program must deal accordingly with those fields that are protected for input, or that are numeric only.

You can send all attention identifier (AID) keys such as Enter, PA1, etc, but keystroke input will no longer be accepted after the first AID character is received. The remainder of the input data string will be ignored.

To send special keystrokes, use the ESC character that you specified with `h_setparms` function call (@ is the default), followed by the keystroke mnemonic, in the calling data string:

Name	Keystroke	Name	Keystroke
ALT_CR	@A@n	PF1	@1
ATTN	@A@Q	PF2	@2
Backtab	@B	PF3	@3
Clear	@C	PF4	@4
Delete	@D	PF5	@5
Down arrow	@D	PF6	@6
DUP	@S@x	PF7	@7
Enter	@E	PF8	@8
Erase EOF	@F	PF9	@9
Erase Input	@A@F	PF10	@a
FM	@S@y	PF11	@b
Home	@0	PF12	@c
Insert	@I	PF13	@d
LDUB	@A@L	PF14	@e
Left arrow	@L	PF15	@f

New Line	@N	PF16	@g
Print	@A@P	PF17	@h
RDUB	@A@R	PF18	@i
Right arrow	@R	PF19	@j
Tab	@T	PF20	@k
Test	@A@C	PF21	@l
Up arrow	@U	PF22	@m
Reset	@R	PF23	@n
PA1	@x	PF24	@o
PA2	@y	PA3	@z

For information regarding the use of the keys listed above, see the *AT&T 3270 Emulator+ User's Guide*.

Calling arguments:

- *func* points to the symbolic H_SENDKEY
- *data* points to the string of keystrokes that you want to send
- *length* points to the length of the string pointed to by *data*
- *position* is not applicable to *h_sendkey*

SEE ALSO

h_connect(3X), *h_qsys*(3X), *h_setparms*(3X).

NOTES

A receiving session left in shift lock (CapsLock) state will cause numeric data to be sent as alphabetic data.

RETURN VALUES

h_sendkey returns one argument: a return code, defined in the *<xhllapi.h>* header file. The return code is placed at the location pointed to by the *position* calling argument, and is also returned as the function value for the *h_sendkey* call.

The return code for *h_sendkey* will have one of these values:

- HE_SUCCESS: the keystrokes were sent, and status is normal.
- HE_INVALID: your HLLAPI program is not connected to a valid session
- HE_PARM: an incorrect parameter was passed to HLLAPI
- HE_BUSY: the 3270 host session was busy; all the keystrokes could not be sent
- HE_INHBT: input to the target session was inhibited or rejected; of the keystrokes could not be sent
- HE_SYSERR: a system error was encountered; call *h_qsys* to find out the reason for failure

NAME

`h_setparms` – set session parameters

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
```

```
int *func;
```

```
char *data;
```

```
int *length;
```

```
int *position;
```

DESCRIPTION

h_setparms lets you change certain default session options in the HLLAPI library. These options are described next in terms of the functions that they affect, and are presented in mutually exclusive groups; an asterisk (*) following the name of a parameter within each group specifies that this is the default option.

The following parameters affect *h_search*, *h_qcur*, and *h_qsess*:

Parameter	Description
NEWRET *	HLLAPI release 3.0 is in use; this release uses the standard return codes
OLDRET	HLLAPI release 1.0 or 2.0 is in use; these versions do not use the standard return codes, but allow the return code to include the result code

These parameters affect all Copy functions:

Parameter	Description
ATTRB	pass back all codes that do not have an ASCII equivalent (except EABs) as their original values
NOATTRB *	convert all unknown values to blanks
EAB	pass the presentation space data with extended attribute bytes; you will get two characters for every one that appears on the screen (remember that you must allocate a buffer twice the size of the presentation space, e.g., 2x1920 for a Model 2 screen)
NOEAB *	pass data only (no EABs)
STRLEN *	an explicit length will be passed for all strings
STREOT	string lengths are not explicitly coded; they end with an EOT (End Of Text) character
EOT=n	allows you to specify the EOT to show the end of a string in STREOT mode. Binary zero is the default. You may use any character except a blank after the equals sign.

The following parameters affect *h_connect*:

Parameter	Description
CONPHYS	do a physical connection; i.e., during an <i>h_connect</i> function call, jump to the requested presentation space, and transfers control to the user at the terminal to enter data using the keyboard
CONLOG *	do a logical connect with the requested presentation space, and do not transfer control or allow the user at the terminal to be aware of this connection unless the DISPLAY option (below) has been specified
DISPLAY	allow the user to view the results of your HLLAPI application program during an <i>h_connect</i> function call (if you specified the CONLOG option, above)
NODISPLAY *	do not allow the user at the terminal to view the results of your HLLAPI application program execution during a <i>h_connect</i> function call (if you specified the CONLOG option, above)

These parameters affect *h_sendkey*:

Parameter	Description
ESC= <i>n</i>	specify the escape character for keystroke mnemonics (@ is the default); any character is valid except a blank
AUTORESET *	the application will attempt to reset all inhibited conditions by prefixing all strings of keys sent using <i>h_sendkey</i> with a keyboard reset key sequence
NORESET	do not AUTORESET

These parameters that affect all Search functions:

Parameter	Description
SRCHALL *	the search will scan the entire presentation space
SRCHFROM	the search will start from a specified beginning position
SRCHFRWD *	the search will be performed in an ascending direction, and a search will be satisfied if the first character of the requested string starts within the bounds specified for the search
SRCHBKWD	the search will be performed in a descending direction, and a search will be satisfied if the first character of the requested string starts within the bounds specified for the search

The parameters related to using TRACE to debug your HLLAPI application program are:

Parameter	Description
TRON	turns TRACE on
TROFF *	turns TRACE off; the trace function may conflict with messages on the screen from languages or applications that manage their own displays

The following parameters affect the *h_wait* and *h_getkey*:

Parameter	Description
TWAIT *	<i>h_wait</i> will return control to your application program after a pending event completes, if there is one; otherwise, <i>h_wait</i> will wait approximately one minute before timing out on XCLOCK or XSYSTEM and returning control to your application program
LWAIT	<i>h_wait</i> will wait indefinitely or until XCLOCK or XSYSTEM clears, i.e., after an event occurs; this option is not recommended since control does not return to your application program if there is no event pending
NWAIT	<i>h_wait</i> checks the status of an event, and returns control immediately to your application program (no wait)

The parameters that affect the *h_pause* function call are:

Parameter	Description
FPAUSE *	full duration pause
IPAUSE	interruptible pause; <i>h_strthost</i> and a host event will satisfy a <i>h_pause</i>

The following parameters affect the *h_send* and *h_recv*:

Parameter	Description
QUIET	keeps SEND and RECEIVE and any other messages sent to the screen from being displayed and HLLAPI will keep track of the message number and discard the message; the UNIX System provides the capability to redirect I/O, and therefore, the use of this parameter may have no effect
NOQUIET *	restores message display; the use of this parameter may have no effect since the UNIX System provides the capability to redirect I/O
TIMEOUT= <i>n</i>	where " <i>n</i> " is a value from the table below. TIMEOUT messages will be displayed every 30 seconds until the operator presses CTRL+BREAK (these messages would not be visible in the QUIET mode); TIMEOUT=0 is the standard for operator usage of SEND and RECEIVE, and TIMEOUT= <i>n</i> causes a one-character specifier from Figure 4-1 to tell the HLLAPI library how many 30 second cycles (how many messages with INDFT010) it should accept before issuing a CTRL+BREAK

Character Specifiers Used with the Timeout Option:

Character	Value (in 30 sec cycles)	Character	Value (in 30 sec cycles)
1	0.5	8	4.0
2	1.0	9	4.5
3	1.5	J	5.0
4	2.0	K	5.5
5	2.5	L	6.0
6	3.0	M	6.5
7	3.5	N	7.0

The parameters that affect the handling of IBM Personal Computer HLLAPI unsupported functions are listed below. These functions are:

- Define Presentation Space
- Switch Presentation Space
- Display Cursor
- Display Presentation Space
- Delete Presentation Space
- Get 3270 Aid Key

Parameter	Description
UNSUP_OK	returns a return code of "0" (operation successful) unconditionally without taking any further actions
UNSUP_NG	returns a return code of "10" (function not supported) unconditionally without doing anything
UNSUP_VAR *	returns either a return code of "0" or "10" depending on the function

Calling arguments:

- *func* points to the symbolic H_SETPARMS
- *data* points to a character string that contains the session parameters previously described, separated by commas or blanks
- *length* points to the length of the data string (no EOT)
- *position* is not applicable

SEE ALSO

h_qsys(3X).

NOTES

The parameter EOT=*n* also affects *h_send*, *h_recv*, *h_invoke*, and *h_redir*.

ESC=*n* also affects *h_getkey*.

RETURN VALUES

h_setparms returns one argument: a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the *position* calling argument, and is also returned as the function value for the *h_setparms* call.

The return code for *h_setparms* will have one of these values:

- HE_SUCCESS: the session parameters have been set
- HE_PARM: the length of the parameter list is invalid
- HE_SYSERR: a system error was encountered; call the *h_qsys* function to find out the reason for failure

NAME

`h_srchfld` – search field

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
```

```
int *func;
```

```
char *data;
```

```
int *length;
```

```
int *position;
```

DESCRIPTION

`h_srchfld` searches the last connected presentation space for the occurrence of the string specified in the `data` calling parameter. This function returns the decimal position of the string, numbered from the beginning of the presentation space, if the target string is found (position 1 is the "row 1, column 1" position). `h_srchfld` can be used to search for either protected or unprotected fields, but only in a field-formatted host presentation space.

The search will not wrap from the bottom of the screen to the top; you can use `h_setparms` to determine whether your searches will search forward or backward.

Calling arguments:

- `func` points to the symbolic H_SRCHFLD
- `data` points to the target search string
- `length` points to the length of the location pointed to by `data` if STRLEN (under `h_setparms` is on), or to a blank if STREOT is on
- `position` points to the position within the presentation space where the search will begin

SEE ALSO

`h_qsys(3X)`, `h_setparms(3X)`.

NOTES

If you pass a `position` that does not coincide with the beginning of a field, the search will start at the beginning of the field that contains the presentation space position passed.

RETURN VALUES

`h_srchfld` returns two arguments:

- the first argument is placed at the location pointed to by the `length` calling parameter, and has one of two values:
 - = 0: means that the string was not found
 - > 0: means that the string was found at the presentation space position shown
- The second argument is a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the `position` calling argument, and is also returned as the function value for the `h_srchfld` call.

The return code for *h_srchfld* will have one of these values:

- HE_SUCCESS: *h_srchfld* was successful
- HE_INVALID: your programmed operator was not connected to a valid presentation space
- HE_POS: the specified presentation space position is invalid
- HE_SYSERR: a system error was encountered; call the *h_qsys* function to find out the reason for failure
- HE_NOFIELD: the search string was not found, or the screen was unformatted

NAME

`h_startkey` – start keystroke intercept

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
struct start_keystroke *data;
int *length;
int *position;
```

DESCRIPTION

`h_startkey` allows your application program to record keystrokes sent to a session by a terminal operator. The recorded keystrokes may be:

- received through the `h_getkey` function call and sent to the same or another session with `h_sendkey`
- used to trigger some other process

After you issue a `h_startkey` function call, you must give explicit control of the session to the terminal user. You can accomplish this by issuing an `h_connect` function call, and specifying the CONPHYS option with `h_setparms`.

The buffer area that `length` points to must be large enough to hold all keystrokes entered by the terminal user, until control is returned to your HLLAPI application program. If the terminal user enters more keystrokes than this buffer will hold, the most recent ones will be truncated.

The `h_stman` option "Get Storage" returns a 4-byte address in the calling data string parameter that can be concatenated with the first data string bytes for `h_getkey`.

Calling arguments:

- `func` points to the symbolic H_STARTKEY.
- `data` points to a structure of type `start_keystroke`, defined in the `<xhllapi.h>` header file as follows:

```
typedef struct {
    char sk_psid; /* Presentation Space ID */
    char sk_option; /* Option Code */
    char *sk_buffer;
} start_keystroke;
```

The members of the `start_keystroke` structure describe the following:

- sk_psid:** presentation space short name
- sk_option:** an option code character: SK_AID, for AID keystrokes only, or SK_ALL for all keystrokes
- sk_buffer:** address of a buffer space that will be used internally for enqueueing and dequeuing of host events; allocated with `h_stman`.

- *length* points to the length of the location pointed to by *data*.
- *position* is not applicable to *h_startkey*.

SEE ALSO

h_connect(3X), *h_getkey*(3X), *h_postint*(3X), *h_qsys*(3X), *h_sendkey*(3X), *h_setparms*(3X), *h_stman*(3X).

RETURN VALUES

h_startkey returns one argument: a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the *position* calling argument, and is also returned as the function value for the *h_startkey* call.

The return code for *h_startkey* will have one of these values:

- HE_SUCCESS:** *h_startkey* was successful
- HE_PARM:** an invalid option was specified
- HE_BUSY:** the execution of the function was inhibited because the target presentation space was busy
- HE_SYSERR:** a system error was encountered; call the *h_qsys* function to find out the reason for failure

NAME

`h_stman` – storage manager

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, id, subfunc)
int *func;
char *data;
int *id;
int *subfunc;
```

DESCRIPTION

`h_stman` preallocates blocks of storage for use by certain HLLAPI functions. The `h_stman` subfunctions are:

- **Get Storage** *Get Storage* allocates a portion of the storage block for use by a particular HLLAPI function. The size of the allocated storage block will be equal to the requested size, plus 5 bytes; you can allocate a maximum of 128 blocks of storage.
- **Free Storage** *Free Storage* frees a block of storage that was previously allocated with *Get Storage*.
- **Query Free Storage** *Query Free Storage* searches for, and returns, the size of the largest available free storage block. The size of this block must be larger than 5 bytes to be used with *Get Storage*.
- **Free All Storage** *Free All Storage* frees all allocated storage blocks, and regroups them into a single storage pool.

Calling arguments:

- *func* points to the symbolic H_STMAN
- *data* is not applicable, although it is preallocated to 4 bytes
- *id* points to the size of the requested storage area; the maximum storage area is determined by the STSIZE environment variable, defined in the LIM .profile
- *subfunc* points to an `h_stman` subfunction:

01	Get Storage
02	Free Storage
03	Query Free Storage
04	Free All Storage

SEE ALSO

`h_qsys(3X)`.

NOTES

A given storage block will remain the size of the original allocation; later requests for storage will waste the excess storage in the previously allocated block. For example, if a block of 100 bytes was allocated and freed, and later

you requested to reallocate the block with 80 bytes, the excess 20 bytes are wasted.

RETURN VALUES

Get Storage

h_stman returns three arguments for *Get Storage*:

- the first argument is the storage address, expressed as two binary words (4 bytes) in offset segment order. This address is placed at the location pointed to by the *data* calling parameter.
- the second argument is the storage block id (0 - 127), and it is placed at the location pointed to by the *id* calling parameter.
- the third argument is a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the *position* calling argument, and is also returned as the function value for the *h_stman* call.

The return code will have one of these values:

HE_SUCCESS: the requested storage was allocated

HE_INVALID: you requested more storage than is available

HE_SYSERR: a system error was encountered; call the *h_qsys* function to find out the reason for failure

Free Storage

h_stman returns one argument for *Free Storage*: a return code, placed at the same locations as the return code for *Get Storage*. The possible return codes are:

HE_SUCCESS: storage block has been freed

HE_PARM: the storage block ID was invalid

HE_SYSERR: a system error was encountered; call the *h_qsys* function to find out the reason for failure

Query Free Storage

h_stman returns two arguments for *Query Free Storage*:

- the first argument is the size of the largest available block of storage, and it is placed at the location pointed to by the *length* calling parameter.
- the second argument is a return code, placed at the same locations as the return codes for the previously described subfunctions. The possible return codes are:

HE_SUCCESS: *Query Free Storage* was successful

HE_SYSERR: a system error was encountered; call the *h_qsys* function to find out the reason for failure

Free All Storage

h_stman returns two arguments for *Free All Storage*:

- the first argument is the size of the largest available block of storage, and it is placed at the location pointed to by the *length* calling parameter.

- the second argument is a return code, placed at the same location as the return codes for the previously described subfunctions. The possible return codes for *Free All Storage* are:

HE_SUCCESS: *Free All Storage* was successful

HE_SYSERR: a system error was encountered; use *h_qsys* to find out the reason for failure



NAME

`h_stophost` – stop host notification

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

`h_stophost` disables the ability of `h_ghost` to determine if the specified host OIA and/or the presentation space have been updated. `h_stophost` also stops host events from the specified host from affecting `h_pause`.

Calling arguments:

- `func` points to the symbolic H_STOPHOST
- `data` points to the presentation space short name
- `length` and `position` are not applicable

SEE ALSO

`h_pause(3X)`, `h_ghost(3X)`, `h_qsys(3X)`, `h_setparms(3X)`.

RETURN VALUES

`h_stophost` returns one argument: a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the `position` calling argument, and is also returned as the function value for the `h_stophost` call.

The return code for `h_stophost` will have one of these values:

- HE_SUCCESS: `h_stophost` was successful
- HE_INVALID: an invalid presentation space was specified, one labeled with an invalid name
- HE_PROC: no previous `h_stophost` was issued
- HE_SYSERR: a system error was encountered; use `h_qsys` to find out the reason for failure



NAME

`h_stopkey` – stop keystroke intercept

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

`h_stopkey` ends your application program's ability to intercept keystrokes with `h_startkey`.

Calling arguments:

- `func` points to the symbolic `H_STOPKEY`
- `data` points to the short name of the target presentation space
- `length` and `position` are not applicable to `h_stopkey`

SEE ALSO

`h_qsys(3X)`.

RETURN VALUES

`h_stopkey` returns one argument: a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the `position` calling argument, and is also returned as the function value for the `h_stopkey` call.

The return code for `h_stopkey` will have one of these values: The possible return codes are:

- `HE_SUCCESS:` `h_stopkey` was successful
- `HE_INVALID:` an invalid presentation space was specified
- `HE_PROC:` no prior `h_stopkey` was called for this presentation space
- `HE_SYSERR:` a system error was encountered; use `h_qsys` to find out the reason for failure

NAME

`h_strthost` – start host notification

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
struct host_notify *data;
int *length;
int *position;
```

DESCRIPTION

`h_strthost` determines if the specified presentation space and/or the specified OIA have been updated, and enables the specified presentation space ID to end a pause started with `h_pause`. After using `h_strthost`, your application program can use `h_qhost` to determine the specific host event that has occurred.

If you allocate buffer storage within your HLLAPI application program without using `h_stman`, you must use `h_stophost` or `h_reset` functions before you exit HLLAPI. This will avoid storage overlap failure of later programs.

The "Get Storage" option in `h_stman` returns a 4-byte address in the data string parameter that can be concatenated with the first data string bytes for this function.

Calling arguments:

- `func` points to the symbolic H_STRTHOST.
- `data` points to a structure of type `host_notify`, defined in the `<xhllapi.h` header as follows:

```
typedef struct {
    char hn_psid; /* Presentation Space ID */
    char hn_type; /* Update type */
    char *hn_buffer;
} host_notify;
```

You must provide values for all members of the structure of type `host_notify`, which describe the following:

- hn_psid:** a specific presentation space short name (PSID).
- hn_type:** one of these possibilities:
 - the symbolic HN_PRES, asking for notification of presentation space update only
 - the symbolic HN_OIA, asking for notification of OIA Update only
 - the symbolic HN_BOTH, asking for notification of both presentation space and OIA updates, or
- *hn_buffer:** the address of the buffer space that will be used internally for enqueueing and dequeuing host events (specified with `h_stman` function call, using the "Get Storage" subfunction).

- *length* points to the length of the host event buffer; this buffer should be long enough to store the desired keystrokes.
- *position* is not applicable to *h_strthost*.

SEE ALSO

h_pause(3X), *h_qhost*(3X), *h_qsys*(3X), *h_reset* (3X), *h_stman*(3X), *h_stophost*(3X).

RETURN VALUES

h_strthost returns one argument: a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the *position* calling argument, and is also returned as the function value for the *h_strthost* call.

The return code for *h_strthost* will have one of these values:

- HE_SUCCESS:** *h_strthost* was successful
- HE_INVALID:** an invalid presentation space was specified
- HE_PARM:** an error was made when specifying parameters
- HE_SYSERR:** a system error was encountered; use *h_qsys* to find out the reason for failure

NAME

`h_wait` – wait

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

h_wait checks the status of the current connected presentation space. If the 3270 session is waiting on a host response, suggested by XCLOCK or XSYSTEM, *h_wait* will cause your application program to wait approximately one minute to see if the condition clears (if the default option TWAIT is set using *h_setparms*).

h_wait gives host requests, like those made by *h_sendkey*, time to complete before continuing. This is analogous to an operator waiting for the keyboard to unlock before entering data.

Calling arguments:

- *func* points to the symbolic H_WAIT
- *data*, *length*, and *position* are not applicable to *h_wait*

SEE ALSO

h_qsys(3X), *h_setparms*(3X).

NOTES

h_wait uses the WAIT parameters in the *h_setparms* function call (TWAIT, LWAIT and NWAIT).

RETURN VALUES

h_wait returns one argument: a return code, defined in the *<xhllapi.h>* header file. The return code is placed at the location pointed to by the *position* calling argument, and is also returned as the function value for the *h_wait* call.

The return code for *h_wait* will have one of these values:

- HE_SUCCESS: the keyboard is unlocked and ready for input
- HE_INVAL: your application program is not connected to a valid session
- HE_WSCTRL: your application program is connected to WS Ctrl
- HE_BUSY: timeout while still in XCLOCK or XSYSTEM
- HE_INHBT: the keyboard is locked
- HE_SYSERR: a system error was encountered; call the *h_qsys* function to find out the reason for failure

NAME

h_wrchar – copy characters to presentation space

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
char *data;
int *length;
int *position;
```

DESCRIPTION

h_wrchar copies one or more binary characters directly into the current connected presentation space, starting at the location specified by the PS position calling parameter. This function is intended for binary image copying into the presentation space. The copy continues until the field length count is reached; no ASCII to EBCDIC conversion will take place.

h_wrchar operates in STRLEN mode, even if you did not select STRLEN mode with *h_setparms*. This is because *h_wrchar* copies binary characters and therefore, no EOT escape character can be specified. The input data must contain the appropriate extended attribute byte following each regular byte if extended attribute bytes were specified using the EAB parameter under *h_setparms*.

Calling arguments:

- *func* points to the symbolic H_WRCHAR
- *data* points to the binary bytes to be copied into the presentation space
- *length* points to the length of the location pointed to by *data*, and must be greater than 0
- *position* points to the position within the presentation space where the copy will begin. This value must be between:
 - 1 and 1920 for Model 2s (3840 if you are copying extended attribute bytes)
 - 1 and 3564 for Model 5s (7128 if you are copying extended attribute bytes)

SEE ALSO

h_qsys(3X), *h_setparms*(3X).

NOTES

Use the *h_wrchar* carefully, since it can copy any bit combinations, and no checking is done by the HLLAPI library. You are responsible for making sure that the byte(s) are correct for your purposes; use other functions if validation is desired.

RETURN VALUES

h_wrchar returns one argument: a return code, defined in the *<xhllapi.h>* header file. The return code is placed at the location pointed to by the *position*

calling argument, and is also returned as the function value for the *h_wrchar* call.

The return code for *h_wrchar* will have one of these values:

- HE_SUCCESS: *h_wrchar* was successful
- HE_INVALID: the HLLAPI program was not connected to a valid presentation space
- HE_INHBT: the target presentation space is protected or inhibited, or illegal data was sent
- HE_LENGTH: the copy was completed, but the source data was truncated
- HE_POS: the specified presentation space position is invalid
- HE_SYSERR: a system error was encountered; call the *h_qsys* function to find out the reason for failure

NAME

`h_wsctrl` – work station control

SYNOPSIS

```
#include <xhllapi.h>
```

```
int hllapi(func, data, length, position)
int *func;
struct h_wsctrl_struct *data;
int *length;
int *position;
```

DESCRIPTION

`h_wsctrl` provides the programmed operator with block copy and some window management capabilities. HLLAPI treats `h_wsctrl` as a session type, even though it is really a "pseudo-session." As a pseudo-session, `h_wsctrl` can call functions against other sessions (e.g., copy blocks of text from other sessions), even though it has no presentation space associated with it.

A "window" is a view of a host session, which in turn is an external manifestation of a presentation space; recall that a presentation space is a region in storage, and therefore, it is not visible to users. In the AT&T 3270 Emulator+ HLLAPI implementation, a window occupies the entire screen display, and you can view only one window at any given time. You must select foreground and background colors outside HLLAPI, according to your terminal; you cannot alter these features with `h_wsctrl`.

A "screen profile" defines the way that presentation spaces will be viewed on the screen, and the value of the screen profile will always be "0".

`h_wsctrl` does not allow the use of blanks for a presentation space ID, as other functions do. Therefore, where a presentation space ID is required, you must provide the short name character for that presentation space.

Calling arguments:

- `func` points to the symbolic H_WSCTRL
- `data` points to a structure of type `h_wsctrl_struct` that defines the desired `h_wsctrl` request. For all `h_wsctrl_struct` structures (see the "Structures for `h_wsctrl`" sub-section, below), you need to provide the "command code" structure member, and some of the other members depending on the desired request. The remaining members of the structure will be returned by `h_wsctrl`.
- `length` points to the length of the data string, using the `sizeof` operator on the structure in the data string
- `position` is not applicable to `h_wsctrl`

Structures for `h_wsctrl`

You control requests to `h_wsctrl` by specifying a pointer to a structure for each desired request; these structures have been declared in the `<xhllapi.h>` header file. You can only send one request to `h_wsctrl` at a time.

The AT&T 3270 Emulator+ HLLAPI implementation supports the following `h_wsctrl` requests:

Command Code	<i>h_wsctrl</i> Function Name
CA	Change Active Window
CB	Copy Block
RD	Redraw
QA	Query Active Window
QC	Query Background Color
QE	Query Enlarge State
QW	Query Window
QN	Query Window Names

The *h_wsctrl* requests listed below are not supported by the AT&T 3270 Emulator+ HLLAPI implementation:

Command Code	<i>h_wsctrl</i> Function Name
AW	Add Window
CE	Change Enlarged State
CC	Change Screen Background Color
CW	Change Window
CS	Clear Screen
DW	Delete Window

If you make any of these requests when you invoke *h_wsctrl*, the return code will be set to "HE_SUCCESS" or "HE_FUNCT," depending on the option that you selected with *h_setparms* (see the "RETURN VALUES" section, below). Thus, if you specified

- UNSUP_OK: the return code will be set to "HE_SUCCESS"
- UNSUP_NG: the return code will be set to "HE_FUNCT"
- UNSUP_VAR: the return code will be set to "HE_SUCCESS"

The portion of the `<xhllapi.h>` file that declares structures for these command codes, is:

```
#define  CMDSIZ  2
#define  NAMES   20
struct ca_data_type
{
    char    ws_cmd[CMDSIZ];
    char    ws_equ;
    char    ca_scrn;
    char    ca_psid;
} ;    /* data from CHANGE ACTIVE */

struct cb_data_type
{
    char    ws_cmd[CMDSIZ];
    char    ws_equ;
    char    cb_psid;
    short   cb_srcbeg;
    short   cb_srcend;
    char    cb_tpsid;
    short   cb_tbeg;
} ;    /* data from COPY BLOCK */

struct rd_data_type
{
    char    ws_cmd[CMDSIZ];
    char    ws_equ;
    char    rd_scrn;
} ;    /* data from REDRAW SCREEN */

struct qa_data_type
{
    char    ws_cmd[CMDSIZ];
    char    ws_equ;
    char    qa_scrn;
    char    qa_psid;
} ;    /* data from QUERY ACTIVE */

struct qc_data_type
{
    char    ws_cmd[CMDSIZ];
    char    ws_equ;
    char    qc_scrn;
    char    qc_fill;
    char    qc_color;
} ;    /* data from QUERY COLOR */
```

```

struct qe_data_type
{
    char    ws_cmd[CMDSIZ];
    char    ws_equ;
    char    qe_fill[2];
    char    qe_enlarge;
} ;    /* data from QUERY ENLARGE */

struct qw_data_type
{
    char    ws_cmd[CMDSIZ];
    char    ws_equ;
    char    qw_scrn;
    char    qw_psid;
    short   qw_rows; /* number of rows in window */
    short   qw_cols; /* number of cols. in window */
    short   qw_rpos; /* row position, upper left corner */
    short   qw_cpos; /* col position, upper left corner */
    short   qw_wcolor; /* window color code */
    short   qw_bcolor; /* border color code */
    short   qw_flags; /* control flags */
    short   qw_psrow; /* row on PS of upper left corner */
    short   qw_pscol; /* col on PS of upper left corner */
} ;    /* data from QUERY WINDOW */

struct qn_data_type
{
    char    ws_cmd[CMDSIZ];
    char    ws_equ;
    char    qn_scrn;
    char    qn_UNUSED;
    char    qn_names[NAMES];
} ;    /* data from QUERY NAMES */

/* constants for Query Window function in WSCTRL */
#define    FOREGROUND    WHITE
#define    BACKGROUND    BLACK
#define    WCOLOR        (FOREGROUND << 3 | BACKGROUND)

#define    RESERVED      00000
#define    HIDDEN        1
#define    BASECOLOR     1
#define    WINDCOLOR     1
#define    FLAGS         (HIDDEN << 7 |
                          RESERVED << 2 | BASECOLOR << 1 |
                          WINDCOLOR)

```



```

/* Color codes */
#defineBLACK                0
#defineBLUE                 1
#defineRED                  2
#definePINK                 3
#defineGREEN                4
#defineTURQUOISE           5
#defineYELLOW               6
#defineWHITE                7

/* Row and Column identifiers */
#defineROW_ZERO             0
#defineCOL_ZERO             0

/* Screen Profile Identifiers */
#define SCREEN_ZERO        '0'

/* Enlarge State Identifiers */
#defineENLARGED             1
#defineNOT_ENLARGED        0

```

The supported AT&T 3270 Emulator+ HLLAPI *h_wsctrl* calling and returning *data* strings are explained next.

Change Active Window, CA

Changes the active screen profile and presentation space, displays the presentation space specified in *ca_psid* (below), and routes keyboard inputs as keystrokes to this presentation space.

The structure for this request in the `<xhllapi.h>` file is `ca_data_type`; the members of this structure describe the following:

<code>ws_cmd[CMDSIZ]:</code>	command code, CA for Change Active Window
<code>ws_equ:</code>	an equal sign, (=)
<code>ca_scrn:</code>	the screen profile, whose value must be "0"
<code>ca_psid:</code>	the presentation space that HLLAPI is starting

You must supply all members of the `ca_data_type` structure, when making this request.

Copy Block, CB

Copies the contents of a block defined by a set of coordinates in the source presentation space, to a block defined by a set of coordinates in a target presentation space - the extended attribute bytes are also copied.

The structure for this request is `cb_data_type`; the members for this structure describe the following:

<code>ws_cmd[CMDSIZ]:</code>	the command code, CB for Copy Block
<code>ws_equ:</code>	an equal sign, (=)

cb_psid:	short name for the source presentation space
cb_srcbeg:	the starting copy offset in the source presentation space; position 1 = row 1, column 1
cb_srcend:	the ending copy offset; position 1 = row 1, column 1
cb_tpsid:	short name for the target presentation space
cb_tbeg:	the starting copy offset in the target presentation space; again, position 1 = row 1, column 1

You must supply all members of the **cb_data_type** structure, when making this request.

Redraw Screens, RD

This request redraws the specified screen profile. The AT&T 3270 Emulator+HLLAPI implementation only supports enlarged states, and therefore, the full presentation space of the active window will be redrawn.

The structure for this request is **rd_data_type**, and its members describe the following:

ws_cmd[CMDSIZ]:	command code, RD for Redraw Screens
ws_equ:	an equal sign, (=)
rd_scrn:	active screen profile redrawn; must be "0"

You must supply all members of the **rd_data_type** structure, when making this request.

Query Active Window, QA

Returns the short name of the active window in the screen profile.

The structure for this request is **qa_data_type**, and its members describe the following:

ws_cmd[CMDSIZ]:	command code, QA for Query Active Window
ws_equ:	an equal sign
qa_scrn:	active screen profile; must be "0"
qa_psid:	presentation space short name

When making this request, you must supply the **ws_cmd[CMDSIZ]**, **ws_equ**, and **qa_scrn** members of the **qa_data_type** structure. **qa_psid** will be returned by *h_wsctrl* upon execution of the Query Active Window request.

Query Background Color, QC

Returns the background color of the screen profile. Use Query Window (QW) for the background color of a specific window.

The structure for this request is **qc_data_type**; the members of this structure describe the following:

ws_cmd[CMDSIZ]:	command code, QC for Query Background Color
------------------------	---

ws_equ:	an equal sign
qc_scrn:	the screen profile; must be "0"
qc_fill:	unused filler
qc_code:	one of the following color codes: 0: Black 1: Blue 2: Red 3: Pink 4: Green 5: Turquoise 6: Yellow 7: White

When making this request, you must supply the **ws_cmd[CMDSIZ]**, **ws_equ**, **qc_scrn** and **qc_fill** members of the **qc_data_type** structure. **qc_code** will be returned by *h_wsctrl* upon execution of the **Query Background Color** request.

Query Enlarge State, QE

Returns a flag containing the state of the "enlarge" toggle key.

The structure for this request is **qe_data_type**, and its members describe the following:

ws_cmd[CMDSIZ]:	command code, QE for Query Enlarge State
ws_equ:	an equal sign
qe_fill[2]:	unused filler
qe_code:	binary enlarge state code, 0: not enlarged 1: enlarged

When making this request, you must supply the **ws_cmd[CMDSIZ]**, **ws_equ**, and **qe_fill** members of the **qe_data_type** structure. **qe_code** will be returned by *h_wsctrl* upon execution of the **Query Enlarge State** request.

Query Window, QW

Returns the following information about a window on the screen profile:

- how many rows and columns are in the window
- row and column position of the upper left corner of the window on the screen
- window and border colors
- setting of control flags
- row and column position of the upper left corner of the window on the presentation space

The structure for this request is **qw_data_type**; the members of this structure describe the following:

ws_cmd[CMDSIZ]:	the command code, QW for Query Window
ws_equ:	an equal sign
qw_scrn:	screen profile where the queried window is located, and is always "0"
qw_psid:	presentation space short name for the queried window
qw_rows:	rows in the queried window
qw_cols:	columns in the queried window
qw_rpos:	screen row position of the upper left window corner, and is always "0"
qw_cpos:	screen column position of the upper left window corner, and is always "0"
qw_wcolor:	window color code
qw_bcolor:	border color code
qw_flags:	control flags,
qw_psrow:	row on the presentation space of the upper left corner of the window; the value is always "0"
qw_pscol:	column of the presentation space of the upper left corner of the window; the value if always "0"

When making this request, you must supply the **ws_cmd[CMDSIZ]**, **ws_equ**, **qw_scrn** and **qw_psid** members of the **qw_data_type** structure. The remaining members will be returned by *h_wsctrl* upon execution of the **Query Window** request.

Query Window Names, **QN**

Returns the name of the presentation space displayed in a screen profile.

The structure for this request is **qn_data_type**, and its members describe the following:

ws_cmd[CMDSIZ]:	command code, QN for Query Window Names
ws_equ:	an equal sign
qn_scrn:	screen profile of the queried window names, and must be "0"
qn_psid:	presentation space short name for the queried window names
qn_names[NAMES]:	array of window short names

When making this request, you must supply the **ws_cmd[CMDSIZ]**, **ws_equ**, and **qn_scrn** members of the **qn_data_type** structure. The remaining members will be returned by *h_wsctrl* upon execution of the **Query Window Names** request.

SEE ALSO

`h_qsys(3X)`, `h_setparms(3X)`.

NOTES

The Query Background Color (QC) request will always return a "0" in the "C" field for black background color of the base screen; you cannot change the color of the screen, and it is not necessarily "black." "Black" is an arbitrary choice, and it has no relation to the real color of the screen. This request does no useful work in the AT&T 3B environment, and has been included for compatibility with the IBM 3270 Personal Computer HLLAPI specifications.

The Query Enlarge State (QE) request always returns a "1" in the `qe_code` member field, for "enlarged state"; it can never return a "0" since the "unenlarged state" is not supported.

The Redraw Screens (RD) has no effect in the AT&T 3270 Emulator+ HLLAPI environment; it is present for compatibility with the IBM HLLAPI specifications.

RETURN VALUES

`h_wsctrl` returns one argument: a return code, defined in the `<xhllapi.h>` header file. The return code is placed at the location pointed to by the *position* calling argument, and is also returned as the function value for the `h_wsctrl` call.

The return code for `h_wsctrl` will have one of these values:

- HE_SUCCESS: the `h_wsctrl` operation was successful
- HE_INVALID: an invalid screen ID was specified, i.e., a screen other than "0" was specified
- HE_PARM: an error was made in specifying the string length or parameters
- HE_WSCTRL: `h_wsctrl` is owned by another session
- HE_INHBT: the target area for a copy block operation is inhibited or protected
- HE_LENGTH: `h_wsctrl` is an invalid specification
- HE_SYSERR: a system error was encountered; call `h_qsess` to find out the reason for failure
- HE_FUNCT: this function is not available for the AT&T 3270 Emulator+ HLLAPI Release 3.0

NAME

`xlu2clos` – power off the logical unit

SYNOPSIS

```
#include <xlu2io.h>
```

```
int xlu2clos(lu_chan)
unsigned char *lu_chan;
```

DESCRIPTION

`xlu2clos` performs a power-off sequence of the logical unit designated by `*lu_chan`, the value returned by an `xlu2open(3X)` call. Unread data are discarded.

`xlu2clos` will fail if any of the following are true, with `errapi` set as shown:

[E_ARG1]	Invalid <code>*lu_chan</code> parameter.
[E_CNTX]	<code>*lu_chan</code> does not exist.
[E_INIT]	API initialization not performed [<code>xlu2init(3X)</code> not called].
[E_BUSY]	Busy logical unit.
[E_CTRLIO]	Communication with controller is terminated abnormally; session powered off (see below).
[E_TTY]	Error in controlling terminal device (see below).
[400+]	Program check (see Appendix B).
[500+]	Communication check (see Appendix C).

If the call fails with an `E_CTRLIO` or `E_TTY` error, the error may have occurred on another session. In this case, the returned value of `*lu_chan` may be set to that session and will be different than the input value. This does not indicate that the call has failed for the input session. The results of this call can be determined by use of an `xlu2info(2)` call for the input `*lu_chan`.

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and `errapi` is set to indicate the error.

NAME

`xlu2ctl` – logical unit control functions

SYNOPSIS

```
#include <xlu2io.h>
```

```
int xlu2ctl(lu_chan, request, arg, retcod, secs)
unsigned char *lu_chan;
long request, arg, *retcod;
long secs;
```

DESCRIPTION

`xlu2ctl` performs a specified control function on a logical unit. On input, `*lu_chan` designates an active session-logical unit (LU), returned by an `xlu2open(3X)` call. The returned value of `*lu_chan` may be different than the input value (see LUEVENT, below). `request` and `arg` define the control function to be performed on the LU.

`*retcod` is a returned value. For an LUEVENT KY_CTRL control function (see LUEVENT, below), `*retcod` is set to the application request key entered by the user. For all other values of `request` on LUD_3270 sessions (see LUDMOD, below), `*retcod` is set to the session status. Bitwise ANDing `*retcod` with the following macros, defined in `xapi.h`, returns true if the corresponding activity has been detected on the `lu_chan` LUD_3270 session:

LUR_NOTH (<code>lu_chan</code>)	no activity on session
LUR_PRES (<code>lu_chan</code>)	presentation space update
LUR_STAT (<code>lu_chan</code>)	status update
LUR_PRNT (<code>lu_chan</code>)	host print request
LUR_BELL (<code>lu_chan</code>)	host ring bell request

`secs` is the maximum number of seconds `xlu2ctl` will wait for the requested function to complete. If the call does not complete before `secs` number of seconds have elapsed, the call will fail with `errapi` set to `E_INTR`, and with `*retcod` and `*lu_chan` updated. If the call completes before `secs` seconds have elapsed, `xlu2ctl` will return normally with the appropriate return code. A `secs` value of 0 indicates no timing should be performed.

The control functions for the values of `request` are as follows:

LUAMOD

For an LU in LUD_3270 mode (see LUDMOD, below) and for a formatted display, sets the screen buffer in the access mode, specified by `arg`:

- 0 access all fields (default)
- 1 access protected fields only
- 2 access unprotected fields only

For LUD_RAW or LUD_TRAW mode (see LUDMOD, below) or an unformatted screen (an unformatted LU-LU session screen or an SSCP-LU session), this request is ignored.

LUDTIM

Sets the number of seconds, specified by *arg*, that each API call display is maintained on the screen during LUV_TRC mode (see LUVMOD, below). The default value is zero. If *arg* specifies a negative value, it is treated as zero.

LUDMOD

Changes the data transfer mode for the LU, as set by *xlu2open(3X)*, to the mode specified by *arg*:

- LUD_RAW** Data is transferred between the host and the user application in raw form, unaltered and unprocessed, using any (or no specific) type of data stream.
- LUD_TRAW** Data is transferred between the host and the user application in transparent raw form, unaltered and unprocessed, using any (or no specific) type of data stream.
- LUD_3270** Data is transferred between the host and the user application through the 3278/9 screen buffer with data stream processing including validation and translation.

If the LU is busy [Intervention Required (IVR) mode; see, e.g., KY_BUSY under LUEVENT], the data transfer mode is not altered and an E_BUSY error is returned. If the LU is in the process of sending data to or receiving data from the controller (i.e., the LU's keyboard is locked), the data transfer mode is not altered and an OPI_FUNCT error is returned.

If a session's data transfer mode is changed so that a different type of data stream will be used between the emulator and host, the user application program should send notification of the change to the host. In this case, the host application should be in receive state.

LUVMOD

Sets the screen buffer visibility mode for the LU, as specified by *arg*:

- LUV_NON** Screen update is disabled.
- LUV_DSP** Screen update is enabled.
- LUV_TRC** Screen update is enabled and a visual trace of API calls is provided in the Operator Information Area, to aid in debugging (see Appendix D).
See LUDTIM, above.
- LUV_NTD** Screen update goes to **stdout**, but **stdin** does not have to be a terminal device.

LUEVENT

Causes the event specified in *arg* to be reported to the physical unit. For this value of *request*, *xlu2ctl* will block until the controller acknowledges and completes the processing of the event. The LU's screen buffer may also be transmitted. Control is returned to the

application program as described below.

For an LU in LUD_3270 mode (see LUDMOD, above), the valid values for *arg* are:

KY_ATTn	3278/9 ATTN Key (SNA only)
KY_BAKTAB	move cursor left to the beginning of the previous field
KY_BUSY	enter IVR mode
KY_CLEAR	CLEAR key
KY_CTRL	passes control of the terminal emulation process interface from application program to interactive user, until the user ends control with an [ESC] FD.
KY_DOWN_A	move cursor down one line
KY_E_EOF	3278/9 Erase to End of Field key
KY_E_INPUT	3278/9 Erase Input key
KY_ENTER	ENTER key
KY_HOME	move cursor to home position
KY_LEFT_A	move cursor left one position
KY_NMSG	await next message from the controller/host (no need for keyboard to unlock)
KY_PAn	3278/9 PROGRAM ACCESS keys 1, 2, 3
KY_PEND	awaits the next host event. This request is particularly useful when the session resulting from an <i>xlu2open</i> call is INACTIVE and screen update is not forthcoming.
KY_PFn	3278/9 PROGRAM FUNCTION keys 1, 2, 3, ...24
KY_RESET	3278/9 RESET key
KY_RIGHT_A	move cursor right one position
KY_SYS_REQ	3278/9 SYS_REQ key (SNA only)
KY_TAB	move cursor right to the beginning of the next field
KY_TST_REQ	3278/9 TEST_REQ key (BSC only)
KY_UNBUSY	resume communication with host after KY_BUSY request
KY_UP_A	move cursor up one line
KY_WAIT	awaits the next host screen update

For an LU in LUD_RAW or LUD_TRAW mode (see LUDMOD, above), only the following values of *arg* are valid:

KY_ATTN	3278/9 ATTN key (SNA only)
KY_BUSY	entering IVR mode
KY_NMSG	await next message from the controller/host (no need for keyboard to unlock)
KY_PEND	awaits the next host event. This request is particularly useful when the session resulting from an <i>xlu2open</i> call is INACTIVE and screen update is not forthcoming.
KY_SYS_REQ	3278/9 SYS_REQ key (SNA only)
KY_TST_REQ	3278/9 TEST_REQ key (BSC only)
KY_UNBUSY	resume communication with host after KY_BUSY request

For all modes, if the LU is in the process of sending data to or receiving data from the controller (i.e., the LU's keyboard is locked), the event (excluding KY_WAIT, KY_PEND and KY_CTRL) is rejected with an OPI_FUNCT error.

The calling program is blocked until one of the following conditions occurs:

- For a cursor movement request, following cursor movement.
- For a request on any LU that results in communication with the host or the controller, until keyboard unlock.
- For KY_WAIT on any LU, until the host responds with a screen update and keyboard unlock. KY_WAIT blocks until the host updates any LU's screen buffer and the keyboard is unlocked.
- For KY_PEND on any SNA LU, when a host or controller event is received, resulting in valid session ownership, with keyboard unlock.

On an LU which is connected to an SNA controller, KY_PEND blocks until the host or the controller places an LU in a valid session with the keyboard unlocked. On a LU which is connected to a BSC controller, KY_PEND blocks the application program until the keyboard is unlocked.

- For KY_PEND and KY_NMSG, when a segment arrives on a LUD_RAW or LUD_TRAW channel
- For KY_CTRL, when the interactive user strikes KY_CTRL, KY_PREVS, KY_NEXTS, KY_IDENT, KY_SHELL, or KY_U1 through KY_U10.

In this case, the application request key entered by the user is returned in **retcod*.

For LUEVENT, the returned value of **lu_chan* may be different than the input value, corresponding to another session on which a previously initiated event has completed.

LUEVIMED

Causes the event specified in *arg* to be reported to the physical unit. For this value of *request*, *xlu2ctl* will block until the controller acknowledges the event, but does not wait for the completion of the event. The LU's screen buffer may also be transmitted. Control is returned to the application program as described below.

For an LU in LUD_3270 mode (see LUDMOD, above), the valid values for *arg* are:

KY_ATTN	3278/9 ATTN key (SNA only)
KY_BAKTAB	move cursor left to previous field
KY_BUSY	entering IVR mode
KY_CLEAR	CLEAR key
KY_DOWN_A	move cursor down one line
KY_ENTER	ENTER key
KY_E_EOF	3278/9 Erase to End of Field key
KY_E_INPUT	3278/9 Erase Input Key
KY_HOME	move cursor to home position
KY_LEFT_A	move cursor left one position
KY_NMSG	await next message from the controller/host (no need for keyboard to unlock)
KY_PAn	3278/9 PROGRAM ACCESS keys 1, 2, 3
KY_PFn	3278/9 PROGRAM FUNCTION keys 1, 2, ...,24
KY_RESET	3278/9 RESET key
KY_RIGHT_A	move cursor right one position
KY_SYS_REQ	3278/9 SYS_REQ key (SNA only)
KY_TAB	move cursor right to next field
KY_TST_REQ	3278/9 TEST_REQ key (BSC only)
KY_UNBUSY	resume communication with host after KY_BUSY
KY_UP_A	move cursor up one line

If the LU is in the process of sending data to or receiving data from the controller (i.e., the LU's keyboard is locked), the event is rejected with an OPI_FUNCT error.

For KY_NMSG, the returned value of **lu_chan* may be different than the input value, corresponding to another session on which a message has been received from the controller/host. If there is no message for any session, the call will return with **lu_chan* set to *sess_max* [see *xlu2info(3X)*].

For an LU in LUD_RAW or LUD_TRAW mode (see LUDMOD, above), only the following values of *arg* are valid:

KY_ATT_N	3278/9 ATTN key (SNA only)
KY_BUSY	entering IVR mode
KY_NMSG	await next message from the controller/host (no need for keyboard to unlock)
KY_SYS_REQ	3278/9 SYS_REQ key (SNA only)
KY_TST_REQ	3278/9 TEST_REQ (BSC only)
KY_UNBUSY	resume communication with host after KY_BUSY

For LUEVIMED, the calling program is blocked until one of the following occurs:

- The cursor is repositioned on the LUD_3270 LU.
- All relevant data for the LU are transferred to the controller.

Note: Although LUEVENT and LUEVIMED appear to operate in a similar manner, they return to the caller under different conditions. LUEVIMED merely waits for the controller to acknowledge the event, without waiting for completion of processing. LUEVENT waits for the controller to acknowledge and also complete processing of an event.

xlu2ctl will fail if any of the following are true, with *errapi* set as shown:

[E_ARG1]	Invalid <i>*lu_chan</i> parameter.
[E_ARG2]	Invalid value of <i>request</i> .
[E_ARG3]	Invalid <i>arg</i> .
[E_ARG4]	<i>retcod</i> is the null pointer.
[E_INIT]	API initialization not performed [<i>xlu2init(3X)</i> not called].
[E_CNTX]	<i>*lu_chan</i> does not exist.
[E_BUSY]	Busy logical unit.
[E_XLIB]	Necessary library function(s) excluded at link time.
[OPI_WHAT]	Invalid operation.
[OPI_FUNCT]	Invalid operation.
[E_DMODOE]	LU is in LUD_RAW or LUD_TRAW mode, or could not enter requested data transfer mode.
[E_BNDREJ]	Host mismatch on buffer size; bind rejected.
[E_DOVFLO]	Raw outbound data has overflowed internal queue.
[E_DCOD]	LUD_TRAW is unavailable on ASCII line.
[E_INTR]	<i>secs</i> is non-zero and the function did not complete in <i>secs</i> seconds or this call was blocking and an <i>xlu2intr(3X)</i> was issued (see below).
[E_CTRLIO]	Communication with controller is terminated abnormally; session powered off (see below).

- [E_TTY] Error in controlling terminal device (in LUV_DSP mode; in LUV_TRC mode; in LUV_NON/LUV_NTD mode on LUEVENT with KY_CTRL or on LUDTIM). See below.
- [400+] Program check (see Appendix B).
- [500+] Communication check (see Appendix C).

If the call fails with an E_CTRLIO or E_TTY error, the error may have occurred on another session. In this case, the returned value of **lu_chan* may be set to that session and will be different than the input value. Also, if the call fails with E_INTR, this does not indicate that the call has failed. The control function may succeed at a later time. The results of the call can be determined by use of an *xlu2info(2)* call for the input **lu_chan*.

NOTES

Before selecting LUD_RAW and LUD_TRAW (under LUDMOD), you must turn the display off with LUV_NON (under LUVMOD). Otherwise, you will obtain an ERRAPI return value of E_DMODE.

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errapi* is set to indicate the error.



NAME

`xlu2func` – perform special functions on the logical unit

SYNOPSIS

```
#include <xlu2io.h>
```

```
int xlu2func(lu_chan, func, name, secs)
unsigned char *lu_chan;
long func;
unsigned char *name;
long secs;
```

DESCRIPTION

`xlu2func` performs various special functions on the logical unit specified by `*lu_chan`, the value returned by an `xlu2open(3X)` call. The functions are specified by the value of `func`:

```
LUPRND    print capability disabled
LUPRNF    print to a file
LUPRNC    print request command
LURSET    clear inhibit condition
```

`name` is the function object, according to `func`:

```
LUPRND    name may be NULL
LUPRNF    file path name
LUPRNC    command string
LURSET    RESET-INHIBIT-condition mode
```

For print functions (LUPRND, LUPRNF and LUPRNC), `xlu2func` fails with `errapi` set to `E_DMODE` if the `*lu_chan` session is not in `LUD_3270` mode.

LURSET controls the resetting or clearing of inhibit conditions before returning from subsequent API system calls. For LURSET, `name` may be:

```
LUI_RSYS  Reset SYSTEM inhibit conditions (default)
LUI_SYS   SYSTEM inhibit conditions not automatically reset
```

`secs` is the maximum number of seconds `xlu2func` will wait for the requested function to complete. If the call does not complete before `secs` number of seconds have elapsed, the call will fail with `errapi` set to `E_INTR`. If the call completes before `secs` seconds have elapsed, `xlu2func` will return successfully. A `secs` value of 0 indicates no timing should be performed.

`xlu2func` will fail if any of the following are true, with `errapi` set as shown:

```
[E_ARG1]   Invalid lu_chan parameter.
[E_ARG2]   Invalid func parameter.
[E_PATH]   Invalid name parameter for func LUPRNC or LUPRNF.
[E_ICOND]  Invalid name parameter for func LURSET.
```

- [E_INIT] API initialization not performed [*xlu2init*(3X) not called].
- [E_CNTX] **lu_chan* does not exist.
- [E_DMODE] *func* is LUPRND, LUPRNF or LUPRNC and *lu_chan* is in LUD_RAW or LUD_TRAW mode.
- [E_DOVFLO] Raw outbound data has overflowed internal queue.
- [E_INTR] *secs* is non-zero and the function did not complete in *secs* seconds or this call was blocking and an *xlu2intr*(3X) was issued (see below).
- [E_CTRLIO] Communication with controller terminated abnormally; session powered off (see below).
- [E_TTY] Error in controlling terminal device (LUV_DSP and LUV_TRC modes only, see *xlu2open*). Also see below.

If the call fails with an E_CTRLIO or E_TTY error, the error may have occurred on another session. In this case, the returned value of **lu_chan* may be set to that session and will be different than the input value. Also, if the call fails with E_INTR, this does not indicate that the call has failed. The function may succeed at a later time. The results of the call can be determined by use of an *xlu2info*(2) call for the input **lu_chan*.

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errapi* is set to indicate the error.

NAME

`xlu2gets` – get a string from the LUD_3270 logical unit's screen buffer

SYNOPSIS

```
#include <xlu2io.h>
```

```
int xlu2gets(lu_chan, s, n)
unsigned char *lu_chan,
unsigned char *s;
long *n;
```

DESCRIPTION

`xlu2gets` retrieves the data from the screen buffer field of the LUD_3270 mode logical unit specified by `*lu_chan`, the value returned by an `xlu2open(3X)` call. The field is determined by the buffer location to which the cursor is currently pointing. The contents of the field are converted to a null terminated ASCII string and placed into the user buffer pointed to by `s`.

On call, `*n` specifies the maximum number of characters the user buffer can hold excluding the terminating null. On return, `*n` is set to the number of characters actually placed into the user buffer. For a formatted display, a protected or unprotected field can be retrieved and the maximum returned value of `*n` is the number of characters permitted in the field. For an unformatted display, this is the requested length. If the cursor is positioned to an attribute byte in a formatted display, the call will fail with `errapi` set to `E_ACCS`.

For a formatted screen buffer, the cursor is repositioned to the beginning of the next protected or unprotected field, depending on the value of the access mode [LUAMOD, see `xlu2ctl(3X)`] for this LU (consecutive attributes are skipped). For an unformatted screen buffer, the cursor is repositioned to the first location following the end of the transferred string.

If a screen wrap occurs during string transfer, the field transfer is continued until the field or the requested number of characters is exhausted. The cursor is not positioned to the next field and an `E_OFFB` warning code is returned in `errapi`.

Data character translation of the data stream is performed according to the character echo string definitions specified in the customization file, `sc_ifil`, in the `xlu2init(3X)` call. Null characters in the screen buffer field are not translated. They are transferred as nulls to the user buffer, so that the return string may contain embedded nulls.

`xlu2gets` will fail if any of the following are true, with `errapi` set as shown:

[E_ARG1]	Invalid <code>lu_chan</code> parameter.
[E_ARG2]	<code>s</code> is the null pointer.
[E_ARG3]	<code>n</code> is the null pointer or the value of <code>*n</code> is less than 1.
[E_INIT]	API initialization not performed [<code>xlu2init(3X)</code> not called].
[E_CNTX]	<code>*lu_chan</code> does not exist.
[E_DMODE]	<code>*lu_chan</code> is in LUD_RAW or LUD_TRAW mode.

[E_BUSY]	Busy logical unit.
[OPI_WHAT]	Invalid operation.
[OPI_FUNCT]	Invalid operation.
[E_ACCS]	Cursor positioned to an attribute byte
[E_CTRLIO]	Communication with controller is terminated abnormally; session powered off. See below.
[E_TTY]	Error in controlling terminal device (LUV_DSP and LUV_TRC modes only, see <i>xlu2open</i>). See below.
[400+]	Program check (see Appendix B).
[500+]	Communication check (see Appendix C).

If the call fails with an E_CTRLIO or E_TTY error, the error may have occurred on another session. In this case, the returned value of **lu_chan* may be set to that session and will be different than the input value. This does not indicate that the call has failed for the input session. The results of this call can be determined by use of an *xlu2info(2)* call for the input **lu_chan*.

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value is returned. A return value of -1 indicates a failure and *errapi* is set to indicate the error. A return value of -2 indicates a warning condition and *errapi* is set as follows:

E_OFFB	Cursor is off the screen buffer boundary.
--------	---

NAME

xlu2info – obtain 3278/9 status line and cursor position information

SYNOPSIS

```
include <xlu2io.h>
```

```
int xlu2info(ibuf)
XLU2IBUF *ibuf;
```

DESCRIPTION

xlu2info returns information regarding the session-logical units that have been established by this process, including the contents of the 3278/9 status line and cursor position.

The information is returned in an XLU2IBUF structure pointed to by *ibuf*, defined in *xapi.h*, which contains the following elements:

```
unsigned char act_dt;          /* the bit in positions "3210" indicates
                             activity on the corresponding lu_chan
                             session-logical unit */
unsigned char lu_chan        /* index to ch_dat structure */
unsigned char model;         /* 3278/9 model number: 2,5 */

struct                       /* ch_dat structure */
{
    unsigned char lu_opn;    /* lu connection is open indicator */
    unsigned char lu_act;    /* session activity; this can be examined
                             in the same manner as the retool argument of the
                             xlu2ctl function. NOTE: Only LUD 3270 session
                             (see ludmod) activity is updated */
    long luamod;            /* access all/only protected/only unprot flds */
    long luvmod;            /* screen visibility mode on/off with opia in
                             TE/API mode */
    long luidtim;           /* API function display time on opia in sec */
    long ludmod;            /* data transfer mode:
                             LUD_RAW, LUD_TRAW or LUD_3270 */
    long lusmod;            /* requested security mode: DF/HI SECUR */
    long lusedfl;           /* default security mode: HL/NM_SECUR */
    unsigned char luirst;   /* Inhibit condition reset
                             (for values, see xapi.h) */
    unsigned char lubusy;    /* =1, if lu is in IVR mode; otherwise, =0 */
    unsigned char *pres_spc; /* presentation space address */
    unsigned char *extn_buf; /* extended attribute buffer address */
    short *attr_bma;        /* attribute bit map address */
    int attr_bms;           /* attribute bit map size (short units) */
    short c_line;           /* current line number */
    short c_col;            /* current column number */
    short c_pos;            /* current cursor position */
    short c_beg;            /* beginning of current field */
    short c_end;            /* end of current field */
    unsigned char c_att;     /* current field attribute */
    unsigned char c_fm;      /* current screen buffer:
                             unformatted (0)/formatted (1) */
    long err_cond;          /* API-detected error */
    unsigned char keyb_lck;  /* keyboard state: WAIT or INHIBIT */
    unsigned char wait_lck; /* keyboard state: WAIT (1)/clear (0) */
    unsigned char inhb_lck; /* keyboard state: INHIBIT (2)/clear (0) */
    short err_400;          /* 400 series error messages */
};
```

```

short err_500;          /* 500 series error message */
short opia;            /* terminal status (for values,
                        see inhibit condition section in xapi.h) */
unsigned char bscmod;  /* 1=bsc, 0=sna */
unsigned char line_chr; /* data code: ASCII or EBCDIC */
short max_siz;        /* maximum xlu2writ segment size */
unsigned char sna_own; /* if sna, owner (for values,
                        see session ownership section in xapi.h) */
unsigned char lu_prt;  /* lu port number */
unsigned char pu_nam[PATHSIZ]; /* controller path name */
unsigned char prn_f1[PATHSIZ]; /* printer path name */
long prn_md;          /* printer assignment mode. 0=none, 1=file,
                        2=command */
} ch_dat[sess_max];

```

lu_chan, the index to *ch_dat*, is the value returned by *xlu2open*. *sess_max* is the maximum number of logical units (currently 4) that can be open at one time by one process, defined in *xapi.h*.

The cause of an error return from an API system call is indicated in *errapi*. If the error is detected during function call parameter validation, only *errapi* is set. If the returned error is due to a logical unit INHIBIT condition, *errapi* and the *opia* field are set. If the error is due to a 400 or 500 series error, *errapi* and the *err_400* or the *err_500* field are set. Finally, if the error is due to any of the other error conditions listed for the call, *errapi* and the *err_cond* field are set.

DIAGNOSTICS

Upon completion, a value of 0 is returned. There are no defined error returns.

NAME

`xlu2init` – initialize terminal function library

SYNOPSIS

```
#include <xlu2io.h>
```

```
int xlu2init (sc_ifil, ky_ifil, rt_mfil, t_modl, ds_vers)
unsigned char *sc_ifil, *ky_ifil, *rt_mfil, t_modl, ds_vers;
```

DESCRIPTION

`xlu2init` performs local API initialization functions for the calling process including reading the runtime files and assigning the terminal model number. `sc_ifil` is the pathname of the screen control customization object file, output of the `scinit` utility, described in the *AT&T 3270 Emulator+ System Administrator's Guide*. `ky_ifil` is the pathname of the keyboard mapping customization object file, output of the `kyinit` utility, described in the *Administrator's Guide*. `rt_mfil` is the pathname of the runtime message object file, described in the *Administrator's Guide*. `t_modl` is the 3278/9 terminal model number (2 or 5).

`ds_vers` determines whether extended field attributes will be processed. If `ds_vers` is set to `LUD_EXT`, extended field attributes will be displayed if screen visibility is enabled [see `LUVMOD` in `xlu2ctl(3X)`]. If `ds_vers` is not set to `LUD_EXT` (i.e., set to `LUD_RAW`, `LUD_TRAW` or `LUD_3270`), extended field attributes orders and commands will be ignored.

`xlu2init` is performed only once in an API program, and must precede any other API call.

`xlu2init` will fail if any of the following are true, with `errapi` set as shown:

[E_ARG1]	<code>sc_ifil</code> is the null pointer.
[E_ARG2]	<code>ky_ifil</code> is the null pointer.
[E_ARG4]	Invalid parameter.
[E_SCTL]	Error in opening/reading <code>sc_ifil</code>
[E_KCTL]	Error in opening/reading <code>ky_ifil</code>
[E_RTMG]	<code>rt_mfil</code> is invalid.
[E_MODL]	Physical terminal (as described by <code>sc_ifil</code>) can not support model number.
[E_INIT]	<code>xlu2init</code> previously called for this API process.
[E_XLIB]	Necessary library function(s) excluded at link time.

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and `errapi` is set to indicate the error.



NAME

`xlu2intr` – interrupt API call

SYNOPSIS

```
#include <xlu2io.h>
```

```
int xlu2intr() ;
```

DESCRIPTION

This call is intended to be used from an API application signal catching routine. `xlu2intr` will cause to abort certain API function calls previously issued by this process which are currently in progress and are also currently blocked from completing. The calls capable of blocking are `xlu2ctl(3X)`, `xlu2func(3X)`, `xlu2open(3X)` and `xlu2writ(3X)`. When the signal catcher completes and the interrupted process resumes, any of these calls in the described state will immediately terminate processing and return a failure with *errapi* set to `E_INTR`.

When a signal is caught during execution of an API application, the signal catching routine can not call any other API functions since the signal may have occurred during execution of an API function and API does not allow concurrent calls from the same process.

Signals do not automatically cause API function calls to be interrupted. `xlu2intr` allows the catching routine to notify the mainline process of the occurrence of the signal, such as a timer, by a procedure such as setting a flag. When the mainline application recognizes the flag, it can handle the event which has been signaled and make other API calls as required.

`xlu2intr` will fail if the following is true with *errapi* set as shown:

[E_INIT] API initialization not performed

DIAGNOSTICS

Upon successful completion, one of the following values is returned:

- 0 No API call in progress
- 1 API function in progress
- 2 Terminal Emulation in progress

Otherwise, a value of -1 is returned and *errapi* is set to indicate the error.



NAME

`xlu2open` – power on the logical unit

SYNOPSIS

```
#include <xlu2io.h>
```

```
int xlu2open (lu_chan, pu_name, lu_port, ludmod, luvmod, lusmod, secs)
unsigned char *lu_chan, *pu_name, *lu_port;
long ludmod, luvmod, *lusmod, secs;
```

DESCRIPTION

`xlu2open` requests the services of a port within the specified BSC or SNA 3274 controller and performs a power-on sequence of a logical unit with respect to the host. `pu_name` points to a character string containing the name of the 3274 controller process communication path.

On call, `lu_port` points to a character string containing the 3274 controller port number. `lu_port` may also point to a string containing multiple port numbers separated by a comma or hyphen, representing several (comma) or a range of acceptable ports. A null `lu_port` pointer indicates that the user will accept any available 3270 controller port. The controller port number will be returned in the `lu_prt` field of the `XLU2IBUF` structure for this session [see `xlu2info(3X)`]. Values contained in `*lu_port` can be the following, according to the type of controller specified in `*pu_name`.

Standard	3270 SNA	0 - 31
	3270 BSC	0 - 31
Enhanced	3270 SNA	1 - 254

These values also depend on the total number of logical units for which the controller is configured, as described in the *AT&T 3270 Emulator+ System Administrator's Guide*.

`ludmod` indicates the mode of data transfer between the host and the API user application program:

LUD_RAW	Data is transferred between the host and the user application in raw form, unaltered and unprocessed, using any (or no specific) type of data stream.
LUD_TRAW	Data is transferred between the host and the user application in transparent raw form, unaltered and unprocessed, using any (or no specific) type of data stream.
LUD_3270	Data is transferred between the host and the user application through the 3278/9 screen buffer with data stream processing including validation and translation.

LUD_TRAW is ignored if `*pu_name` is not a BSC controller. If the controller is BSC and LUD_TRAW is selected, but the character set has been configured as ASCII (see "BSC Configuration" in the *Administrator's Guide*), `xlu2open` will fail with `errapi` set to `E_XPRNT`.

`luvmod` indicates the screen visibility mode for the logical unit:

LUV_NON	Screen update is disabled.
LUV_DSP	Screen update is enabled.
LUV_TRC	Screen update is enabled and a visual trace of API calls is provided in the Operator Information Area, to aid in debugging (see Appendix D). Also see LUDTIM in <i>xlu2ctl(3X)</i> .
LUV_NTD	Screen update goes to stdout , but stdin does not have to be a terminal device.

lusmod is valid only if **pu name* is a SNA controller and will be ignored for BSC. On call, **lusmod* indicates how the security mode for the logical unit should be set:

DF_SECUR	Default to what is specified in the cluster controller customization file (see below and "SNA Configuration" in the <i>Administrator's Guide</i>).
HI_SECUR	The security mode of the logical unit is set to high security.

On successful return, *lusmod* specifies the default controller configuration setting (see the *Administrator's Guide*):

NM_SECUR	An <i>xlu2clos(3X)</i> call may not deactivate the session ownership established by the host.
HI_SECUR	An <i>xlu2clos</i> call is ensured to deactivate the session ownership established by the host.

The calling program is blocked until power-on is completed or *secs* seconds have elapsed. Power-on completion occurs when the logical unit has valid session ownership (SNA only), with keyboard unlock, the power-on is rejected, or E_CTRLIO error is detected on this or any other session.

secs is the maximum number of seconds *xlu2open* will wait for this function to complete. If the call does not complete before *secs* number of seconds have elapsed, the call will fail with *errapi* set to E_INTR. If the call completes before *secs* seconds have elapsed, *xlu2open* will return successfully. A *secs* value of 0 indicates no timing should be performed.

xlu2open will fail if any of the following are true, with *errapi* set as shown:

[E_PATH]	<i>pu_name</i> is the null pointer.
[E_ARG3]	Invalid <i>lu_port</i> parameter.
[E_ARG4]	Invalid <i>ludmod</i> parameter.
[E_ARG5]	Invalid <i>luvmod</i> parameter.
[E_SECUR]	Invalid <i>lusmod</i> parameter.
[E_INIT]	API initialization not performed [<i>xlu2init(3X)</i> not called].
[E_SESLIM]	Number of sessions exceeds limit.
[E_XLIB]	Necessary library function(s) excluded at link time.

[E_RSRC]	Controller has insufficient resources, i.e., there are no ports left.
[E_BNDREJ]	Host mismatch, bind rejected.
[E_TRMBSY]	SNA/BSC terminal is busy.
[E_HOSTID]	Illegal SNA/BSC host identification.
[E_CNTID]	Illegal controller identification.
[E_LINID]	Illegal line identification.
[E_ILSIZE]	Bind and device size mismatch; bind rejected.
[E_XPRNT]	Request for transparent line characteristic (LUD_TRAW) rejected.
[E_INTR]	<i>secs</i> is non-zero and the function did not complete in <i>secs</i> seconds or this call was blocking and an <i>xlu2intr(3X)</i> was issued.
[E_CTRLIO]	Error in opening/reading/writing on communication path to controller.
[E_TTY]	Error in controlling terminal device (LUV_DSP and LUV_TRC modes only).
[400+]	Program check (see Appendix B).
[500+]	Communication Check (see Appendix C).

If the call fails with an E_CTRLIO or E_TTY error, the error may have occurred on another session and does not indicate that the call has failed. Also, if the call fails with E_INTR, this does not indicate that the call has failed. The open may succeed at a later time. The results of the call can be determined by use of an *xlu2info(2)* call for the input **lu_chan*.

DIAGNOSTICS

Upon successful completion, *xlu2open* returns 0 with **lu_chan* set to an integer in the range 0 to 3 [see *sess_max* in *xlu2info(3X)*]. This value represents a logical unit channel for this process which will be referenced in all subsequent API calls for this logical unit.

Otherwise, a value of -1 is returned and *errapi* is set to indicate the error. For all errors other than E_INTR, 400+ and 500+, the logical unit is considered closed. In order to power on the logical unit, the *xlu2open* call must be repeated.

NAME

`xlu2puts` – put a string to the LUD_3270 logical unit's screen buffer

SYNOPSIS

```
#include <xlu2io.h>
```

```
int xlu2puts(lu_chan, s, n)
unsigned char *lu_chan, *s;
long *n;
```

DESCRIPTION

`xlu2puts` writes a string to the screen buffer at the current cursor position of the LUD_3270 mode logical unit specified by `*lu_chan`, the value returned by an `xlu2open(3X)` call.

The null terminated string is retrieved from the user buffer pointed to by `s`. If the screen field is smaller than the length of the string (excluding the terminating null), the string is truncated when placed into the screen buffer. In this case, `errapi` is set to `OPI_TOOM`, but `xlu2puts` does not fail. In all cases, `xlu2puts` returns in `*n`, the number of characters actually written to the screen buffer.

For a formatted screen buffer, the cursor is repositioned to the beginning of the next protected or unprotected field, depending on the value of the access mode [`LUAMOD`, see `xlu2ctl(3X)`] for this LU (consecutive attributes are skipped). For an unformatted screen buffer, the cursor is repositioned to the first location following the end of the transferred string.

If a screen wrap occurs during string transfer, the string transfer is continued until the field or the string is exhausted. The cursor is not positioned to the next field and an `E_OFFB` warning code is returned in `errapi`.

If an error (e.g. `OPI_NUM`) occurs during string transfer, the transfer is aborted (part of the string is transferred until the error is encountered) and the cursor is not positioned to the next field.

Character translation on the input string is performed according to the keyboard mapping specified by the customization file, `ky_ifil`, in the `xlu2init(3X)` call. Only data or numeric characters are allowed in the string.

`xlu2puts` will fail if any of the following are true, with `errapi` set as shown:

[E_ARG1]	Invalid <code>*lu_chan</code> parameter.
[E_ARG2]	<code>s</code> is the null pointer.
[E_ARG3]	<code>n</code> is the null pointer.
[E_INIT]	API initialization not performed [<code>xlu2init(3X)</code> not called].
[E_CNTX]	<code>*lu_chan</code> does not exist.
[E_DMODE]	<code>*lu_chan</code> is in <code>LUD_RAW</code> or <code>LUD_TRAW</code> mode.
[E_BUSY]	Busy logical unit.
[E_ACCS]	Access mode violation.

[OPI_NUM]	Non-numeric data entered in numeric field.
[OPI_ELSE]	Improperly positioned cursor.
[OPI_TOOM]	Field overflow (however, <i>xlu2puts</i> does not return -1; returns the size of the string transferred in <i>n</i>).
[OPI_BAD]	Bad key translation.
[OPI_WHAT]	Invalid operation.
[OPI_FUNCT]	Invalid operation.
[E_CTRLIO]	Communication with controller is terminated abnormally; session powered off. See below.
[E_TTY]	Error in controlling terminal device (LUV_DSP and LUV_TRC modes only, see <i>xlu2open</i>). See below.
[400+]	Program check (see Appendix B).
[500+]	Communication check (see Appendix C).

If the call fails with an E_CTRLIO or E_TTY error, the error may have occurred on another session. In this case, the returned value of **lu_chan* may be set to that session and will be different than the input value. This does not indicate that the call has failed for the input session. The results of this call can be determined by use of an *xlu2info(2)* call for the input **lu_chan*.

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value is returned. A return value of -1 indicates a failure and *errapi* is set to indicate the error. A return value of -2 indicates a warning condition and *errapi* is set as follows:

E_OFFB	Cursor is off the screen buffer boundary.
--------	---

NAME

xlu2read — read the next raw segment on the LUD_RAW/LUD_TRAW logical unit

SYNOPSIS

```
#include <xlu2io.h>
```

```
int xlu2read(lu_chan, segmt)
unsigned char *lu_chan;
RAWSEG *segmt;
```

DESCRIPTION

xlu2read performs a data block read operation on the LUD_RAW/LUD_TRAW logical unit channel which has the oldest queued data. The returned value of **lu_chan* is set to the logical unit, which will be one of the values returned by an *xlu2open*(3X) call.

The elements of the RAWSEG structure, defined in *xapi.h*, are described in *xlu2writ*(3X). *xlu2read* uses *segmt->s* and *segmt->n* as inputs and uses all elements, including *segmt->s* and *segmt->n*, as outputs.

xlu2read executes a non-pended read, returning to the caller even if nothing is read. Two types of messages are read: data segment and control. A control message is notification of status update, a bind or a host transmission completion. *xlu2read* returns in *segmt->typ*:

LUR_NUL	if nothing is read
LUR_END	if chain complete/cancel is detected
LUR_BND	if a bind is detected (SNA only)
LUR_STS	if status is updated
LUR_DAT	if data is read

If *segmt->typ* is set to LUR_DAT, a data segment message has been read and is contained in the buffer pointed to by *segmt->s*. The *seq*, *blk* and *cmd* fields of RAWSEG are significant only in this case.

xlu2read reads a data block of up to *segmt->n* bytes into **segmt->s*. If the input value of *segmt->n* is less than *read_max* (defined in *xapi.h*), data will be silently discarded. If the logical unit channel on which the data is read is connected to a BSC controller, the data returned in *segmt->s* is the data received following STX. On return, *segmt->n* is set to the actual number of bytes read.

A data block returned by *xlu2read* can have a maximum length of *read_max* bytes. If the host application sends data in block sizes greater than *read_max*, *xlu2read* will set the value of *segmt->seq* to indicate the segmentation performed by the communication protocol. *segmt->seq* indicates the relative position of the read data segment in a sequence of received data segments forming a data block. The segmentation is performed by the controller on a block of data which it receives from the host. *xlu2read* returns in *segmt->seq*:

LU_FIS	first in segment
LU_MIS	middle in segment
LU_LIS	last in segment
LU_OIS	only in segment

If blocking is performed by the host, *xlu2read* returns in `segmt->blk` an indication of the relative position of the read data block in a sequence of received data blocks forming a command. In this case, `segmt->blk` can have the values:

LU_DSB	first data block
LU_DSC	middle data block
LU_DSE	last data block
LU_DSO	only data block

In the case of a BSC controller where command chaining is performed by the host, *xlu2read* returns in `segmt->cmd` an indication of the relative position of the read data block in a sequence of received commands forming a chain. In this case, `segmt->cmd` can have the values:

LU_DSN	new chain
LU_DSS	same chain

xlu2read will fail if any of the following are true, with *errapi* set as shown:

[E_ARG1]	Invalid <i>*lu_chan</i> parameter.
[E_ARG2]	<i>segmt</i> is the null pointer.
[E_INIT]	API initialization not performed [<i>xlu2init</i> (3X) not called].
[E_CNTX]	<i>*lu_chan</i> does not exist.
[E_DMODE]	<i>*lu_chan</i> is in LUD_3270 mode.
[E_BUSY]	Busy logical unit.
[E_DOVFLO]	Raw outbound data has overflowed internal queue.
[E_CTRLIO]	Communication with controller is terminated abnormally; session powered off. See below.
[E_TTY]	Error in controlling terminal device (LUV_DSP or LUV_TRC modes only, see <i>xlu2open</i>). See below.
[400+]	Program check (see Appendix B).
[500+]	Communication check (see Appendix C).

If the call fails with an E_CTRLIO or E_TTY error, the error may have occurred on another session. In this case, the returned value of **lu_chan* may be set to that session and will be different than the input value. This does not indicate that the call has failed for the input session. The results of this call can be determined by use of an *xlu2info*(2) call for the input **lu_chan*.

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errapi* is set to indicate the error.

NAME

`xlu2seek` – position the cursor to a field in screen buffer for LUD_3270 logical unit

SYNOPSIS

```
#include <xlu2io.h>
```

```
int xlu2seek(lu_chan, foffset, ptrname)
unsigned char *lu_chan;
long foffset, ptrname;
```

DESCRIPTION

`xlu2seek` positions the cursor to the beginning of a field, in the screen buffer of the LUD_3270 logical unit (LU) specified by `*lu_chan`, the value returned by an `xlu2open(3X)` call.

Depending on the value of the access mode [LUAMOD, see `xlu2ctl(3X)`] for this LU, `xlu2seek` may reference protected fields, unprotected fields, or both. For an unformatted screen buffer, the entire space is considered one field and the `foffset` parameter unit is a screen buffer location. For a formatted screen buffer, the `foffset` parameter unit is a screen buffer field. Buffer formatting is indicated by the value of `c_frm` in the XLU2IBUF structure for this LU, retrieved by an `xlu2info(3X)` call.

A positive `foffset` indicates forward movement. A negative value indicates backward movement. `foffset` reflects the field offset from `ptrname`. The values for `ptrname` are:

- 0 position from the start of screen buffer
- 1 position from the current position in screen buffer
- 2 position from the end of screen buffer

`xlu2seek` does not allow the cursor to wrap freely from the end of the screen buffer (the lower right-hand corner) to the beginning (the upper left-hand corner). The movement of the cursor through the screen buffer is effected by the value of `foffset` as follows:

- If `foffset` is zero, the cursor is positioned at the beginning of the current field, which is the first non-attribute character following the first attribute character at, or preceding, the current cursor position. If a screen back wrap occurs while searching for the preceding attribute, no error is generated, providing an attribute character is contained at the end of the screen buffer. Otherwise, an error condition exists. For an error, the current cursor position is invalid and must be reset to the location the user prefers.
- If `foffset` is greater than zero, the cursor skips over the number of non-consecutive attributes, not counting the starting location. A set of consecutive attributes is the equivalent of one attribute preceded and followed by non-attribute characters. A screen wrap results in an error condition. The current cursor position is invalid and must be reset to the location the user prefers.

- If *foffset* is less than zero, the cursor skips over the number of non-consecutive attributes. If the starting location is an attribute character, it is counted as one skip. A set of consecutive attributes is the equivalent of one attribute preceded and followed by non-attribute characters. If a screen back wrap occurs when *foffset* is less than zero, no error is generated, as long as an attribute character, which exhausts *foffset*, is found at the end of the screen buffer. Otherwise, an error condition exists. For an error, the current cursor position is invalid and must be reset to the location the user prefers.

xlu2seek will fail if any of the following are true, with *errapi* set as shown:

[E_ARG1]	Invalid <i>*lu_chan</i> parameter.
[E_ARG3]	Invalid <i>ptrname</i> .
[E_INIT]	API initialization not performed [<i>xlu2init</i> (3X) not called].
[E_CNTX]	<i>*lu_chan</i> does not exist.
[E_DMODE]	<i>*lu_chan</i> is in LUD_RAW or LUD_TRAW mode.
[E_BUSY]	Busy logical unit.
[E_OFFB]	Cursor is beyond the screen buffer boundary.
[OPI_WHAT]	Invalid operation.
[OPI_FUNCT]	Invalid operation.
[E_CTRLIO]	Communication with controller is terminated abnormally; session powered off (see below).
[E_TTY]	Error in controlling terminal device (LUV_DSP and LUV_TRC modes only, see <i>xlu2open</i>). See below.
[400+]	Program Check (see Appendix B).
[500+]	Communication Check (see Appendix C).

If the call fails with an E_CTRLIO or E_TTY error, the error may have occurred on another session. In this case, the returned value of **lu_chan* may be set to that session and will be different than the input value. This does not indicate that the call has failed for the input session. The results of this call can be determined by use of an *xlu2info*(2) call for the input **lu_chan*.

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, -1 is returned and *errapi* is set to indicate the error.

NAME

`xlu2writ` — write a raw segment on the LUD_RAW/LUD_TRAW logical unit

SYNOPSIS

```
#include <xlu2io.h>
```

```
int xlu2writ(lu_chan, segmt, secs)
unsigned char *lu_chan;
RAWSEG *segmt;
unsigned secs;
```

DESCRIPTION

`xlu2writ` performs a data block write operation on the LUD_RAW/LUD_TRAW mode logical unit (LU) channel specified by `*lu_chan`, the value returned by an `xlu2open(3X)` call.

The RAWSEG structure, defined in `xapi.h`, contains the following members:

```
unsigned char *s; /* segment data */
long n;          /* segment data size */
unsigned char cod; /* segment data: EBCDIC/ASCII */
long seq;       /* segment sequence in block */
long blk;       /* block sequence in command */
long cmd;       /* command sequence in chain */
long typ;       /* segment type */
```

On call, `segmt->cod` must be set to either the value ASCII or EBCDIC (defined in `xapi.h`) to indicate the character set of the data in the buffer. EBCDIC data cannot be sent on an ASCII line.

`xlu2writ` writes `segmt->n` data bytes from the buffer pointed to by `segmt->s`. For SNA sessions, the maximum value of `segmt->n` is `writ_max` (defined in `xapi.h`) bytes. For BSC sessions, the maximum value of `segmt->n` is `writ_bsc` (defined in `xapi.h`) bytes. These values reflect the maximum block size that the physical line can handle. `writ_max` and `writ_bsc` are C language macros and are therefore compile-time values. At run-time on an SNA session, the maximum data segment size is controlled by the host when binding session ownership. `max_siz` in the XLU2IBUF structure [see `xlu2info(3x)`] indicates this value.

If the application program needs to send a block of data greater than the maximum block size, the user must segment the data and indicate the sequence of segments to `xlu2writ`. This is done by setting the value of the `segmt->blk` parameter as follows:

```
LU_DSB    data stream begins (first segment in sequence)
LU_DSC    data stream continues (middle segment in sequence)
LU_DSE    data stream ends (last segment in sequence)
LU_DSO    data stream only (only segment in sequence)
```

If `segmt->blk` indicates this is the first or only data block in the sequence, the controller is asked for permission to transmit. If permission is denied, the call will fail with `errapi` set to OPI_FUNCT or 480. If permission is granted or not needed (i.e., `segmt->blk` indicates middle or last), the data in the

buffer, at `segmt->s`, is written to the controller.

`segmt->blk` must have continuity over consecutive `xlu2writ` calls, so that the following sequence of `xlu2writ` calls will result in an error return from the final call:

An `xlu2writ`, with `segmt->blk` first/middle is successful and is followed by `xlu2writ`, with `segmt->blk` first/only.

An `xlu2writ`, with `segmt->blk` last/only, is successful and is followed by `xlu2writ`, `segmt->blk` last/middle.

If the host is sending data to the LU when `xlu2writ` is attempted, the call will fail with `errapi` set to `E_DREJ`.

On success, `xlu2writ` returns the size of the data buffer actually written in `segmt->n`.

The calling program is blocked until the data block is written to, and accepted by, the controller or until `secs` seconds have elapsed. `secs` is the maximum number of seconds `xlu2writ` will wait for the write to complete. If the call does not complete before `secs` number of seconds have elapsed, the call will fail with `errapi` set to `E_INTR`. If the call completes before `secs` seconds have elapsed, `xlu2writ` will return normally with the appropriate return code. A `secs` value of 0 indicates no timing should be performed.

`xlu2writ` will fail if any of the following are true, with `errapi` set as shown:

- [E_ARG1] Invalid `*lu_chan` parameter.
- [E_ARG2] `segmt` is the null pointer.
- [E_INIT] API initialization not performed [`xlu2init(3X)` not called].
- [E_CNCTX] `*lu_chan` does not exist.
- [E_DMODE] `*lu_chan` is in LUD_3270 mode.
- [E_BUSY] Busy logical unit.
- [OPI_FUNCT] Invalid operation.
- [E_DSEQ] Sequencing error: invalid `segmt->blk`.
- [E_DSIZ] Data segment too long.
- [E_DCOD] EBCDIC data cannot be sent on ASCII line.
- [E_DREJ] User may not send data at this time.
- [E_DOVFLO] Raw outbound data has overflowed internal queue.
- [E_INTR] `secs` is non-zero and the function did not complete in `secs` seconds or this call was blocking and an `xlu2intr(3X)` was issued (see below).
- [E_CTRLIO] Communication with controller is terminated abnormally; session powered off (see below).
- [E_TTY] Error in controlling terminal device (LUV_TRC only, see `xlu2open`). See below.

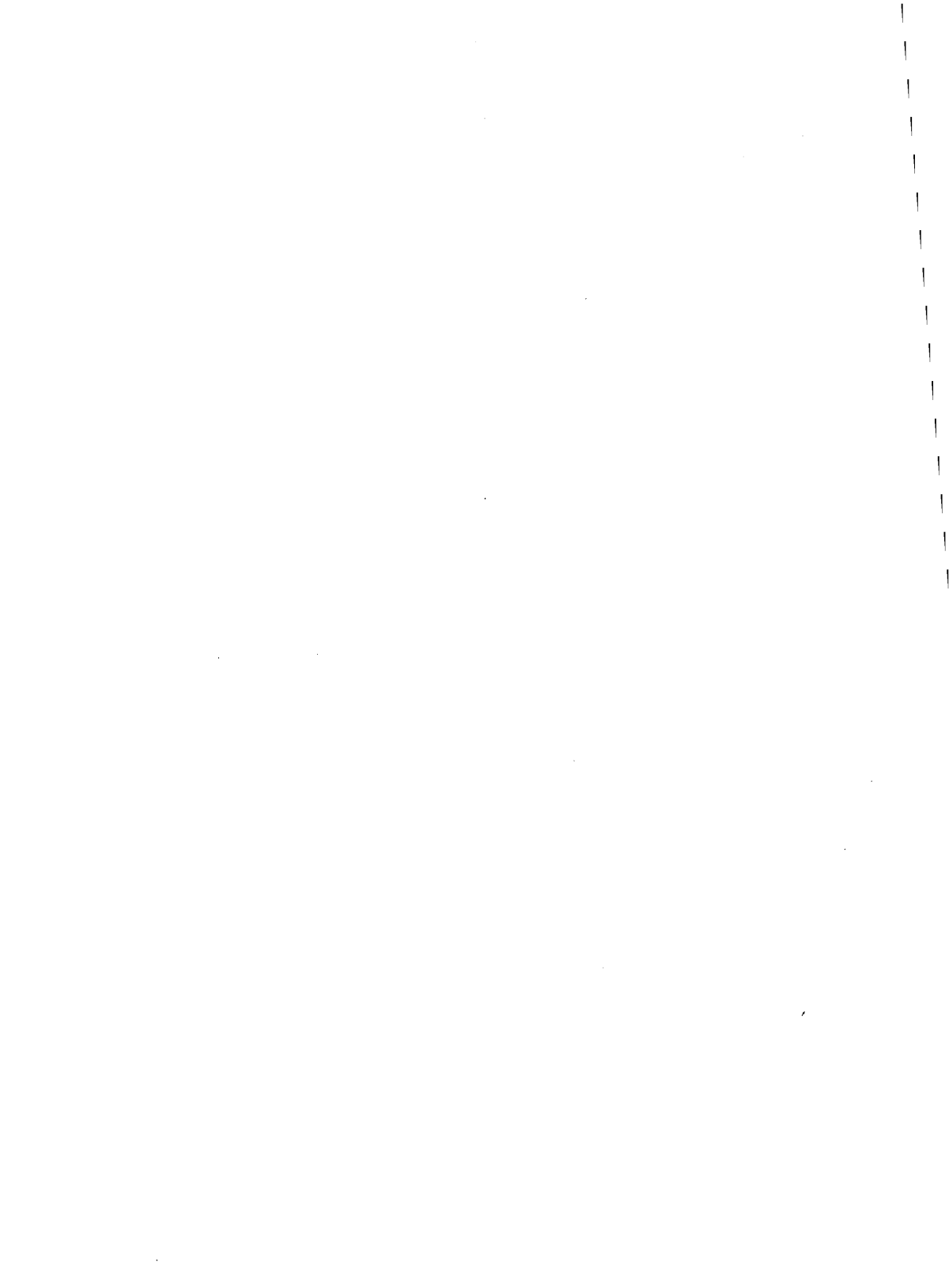
[400+] Program check (see Appendix B).

[500+] Communication check (see Appendix C).

If the call fails with an E_CTRLIO or E_TTY error, the error may have occurred on another session. In this case, the returned value of **lu_chan* may be set to that session and will be different than the input value. This does not indicate that the call has failed for the input session. The results of this call can be determined by use of an *xlu2info(2)* call for the input **lu_chan*.

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errapi* is set to indicate the error.



A

AT&T 3270 Emulator+ HLLAPI Examples

Example 1

A-1

Example 2

A-3



Example 1

This example illustrates a C language application program that records keystrokes sent to a session by a terminal operator using the `h_startkey` HLLAPI function call.

The `h_startkey` function call requires that you give explicit control of the session to the terminal user. You can do this by specifying the `CONPHYS` option with the `h_setparms` function call, and by calling the `h_connect` function call after issuing the `h_startkey` call.

```
#include <stdio.h>
#include "xhllapi.h"

#define LEN 1024

main(argc, argv)
int argc;
char *argv[];
{
    int func, ret, leng;
    char data[256];
    get_key key;
    char *malloc();

    clim(H_SETPARMS, "conphys");

    func = H_STARTKEY;
    stroke.sk_psid = 't';
    stroke.sk_option = SK_ALL;
    stroke.sk_buffer = malloc(LEN);
    leng = LEN;

    hllapi(&func, &stroke, &leng, &ret);

    clim(H_CONNECT, "t");

    key.gk_psid = 't';

    while (clim(H_GETKEY, &key) != HE_NOKEYS) {
        printf("%c %s0, key.gk_option, key.gk_buffer);
    }

    clim(func, data)
```

Example 1

continued

```
int func;
char *data;
{
    int ret, leng;

    leng = strlen(data);

    hllapi(&func, data, &leng, &ret);

    return(ret);
}
```

Example 2

This example illustrates a C language application program that allows a user to do file transfers from the command line. The executable should be built with the name `receive`, and an `ln(1)` command should be used to give the executable another name called `send`. The program interrogates which way it was invoked to determine the direction of the file transfer. The rest of the parameters are the same as for the `H_SEND` and `H_RECEIVE` commands.

NOTE

Since the parameters are given on the command line, certain characters will need to be escaped from the shell, like the colon after the presentation space short name. This can be accomplished by enclosing the particular command line parameter in double quotes. For example:
`send xhllapi.c "t:xfer.text(hllapi)" ASCII CRLF`

The program first goes to terminal emulation mode allowing the user to log on. When the host session is at the environment ready message, the user may exit the session (with `<ESC> f d`) and the file transfer will start. The ready message for TSO is "READY" and for CMS is "R;". Afterwards, the program places the user back in terminal emulation mode to log out of the host session. When the user exits the host session again, the program terminates.

Example 2

```
#include <stdio.h>
#include "hllapi.h"

main(argc, argv)
int argc;
char *argv[];
{
    int func, ret, leng;
    char data[256];
    int i;

    clim(H_SETPARMS, "comphys");

    clim(H_CONNECT, argv[2]);

    data[0] = 0;

    for (i = 1; i < argc, i++) {
        strcat(data, argv[i]);
        strcat(data, " ");
    }

    clim(*argv[0] == 'r' ? H_RECV : H_SEND, data);
    clim(H_CONNECT, argv[2]);
}

clim(func, data)
int func;
char *data;
{
    int ret, leng;

    leng = strlen(data);
    hllapi(&func, data, &leng, &ret);

    return(ret);
}
```

B

AT&T 3270 Emulator+ HLLAPI Functions

AT&T 3270 Emulator+ HLLAPI Functions

B-1



AT&T 3270 Emulator+ HLLAPI Functions

The following is a list of the AT&T 3270 Emulator+ HLLAPI supported functions:

IBM 3270 PC HLLAPI Function Number	Name	AT&T 3270 Emulator+ Symbolic Name
1	Connect Presentation Space	H_CONNECT
2	Disconnect Presentation Space	H_DISC
3	Send Key	H_SENDKEY
4	Wait	H_WAIT
5	Copy Presentation Space	H_COPY
6	Search Presentation Space	H_SEARCH
7	Query Cursor Location	H_QCUR
8	Copy Presentation Space to String	H_COPYPSS
9	Set Session Parameters	H_SETPARMS
10	Query Session	H_QSESS
11	Reserve	H_RESV
12	Release	H_REL
13	Copy Operator Information Area	H_CPOIA
14	Query Field Attribute	H_QATTR
15	Copy String to Presentation Space	H_CPSTR
16	Work Station Control	H_WSCRTL
17	Storage Manager	H_STMAN
18	Pause	H_PAUSE
20	Query System	H_QSYS
21	Reset System	H_RESET
22	Query Session Status	H_QSTATUS
23	Start Host Notification	H_STRTHOST
24	Query Host Update	H_QHOST
25	Stop Host Notification	H_STOPHOST
30	Search Field	H_SRCHFLD
31	Find Field Position	H_FNDPOS
32	Find Field Length	H_FNDLEN
33	Copy String to Field	H_CPSTRF

AT&T 3270 Emulator+ HLLAPI Functions

IBM 3270 PC HLLAPI Function Number	Name	AT&T 3270 Emulator+ Symbolic Name
34	Copy Field to String	H_CPFIELD
50	Start Keystroke Intercept	H_STARTKEY
51	Get Key	H_GETKEY
52	Post Intercept Status	H_POSTINT
53	Stop Keystroke Intercept	H_STOPKEY
90	Send File	H_SEND
91	Receive File	H_RECV
92	Invoke DOS Program	H_INVOKE
93	DOS Redirect	H_REDIR
99	Convert Position or Row Col	H_CONV

These functions are AT&T HLLAPI extensions, and are not available in the IBM HLLAPI:

Function Number	Name	AT&T 3270 Emulator+ Symbolic Name
111	Change Current Presentation Space Position	H_CHCUR
112	Write a Character in Presentation Space	H_WRCHAR
113	Connect and Interact with Presentation Space	H_CONNINT

The following IBM HLLAPI functions are not supported in the AT&T 3270 Emulator+ HLLAPI implementation:

IBM 3270 PC**HLLAPI****Function****Number****Name**

35	Define Presentation Space
36	Switch Presentation Space
37	Display Cursor
38	Display Presentation Space
39	Delete Presentation Space
54	Get 3270 AID Key

C **The xhllapi.h File**

The xhllapi.h File

C-1

The xhllapi.h File

```
/* HLLAPI Functions */

#define H_CONNECT      1      /* Connect Presentation Space */
#define H_DISC        2      /* Disconnect Presentation Space */
#define H_SENDKEY     3      /* Send Key */
#define H_WAIT        4      /* Wait */
#define H_COPY        5      /* Copy Presentation Space */
#define H_SEARCH      6      /* Search PS */
#define H_QCUR        7      /* Query Cursor Location */
#define H_COPYPSS     8      /* Copy Presentation Space to String */
#define H_SETPARMS    9      /* Set Session Parameters */
#define H_QSESS       10     /* Query Session */
#define H_RESV        11     /* Reserve */
#define H_REL         12     /* Release */
#define H_CPOLA       13     /* Copy Operator Information Area */
#define H_QATTR       14     /* Query Field Attribute */
#define H_CPSTR       15     /* Copy String to PS */
#define H_WSCTRL      16     /* Work Station Control */
#define H_STMAN       17     /* Storage Manager */
#define H_PAUSE       18     /* Pause */
#define H_QSYS        20     /* Query System */
#define H_RESET       21     /* Reset System */
#define H_QSTATUS     22     /* Query Session Status */
#define H_STRTHOST    23     /* Start Host Notification */
#define H_QHOST       24     /* Query Host Update */
#define H_STOPHOST    25     /* Stop Host Notification */
#define H_SRCHFLD     30     /* Search Field */
#define H_FNDPOS      31     /* Find Field Position */
#define H_FNDLEN      32     /* Find Field Length */
#define H_CPSTRF      33     /* Copy String to Field */
#define H_CPFIELD     34     /* Copy Field to String */
#define H_DEFPS       35     /* Define Presentation Space */
#define H_SWITCHPS    36     /* Swlth Presentation Space */
#define H_DISPCUR     37     /* Display Cursor */
#define H_DISPPS      38     /* Display Presentation Space */
#define H_DELP        39     /* Delete Presentation Space */
#define H_STARTKEY    50     /* Start Keystroke Intercept */
```

The xhllapi.h File

```
#define H_GETKEY      51      /* Get Key */
#define H_POSTINT    52      /* Post Intercept Status */
#define H_STOPKEY    53      /* Stop Keystroke Intercept */
#define H_GETAID     54      /* Get 3270 AID Key */
#define H_SEND       90      /* Send File */
#define H_RECV       91      /* Receive File */
#define H_INVOKE     92      /* Invoke DOS Program */
#define H_REDIR      93      /* DOS Redirect */
#define H_CONV       99      /* Convert Position or Row Col */
#define H_CHCUR      111     /* Change cursor Position in PS */
#define H_WRCHAR     112     /* Write a character in PS */
#define H_CONNINT    113     /* Connect Interactive */

/* Array Sizes */

#define LONG_NAME 8

/* Return Codes */

#define HE_SUCCESS   0      /* Good Return */
#define HE_INVALID   1      /* Invalid PS */
#define HE_PARM      2      /* Parameter error or Invalid Function */
#define HE_WSCtrl    3      /* WS Ctrl action has occurred */
#define HE_BUSY      4      /* Target PS busy */
#define HE_INHBT     5      /* Function Inhibited */
#define HE_LENGTH    6      /* Data Error */
#define HE_POS       7      /* Invalid PS Position */
#define HE_PROC      8      /* Function Procedure Error */
#define HE_SYSEERR   9      /* System Error */
#define HE_FUNCT     10     /* Function Unavailable */
#define HE_RSRC      11     /* Resource Unavailable */
#define HE_OIA       21     /* Updated OIA */
#define HE_PRES      22     /* Updated presentation space */
#define HE_BOTH      23     /* Both of the above have been updated */
#define HE_DATA      8000   /* Only data portion has been updated */
#define HE_NOFIELD   24     /* No such field */
#define HE_NOKEYS    25     /* Requested keys are not available */
#define HE_UPDATE    26     /* A host presentation space or OIA has */
                             /* been updated */

#define HE_FNUM      301    /* Invalid function number */
#define HE_NOENT     302    /* File not found */
```

```
#define HE_ACCESS      305    /* Access denied */
#define HE_MEM        308    /* Insufficient memory */
#define HE_ENV        310    /* Invalid environment */
#define HE_FORM       311    /* Invalid format */
#define HE_PSID       9998   /* Invalid presentation space */
#define HE_NOTPR      9999   /* Parameter not 'p' or 'r' */

/* Hardware Base */

#define HB_3270XT     'X'    /* 3270 PC or 3270 PC XT */
#define HB_3270AT     'A'    /* 3270 PC AT */
#define HB_UNKNOWN    'U'    /* Unable to Determine */

/* Control Program Type */

#define CP_3270PC     'C'    /* 3270 PC Control Program */

/* Control Program Level */

#define CL_ONE        '1'    /* CP Level 1.22 */
#define CL_TWO        '2'    /* CP Level 2.10 */

/* Query System data string */

typedef struct {
    char sy_vernum;        /* HLLAPI Version Number */
    char sy_levnum[2];    /* HLLAPI Level Number */
    char sy_date[6];      /* HLLAPI Date (MMDDYY) */
    char sy_limver;       /* LIM Version */
    char sy_limlev[2];    /* LIM Level */
    char sy_hwbase;       /* Hardware Base */
    char sy_cptype;       /* Control Program Type */
    char sy_cplevel;     /* Control Program Level */
    char sy_resv1;        /* Reserved */
    char sy_resv2[2];     /* Reserved */
    char sy_psid;         /* Session Short Name */
    char sy_exterr1[4];   /* Extended Error Code 1 */
    char sy_exterr2[4];   /* Extended Error Code 2 */
    char sy_resv[8];      /* Reserved */
} q_system_data;
```

The xhllapi.h File

```
/* Session Types */

#define ST_HOST      'H'      /* Host Session */
#define ST_NOTE     'N'      /* Notepad Session */
#define ST_PC       'P'      /* PC and Alternate Sessions */
#define ST_DFT      'D'      /* DFT Mode */
#define ST_CUT      'C'      /* Cut Mode */

/* Query Sessions data string */

typedef struct {
    char qe_psid;           /* Short name of session */
    char qe_lname[LONG_NAME]; /* Long name of session */
    char qe_stype;        /* Session Type */
    short qe_size;        /* PS Size */
} q_sessions_data;

/* Presentation Space Id's */

#define PS_CURR      ' '      /* Currently connected PS */
#define PS_CURR2    0x0      /* Same */
#define PS_PCRUN    '*'      /* PC session AP running in */
#define PS_WSCtrl   '#'      /* WS Control session */

/* Session Characteristics */

#define SC_EAB 0x80          /* Extended attribute bytes */
#define SC_PSS 0x40          /* Programable symbols supported */

/* Query Session Status data string */

typedef struct {
    char qt_psid;           /* Short name */
    char qt_lname[LONG_NAME]; /* Long name */
    char qt_stype;        /* Session type */
    char qt_schars;       /* Session Characteristics */
    short qt_rows;        /* Rows in presentation space */
    short qt_cols;        /* Columns in PS */
}
```

```
        char qt_pifstat[2];          /* PIF Status */
        char qt_reserved;           /* Reserved */
    } q_status_data;

typedef unsigned char uchar;

/* Copy OIA data string */

typedef struct {
    char    cp_format;              /* OIA Format byte */
    char    cp_image[80];          /* OIA Image Group */

    /* OIA Indicator Group */

    /* Group 1: On-line and screen ownership */

#define SETUP          0x80      /* Setup mode */
#define TEST          0x40      /* Test mode */
#define SSCPOWN       0x20      /* SSCP-LU session owns screen */
#define LUOWN         0x10      /* LU-LU session owns screen */
#define UNOWN         0x08      /* Online and not owned */
#define READY         0x04      /* Subsystem ready */
    uchar   group_1;

    /* Group 2: Character selection */

#define EXTEND         0x80;     /* Extended Select */
#define APL            0x40;
#define KANA           0x20;
#define ALPHA          0x10;
#define TEXT           0x08;
    uchar   group_2;

    /* Group 3: Shift state */

#define NUMERIC        0x80;     /* Numeric Shift */
#define SHIFT          0x40;     /* Upper Shift */
    uchar   group_3;
```

The xhllapi.h File

```
/* Group 4: PSS group 1 */
/* Group 5: Highlight group 1 */
/* Group 6: Color group 1 */

#define SELECT      0x80;      /* Operator Selectable */
#define INHERIT     0x40;      /* Field Inherit */
    uchar  group_4;
    uchar  group_5;
    uchar  group_6;

/* Group 7: Insert */

#define INSERT 0x80; /* Insert mode */
    uchar  group_7;

/* Group 8: Input inhibited */

/* Group 8: Byte 1 */

#define CHECK      0x80; /* Non-resettable machine check */
#define KEY        0x40; /* Reserved for security key */
#define MACHINE    0x20; /* Machine Check */
#define COMM       0x10; /* Communications Check */
#define PROGRAM    0x08; /* Program check */
#define RETRY      0x04;
#define NWORKING   0x02; /* Device not working */
#define VBUSY      0x01; /* Device very busy */

/* Group 8: Byte 2 */

#define BUSY       0x80; /* Terminal busy */
#define WAIT       0x40; /* Terminal wait */
#define SYMBOL     0x20; /* Minus symbol */
#define FUNCTION   0x10; /* Minus function */
#define TOOMUCH    0x08; /* Too much entered */
#define NENOUGH    0x04; /* Not enough entered */
#define WRONG      0x02; /* Wrong number */
#define NUMBER     0x01; /* Numeric field */
```

```
/* Group 8: Byte 3 */

#define UNAUTH      0x40; /* Operator unauthorized */
#define UNAUTHM    0x20; /* Operator unauthorized minus function */
#define IDEAD      0x10; /* Invalid dead key combination */
#define WPLACE     0x08; /* Wrong placed */

/* Group 8: Byte 4 */

#define PENDING    0x80; /* Message pending */
#define PARTITION  0x40; /* Partition wait */
#define SYSTEM     0x20; /* System wait */
#define MISMATCH   0x10; /* Hardware mismatch */
#define NCONFIG    0x08; /* LU not configured at control unit */

/* Group 8: Byte 5 */

#define AUTOKEY    0x80; /* Autokey inhibit */
#define INPUT      0x40; /* Application program has operator */
                    /* input inhibited */

uchar group_8[5];

/* Group 9: PSS Group 2 */
/* Group 10: Highlight Group 2 */
/* Group 11: Color Group 2 */

#define SELECT     0x80; /* Selected */
#define DISABLE    0x40; /* Display disabled (Group 9 only) */
uchar group_9;
uchar group_10;
uchar group_11;

/* Group 12: Communications error reminder */

#define ERROR      0x80; /* Communications error */
#define MONITOR    0x40; /* Response time monitor */
uchar group_12;
```

The xhllapi.h File

```
/* Group 13: Printer status */

#define CUSTOM      0x80; /* Printer code not customized */
#define MALFUNC    0x40; /* Printer malfunction */
#define PRINTING   0x20; /* Printer printing */
#define ASSIGN     0x10; /* Assign printer */
#define WHAT      0x08; /* What printer */
#define PRINTER   0x04; /* Printer assignment */
    uchar group_13;

/* Group 14 & 15: Reserved */

    uchar group_14;
    uchar group_15;

/* Group 16: Autokey play/record status */

#define PLAY       0x80;
#define RECORD    0x40;
    uchar group_16;

/* Group 17: Autokey abort/pause state */

#define OVERFLOW  0x80; /* Recording overflow */
#define PAUSE     0x40;
    uchar group_17;

/* Group 18: Enlarge state */

#define ENLARGE   0x80; /* Window is enlarged */
    uchar group_18;
} cpoia;

#define HN_PRESENTATION 'P' /* Notify if updated presentation space */
#define HN_OPERATOR 'O' /* Notify if updated operator info area */
#define HN_BOTH 'B' /* Both of the above */
#define HN_DATA 'D' /* Notify only if data and not attributes */
/* have been updated */
```



```
typedef struct {
    char    lm_psid;           /* Presentation Space ID */
    char    lm_type;          /* Update type */
    char    *lm_buffer;
} host_notify;

/*****
/*These structures must be used by program sending strings to WS_CTRL. */
*****/
#define  CMDSIZ  2
#define  NAMES  20
struct ca_data_type
{
    char    ws_cmd[CMDSIZ];
    char    ws_equ;
    char    ca_scrn;
    char    ca_psid;
} ;    /* data from CHANGE ACTIVE */

struct cb_data_type
{
    char    ws_cmd[CMDSIZ];
    char    ws_equ;
    char    cb_psid;
    short   cb_srcbeg;
    short   cb_srcend;
    char    cb_tpsid;
    short   cb_tbeg;
} ;    /* data from COPY BLOCK */

struct rd_data_type
{
    char    ws_cmd[CMDSIZ];
    char    ws_equ;
    char    rd_scrn;
} ;    /* data from REDRAW SCREEN */
```

The xhllapi.h File

```
struct qa_data_type
{
    char    ws_cmd[CMDSIZ];
    char    ws_equ;
    char    qa_scrm;
    char    qa_psid;
}; /* data from QUERY ACTIVE */

struct qc_data_type
{
    char    ws_cmd[CMDSIZ];
    char    ws_equ;
    char    qc_scrm;
    char    qc_fill;
    char    qc_code;
}; /* data from QUERY COLOR */

struct qe_data_type
{
    char    ws_cmd[CMDSIZ];
    char    ws_equ;
    char    qe_fill[2];
    short   qe_code;
}; /* data from QUERY ENLARGE */

struct qw_data_type
{
    char    ws_cmd[CMDSIZ];
    char    ws_equ;
    char    qw_scrm;
    char    qw_psid;
    short   qw_rows; /* number of rows in window */
    short   qw_cols; /* number of cols. in window */
    short   qw_rpos; /* row position, upper left corner */
    short   qw_cpos; /* col position, upper left corner */
    short   qw_wcolor; /* window color code */
    short   qw_bcolor; /* border color code */
    short   qw_flags; /* control flags */
    short   qw_psrow; /* row on PS of upper left corner */
    short   qw_pscol; /* col on PS of upper left corner */
}; /* data from QUERY WINDOW */
```

```
struct qn_data_type
{
    char    ws_cmd[CMDSIZ];
    char    ws_equ;
    char    qn_scrn;
    char    qn_psid;
    char    qn_names[NAMES];
};    /* data from QUERY NAMES */

#define SK_AID        'D'        /* Aid keystrokes */
#define SK_ALL        'L'        /* All Keystrokes */

typedef struct {
    char    sk_psid;            /* Presentation Space ID */
    char    sk_option;        /* Option Code */
    char    *sk_buffer;
} start_keystroke;

#define GK_ASCII        'A'        /* ASCII character returned */
#define GK_MNEM        'M'        /* Mnemonic */
#define GK_SHIFT        'S'        /* Special shift */

typedef struct {
    char    gk_psid;            /* Presentation Space ID */
    char    gk_option;        /* Option code character */
    char    gk_buffer[4];
} get_key;

#define PI_ACCEPT        'A'
#define PI_REJECT        'R'

typedef struct {
    char    pi_psid;            /* Presentation Space ID */
    char    pi_option;        /* Option code */
} post_intercept;
```

D File Transfer Messages

File Transfer Messages

D-1

File Transfer Messages

This appendix lists the file transfer messages in numerical order and describes them. The message number and text appear in **bold type**, followed by an explanation and user response.

INDFT001 File transfer command being processed

This message appears when the file transfer command is entered and the system begins processing.

User response: none; wait for message INDFT002 to appear.

INDFT002 Number of bytes of file transferred so far: == < xxxxxxxx

This message lets you know how many bytes of the file have been transferred to or from the host. The number is updated as the file is transferred.

User response: none; wait for message INDFT003 to appear.

INDFT003 File transfer complete

The file transfer was successful.

User response: none.

INDFT004 File transfer complete with records segmented

The file transfer was complete, but records greater than the set logical record length of the file appended will divide and become multiple records.

User response: none.

INDFT005 Filespec incorrect: file transfer canceled

you entered some part of the filespec incorrectly, e.g., the path or filename.

User response: compare the filespec in the file transfer

command, which will still be visible in the session, with the filespec requirements for your system. If it is correct, it is possible that the file does not exist.

INDFT006 Command incomplete: File transfer canceled

The user did not enter any parameters after Send or Receive.

User response: check the Send and Receive command requirements and retry.

INDFT010 Host has not responded with timeout period: Refer to reference manual for more information

The host has not responded to the file transfer within several seconds.

User response: if the host session screen shows HOLDING, you can start file transfer by switching to the host session and pressing PA2 (in TSO). If X SYSTEM or X appears in the host session screen operator information, wait for it to clear. These specify that the system is working slowly. If you want to stop file transfer after several time-out messages appear, switch to the host session, press Reset to clear the operator information area, and press PF2 to stop the file transfer, or CLEAR to continue. This state can be caused by line problems. If IND\$FILE is not installed at the host, jump to the application program session and press Ctrl+Break to stop the file transfer.

E

API Program Check Error Codes

API Program Check Error Codes

E-1

API Program Check Error Codes

This appendix lists the values of *errapi* that can be returned when API system calls fail because of errors in the data received from the host.

Value	Explanation
401	unknown data stream command
402	invalid buffer address in data stream
403	data follows 1-byte commands in data stream
404	data stream ends in order-pending state
405	invalid source device on copy command, or source device buffer locked on copy command, or source and destination device incompatibility on copy command
406	ESC character missing in second position of command sequence
411	Request/Response Unit (RU) too long (LU.T1)
413	function not supported
420	exception response request received when definite response only specified by BIND
421	definite-response request received when exception response only specified by BIND
422	NO response not allowed
423	format indicator not allowed
430	sequence number error
431	chaining error
432	bracket error
433	data traffic inactive
434	direction error
443	read command must have Change Direction but not End Bracket
445	Activate Logical Unit (ACTLU) request is for neither cold activation nor Error Recovery Procedure (ERP)
450	BIND profile error
451	BIND primary protocol error
452	BIND secondary protocol error
453	BIND common protocol error
454	BIND screen size error
455	BIND LU profile error
456	BIND LU1 error
457	BIND cryptography specified
462	data stream error detected by LU.T1
470	unknown data byte X'00' - X'3F' or X'FF'

API Program Check Error Codes

Value	Explanation
480	user request lost due to host SELECT (BSC) or arrival of message from controller
481	user request lost due to host SELECT with bad command (BSC only)
482	host SELECT received with response to user request (BSC only)
483	host SELECT with bad command received with response to user request (BSC only)
490	buffer not available for write command
498	negative response received
499	exception request received

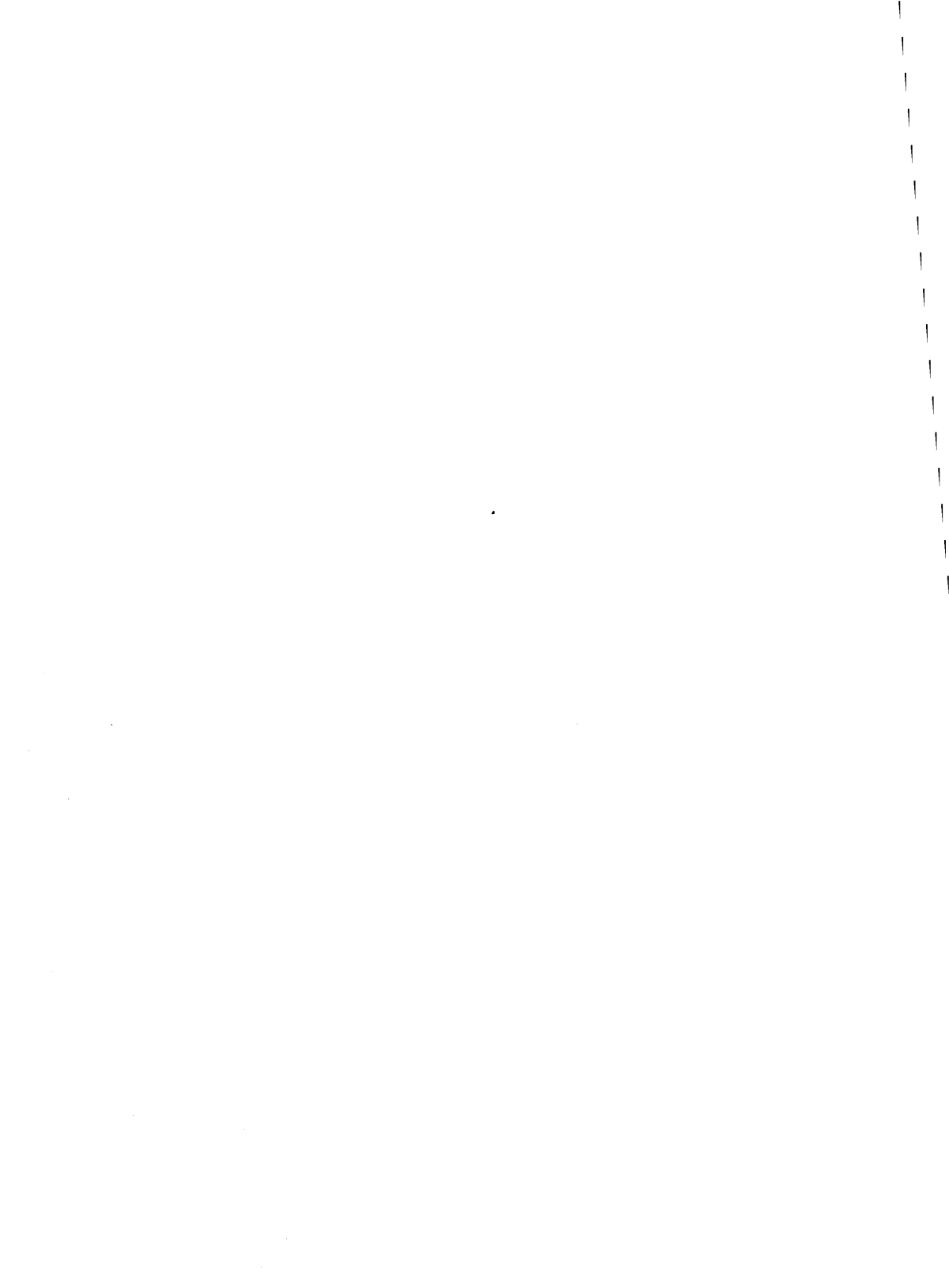
For additional information, see the IBM publication *IBM 3270 Information Display System, 3274 Control Unit Description and Programmer's Guide, GA23-0061*.

F

API Communication Check Error Codes

API Communication Check Error Codes

F-1



API Communication Check Error Codes

This appendix lists the values of *errapi* that can be returned when API system calls fail because of conditions detected at the local communications interface.

Value	Explanation
501	Data Set Ready (DSR) lost
502	Clear To send (CTS) lost
504	Normal Disconnect Mode (NDM)
505	NDM
510	Physical Unit (PU) is not active (this is SNA condition)
518	segmentation error (internal Deactivate Physical Unit (DACTPU)) (this is an SNA condition)
519	received frame too long
520	timeout (no frames)
521	timeout (no flags)
525	20 Exchange Identification (XID) commands received in a row
528	Frame Reject Response (FRMR) sent; Frame Reject Mode (FRM) entered (internal DACTPU)
529	modem acting up (internal DACTPU)
530	clocking or CTS lost (internal DACTPU)
531	received STX....ENQ (BSC only)
532	idle timeout
534	protocol timeout
593	received EOT instead of ACK (BSC only)
594	received RVI to ETB block (BSC only)
595	lost CD while receiving (BSC only)

For additional information, see the IBM publication *IBM 3270 Information Display System, 3274 Control Unit Description and Programmer's Guide, GA23-0061*.



G

API User/System Error Codes

API User/System Error Codes

G-1

API User/System Error Codes

This lists the values of *errapi* that can be returned when an API system call fails because of user application or system errors.

Error Message	Explanation/Possible Cause
1 OPI_NUM	Non-numeric data entered in numeric field
2 OPI_SYS	The keyboard has been disabled by the host
3 OPI_WHAT	Invalid operation
4 OPI_FUNCT	Invalid operation
5 OPI_ELSE	Improperly positioned cursor
6 OPI_TOOM	A field overflow has occurred
7 OPI_BAD	Bad key translation
201 E_ARG1	sc_ifil is the null pointer invalid lu_chan parameter
202 E_ARG2	ky_ifil is the null pointer invalid func parameter invalid request s is the null pointer segmt is the null pointer
203 E_ARG3	invalid lu_port parameter invalid arg invalid ptrname n is the null pointer the value at n is less than 1
204 E_ARG4	invalid ludmod parameter invalid parameter retcod is NULL (LUEVENT with KY_CTRL only)
205 E_ARG5	invalid luvmod parameter
206 E_OFFB	cursor is off the screen buffer boundary
207 E_KCTL	error in keyboard control file
208 E_SCTL	error in screen control file
209 E_RSRC	controller has insufficient resources
210 E_BNDREJ	host mismatch, bind rejected
211 E_TRMBSY	SNA/BSC terminal is busy
212 E_HOSTID	illegal SNA/BSC host identification
213 E_CTRLIO	communication controller problem
214 E_TTY	terminal device control problem
215 POW_OFF	te3278 emulator terminated

Error Message	Explanation/Possible Cause
216 E_ACCS	access mode violation
217 E_CNTR	*lu_chan does not exist
218 E_BUSY	BUSY state mismatch on API call
219 E_INIT	API initialization not performed
220 E_SESLIM	number of sessions exceeds limit
221 E_DMODE	lu is in LUD_3270 mode lu is in LUD_RAW mode lu is in LUD_RAW mode or could not enter requested data transfer mode
222 E_DSIZ	data segment too long
223 E_DSEQ	sequencing error - invalid segmt->blk
224 E_DREJ	controller does not acknowledge data
225 E_ILSIZE	size mismatch between bind and device (bind command is legal)
226 E_MODL	physical terminal (as described by sc_ifil) cannot support model number
227 E_LINID	illegal line identification
228 E_CNTID	illegal controller identification
229 E_DOVFLO	raw data buffer overflow
230 E_PATH	pu_name is the null pointer invalid name parameter
231 E_RTMG	error in opening runtime message file
232 E_XLIB	excluded library function
234 E_NASCII	lu on ASCII line but ses is non-ASCII
235 E_DCOD	LUD_TRAW is unavailable on ASCII line
236 E_XPRNT	request for transparency rejected
237 E_INTR	primitive aborted by interrupt
238 E_ICOND	invalid inhibit condition (LURSET)
239 E_SECUR	invalid security mode request

H

API LUV_TRC Status Displays

API LUV_TRC Mode Status Line Displays

H-1

API LUV_TRC Mode Status Line Displays

An API trace display is output in the status line. Each trace line is of the general form:

`xlu2... n1 misc_params lu: n2 err: n3`

where:

`xlu2...` is the name of the API function

`n1` is the one digit numerical field containing the called value of `lu_chan` (except for `xlu2open`, see below).

`misc_params` are miscellaneous parameters displayed for this call, as described below

`n2` is the one digit numerical field containing the returned value of `lu_chan`

`n3` is the three digit numerical field containing the value of the error code

Trace output, particularly the `misc_params` fields, for each API call trace are described here. The manual pages should be referenced for each description.

`xlu2open pu_name np ludmod luvmmod lu: n2 err: n3`

where:

`pu_name` controller process (in place of `n1`; only the returned value of `lu_chan`, `n3`, is used)

`np` is the three digit field containing the value of `lu_port`

`ludmod` is the one digit field value of `ludmod` (see `xapi.h`)

`luvmmod` is the one digit field value of `luvmmod` (see `xapi.h`)

`xlu2clos lu: n2 err: n3`

has no `misc_params` field entries

API LUV_TRC Mode Status Line Displays

`xlu2func n1 LU... lu: n2 err: n3`

where:

`LU...` is the *func* followed by *name*: LUPRND, LUPRNF followed by the file name, LUPRNC followed by the command string, or LURSET followed by the Reset-Inhibit-condition mode

`xlu2ctl n1 LU... na pos: np col: nc lin: nl lu: n2 err: n3`

where:

`LU...` is the *request*: LUDTIM, LUEVENT, LUEVIMED, LUDMOD, LUVMOD or LUAMOD

`na` is the three digit field value of *arg* (see `xapi.h`)

`np` is the four digit field value of the current cursor position relative to the start of the buffer

`nc` is the three digit field value of the current cursor column

`nl` is the two digit field value of the current cursor line

`xlu2seek n1 foffset: nf ptrname: nn pos: np col: nc lin: nl lu: n2 err: n3`

where:

`nf` is the four digit field value of *foffset*

`nn` is the one digit field value of *ptrname*:

and all other fields are as in `xlu2ctl`.

`xlu2gets n1 req: nr ret: ssss nt pos: np col: nc lin: nl lu: n2 err: n3`

where:

`nr` is the four digit field value of the buffer length

ssss is the initial five characters of the returned string
nt is the four digit field value of the returned string length
and all other fields are as in *xlu2ct1*.

xlu2puts n1 string ret: nt pos: np col: nc lin: nl lu: n2 err: n3

where:

string is the initial four characters of the returned string
nt is the four digit field value of the transferred string length
and all other fields are as in *xlu2ct1*.

xlu2writ n1 nr seq: ns ret: nt lin: nd lu: n2 err: n3

where:

nr is the four digit field value of the data buffer length (*segmt->n*)
ns is the four digit field hex value of the data block sequence
(*segmt->blk*)
nt is the four digit field value of the written data length (*segmt->n*)

xlu2read n1 req: nr ret: nt siz: nz seq: ns lu: n2 err: n3

where:

nr is the four digit field value of the requested buffer length
(*segmt->n*)
nt is the one digit field value of *segmt->typ*
nz is the four digit field value of the size of the returned data
(*segmt->n*)
ns is the four digit field hex value of the relative position of the
data block (*segmt->blk*)



API External Symbols

API External Symbols

I-1

API External Symbols

This appendix lists the global symbols used by the API library. Application programs may not define these symbols externally.

ACtvbf	ACTvqe	AEvtad	AEvtst	API_acs	API_cls
API_ctl	API_dbg	API_get	API_iupd	API_opn	API_pos
API_put	API_ret	API_sek	APaddr	APcall	APtest
APutae	BAktab	BEGfind	BUfdbg	BUffup	CALLzz
CFld_atr	CUr_lt	CUr_rt	CVta2e	DIScae	DIScqce
DSa2c	DSca2a	DScvtc	DSe2a	DSeau	DSersg
DSeua	DSfa	DSfu	DSisbt	DSldb	DSovra
DSpcrt	DSptab	DSptat	DSrdmi	DSrdzz	DSread
DSrta	DSsba	DSscsr	DSscsw	DStuf3	DSvscs
DSwini	DSwrit	ENDfind	EToa	Flxaev	FLd1_ok
FORatt	FWdtab	GLiquid	HEXdump	HILITE	INIt_exit
IPC	IPCbf	IPCinz	IPCpdf	IPCqdf	IPCqui
IPCtcb	IPCtdf	IPCtki	IPCtrace	IPCxeq	ITOA
KIktsk	KYmap	KYtask	KYtcb	LCIpid	MOveae
OUtmap	PANic	PLuscur	POol0	POol0b	POol0x
PRnit	PUtbuf	PUtins	QEvtad	QEvtst	QPopqe
QUeu0	REverse	SCadv	SCbakt	SCcrsl	SCdata
SCdown	SCds01	SCds02	SCds03	SCds04	SCe2a1
SCeeof	SCeras	SChome	SCkey	SClear	SCleft
SCopyi	SCopyo	SCrigt	SCsna	SCtab	SCtask
SCupro	SCuprt	SCvppt	SEndae	SHo_stat	SIgcat
SIgign	ST_fld	SUstsk	TAskst	TH2buf	THAPI
THanal	THaw	THcola	THctrl	THcura	THdlch
THdraw	THEcho	THence	THeras	THerms	THexec
THflsh	THgopp	THgoto	THgoxy	THgtfg	THgtnm
THgtnt	THgtst	THinch	THinit	THink	THipti
THkeyb	THlick	THluiz	THong	THopia	THough
THouts	THpip0	THpipr	THpipw	THpipz	THpnrd
THprin	THputs	THrall	THRill	THshlx	THsig0
THsig1	THsig9	THstrm	THtask	THud	THug
THumb	THus	THusag	THutrm	THwack	THxptx
THzigz	THztrm	UFxaev	UPdate_crt	actvbf	actvqe

API External Symbols

aecb	aevtad	aevtst	apaddr	apcall	api_cls
api_ctl	api_dat	api_dbg	api_fun	api_get	api_iupd
api_opn	api_put	api_red	api_ret	api_sek	api_wrt
apiintr	aptest	aputae	baktab	begfind	bfaval
bltp	buffup	callzz	cfld_atr	chk_prm	clear_map
ctl_prim	cur_lt	cur_rt	cvta2e	dbgfle	det_att
det_clr	det_err	det_fea	discae	discqe	dsa2c
dsalts	dsc2a	dscvtc	dse2a	dseau	dseclr
dsehlt	dsersg	dseua	dsfa	dsfu	dsisbt
dsldb	dsmf	dsovra	dspcrt	dsptab	dsptat
dsrdmi	dsrdzz	dsread	dsrta	dssa	dssba
dsscsr	dsscsw	dssfe	dstuf3	dsvscs	dswini
dswrit	endfind	errapi	errjoin	etoa	find_attr
fixaev	fld1_ok	fldnam	foratt	fwd_att	fwdtab
gldat	glquid	hexdump	hexrel	inibit	init_exit
ipc	ipcbfi	ipcinz	ipcpdf	ipcqdf	ipcqui
ipctcb	ipctdf	ipctki	ipctrace	ipcxeq	is_att
kiktsk	kymap	kytask	kytcb	lclpid	lu2close
lu2ctl	lu2gets	lu2info	lu2open	lu2puts	lu2seek
lu_exst	mov_chrs	mov_numb	moveae	old_func	outmap
panic	pluscur	pool0	pool0b	pool0x	pr_primd
prmnam	prn_arg	prnit	putbuf	putins	qevtad
qevtst	qpopqe	qu0qcb	queu0	rawlib	rcv_msg
replay	req_cmpl	rset_map	scadat	scadv	scafin
scafun	scaget	scapi	scapos	scaput	scared
scarep	scasek	scasnd	scawrt	scbakt	scbonz
sccold	sccrsl	scdata	scdims	scdisp	scdown
scds01	scds02	scds03	scds04	sce2a1	sceeof
sceras	scexec	schome	sclear	sclft	scmgst
scnxte	scopyi	scopyo	scrigt	scsini	scsna
scsswp	sctab	sctask	scupro	scupxx	scvg
scvpprt	scvx	sendae	set_cses	set_map	sho_stat
showbf	sigala	sigcat	sigign	st_fld	sttic_if
sustsk	t	taskst	th2buf	th5rng	thVMIN
thapi	tharaw	tharst	thasav	thasgn	thaw
thcola	thcolr	thctrl	thcura	thdlch	thdpsl
thdraw	thecch	thecho	theclr	thence	theras
therms	thexec	thexta	thflsh	thgopp	thgoto
thgoxy	thgtfg	thgtnm	thgtnt	thgtst	thinch
thinit	think	thintr	thipti	thisgi	thkeyb

thlick	thlnk0	thlnkr	thlnkw	thlnkz	thofhl
thong	thopia	though	thouts	thpip0	thpipk
thpipr	thpipw	thpipz	thprin	thputs	thrall
thrill	thscld	thshlx	thsig0	thsig1	thstrm
thtask	thud	thug	thutrm	thwack	thxptx
thztrm	tinfo	tsep	ufxaev	unodef	update_crt
usr_01	usr_02	usr_03	usr_04	usr_05	usr_06
usr_07	usr_08	usr_09	usr_10	vnonlib	wPrt
ww	xlat	xlu2clos	xlu2ctl	xlu2func	xlu2gets
xlu2info	xlu2init	xlu2intr	xlu2open	xlu2puts	xlu2read
xlu2seek	xlu2writ				

J **Glossary**

Glossary

J-1



Glossary

- attention identifier (AID) key**
A non-ASCII control key (for example, **clear**, **PF**, **PA**, or **Enter**).
- American National Standard Code for Information Interchange (ASCII)**
One of the two standard codes used for exchanging information among data processing systems and associated equipment.
- attribute byte**
In 3270 applications this is the byte used for data security to indicate whether a certain field can be, or has been, updated.
- autoskip**
The cursor will automatically skip to the next unprotected field when it encounters a field that is both protected and numeric.
- combination keys**
Keys that must be used in conjunction with another key(s) to produce a specific function. These keys include **Control** and **Shift**.
- copy**
The function that allows you to mark source lines in one location and move them to a target location.
- current connected presentation space**
The active session to which you are connected.
- extended error code**
A data string generated by an internal system error that is used by the AT&T Tier 4 Support Group for diagnosis.
- logging on**
The procedure by which you provide user identification, and generally a password, before establishing communications with an AT&T 3B computer or a remote host computer.

menu A list of available options from which you select the one that you want.

operator information area (OIA)

The bottommost line of your screen, where you receive information about the status of your work station and the remote host computer.

presentation space (PS)

A region in computer storage that can be displayed on your terminal screen.

PSID

Presentation Space Identifier; short name of the presentation space.

session

A connection between your work station and a remote host computer.

short name

The one letter name (A through Z) of a presentation space.

source presentation space

The presentation space from which information is transported using a copy function.

target presentation space

The presentation space in which the information is placed.

terminal operator

A human user of a HLLAPI application program.

unprotected field

A field that is available for the operator to use to enter or modify data.

X

Index

Index

X-1

Index

3270 data stream mode 1: 8; 4: 1, 9, 11

A

API 1: 1, 8; 4: 1-4, 6-8, 10-14
AT&T 3270 Emulator+ HLLAPI Tutorial 1:
7; 3: 1-2, 4, 9, 11
attribute bytes 2: 13

C

C language functions 1: 1, 8
calling parameters 2: 8-9; 3: 8
Combination keys 2: 2
Communications Functions 1: 7; 2: 4; 3: 2-3

D

data arguments 2: 8
distributed processing 1: 3

E

Environment Functions 1: 7; 2: 7; 3: 3, 8
environment variables 2: 12; 3: 11

F

File Transfer Functions 1: 7; 2: 5; 3: 3, 6
function call 1: 1; 2: 5, 7-10; 3: 1
function number 3: 3
functions 1: 1, 5-7; 2: 3-5, 8-9; 3: 1, 3, 9; 4:
1-2, 8, 11, 13-14

H

hexadecimal format 3: 4, 10
HLLAPI 1: 1-8; 2: 1, 3-12; 3: 1-4, 8-11
host session 1: 3-4; 2: 1

K

keystrokes 1: 3-7

L

link 2: 11-12; 4: 3, 8, 10
Local Environment Functions 1: 7; 2: 3; 3:
2-3, 5
logical unit 4: 1, 9, 11-12
LU 2: 13-14;

O

Operator Information Area 2: 2; 4: 14

P

Presentation Space 1: 4-5, 7; 2: 1, 3-4, 6; 3:
3, 7
presentation space 1: 4-6; 2: 1, 11, 13-14; 3:
2

R

raw data stream mode 1: 8
return code 2: 9-11; 3: 8

S

sessions 4: 7

Signals 4: 11

SNA 4: 1-4, 8, 10

Storage Manager function 2: 5

T

tutorial editing mode 3: 9

U

UNIX System 1: 7; 2: 7, 12-13; 3: 3, 9, 11

W

WS Ctrl 2: 1, 10

