Macintosh® Zorro
Programmer's Guide

Working Draft #3
Writer: Bill Harris

September 14, 1988

Apple Confidential

APPLE COMPUTER, INC.

Contents

Figures and tables

Preface

About this document

What you should know

This document is intended for programmers who want to write applications for the Apple® 3270 Application Programming Interface and for developers who want to develop drivers that support such applications.

The document assumes that you have extensive development experience with the Apple Macintosh® computer, or that you plan to learn about the computer from other documents (such as those listed in "Suggested Reading" later in this preface). You should also know how to work with the IBM presentation space, and how to use the C programming language.

In summary, this document assumes that you are an experienced IBM 3270-type application programmer who wants to learn how to program the same type of applications using the Apple 3270 Application Programming Interface.

What this document includes

This document is divided into these chapters and appendixes:

- Chapter 1, "The Apple 3270 API Architecture," explains the logical architecture of the Apple 3270 Application Programming Interface (API), briefly describes the presentation space, and shows how to issue an API call and how to build an API application.
- Chapter 2, "3270 API Application Guidelines," provides some guidelines about how to write terminal-emulation and file-transfer applications, and lists DFTerm, which is a sample API terminal-emulation program.
- Chapter 3, "API Service Requests," provides a complete description of the API routines, and itemizes and defines each parameter for each verb.
- Chapter 4, "Apple 3270 API Device Drivers," contains information that you should consider if you are developing a 3270 API driver.
- Appendix A, "Error Codes," lists the 3270 API error codes.
- Appendix B, "Scan Codes," lists the 3270 API control keys and their scan codes.

- Appendix C, "Request Codes," lists the 3270 API request codes and some alternate C definitions for convenience.

The document also includes a glossary, a bibliography, and an index.

Suggested reading

The *Apple Technical Library*, published by Addison-Wesley, is a set of technical books from Apple Computer, Inc., that explains the hardware and software of the Macintosh family of computers. The descriptions that follow may help you decide which of the books will be most useful to you.

Macintosh computer documents

The *Apple Technical Library*, published by Addison-Wesley, is a set of technical books from Apple Computer, Inc., that explains the hardware and software of the Macintosh family of computers. The descriptions that follow may help you decide which of the books will be most useful to you.

- *Inside Macintosh*, Volumes I, II, and III. These books cover the Macintosh Toolbox and Macintosh Operating System for the original 64K Macintosh ROM, along with user interface guidelines and hardware information.
- *Inside Macintosh*, Volume IV. This guide covers only what is new for the Macintosh Plus and Macintosh 512K enhanced computers (128K ROM).
- *Inside Macintosh*, Volume V. This guide covers what is different about the Macintosh SE and Macintosh II computers (256K ROM).
- *Technical Introduction to the Macintosh Family*. An introduction to the hardware and software design of the Macintosh family, this book serves as a starting point for the *Apple Technical Library*. It is oriented primarily toward the Macintosh Plus, Macintosh SE, and Macintosh II computers, but it also touches on earlier versions of the Macintosh where these differ from the Macintosh Plus.

- *Programmer's Introduction to the Macintosh Family.* This book provides an overview of software development for the Macintosh family of computers. The book focuses on the differences between event-driven programming and more traditional programming techniques. It covers such topics as QuickDraw graphics, screen displays, and the Macintosh User Interface Toolbox.
- *Human Interface Guidelines: The Apple Desktop Interface.* This guide describes the Apple user interface for anyone who wants to develop applications.
- *Inside Macintosh X-Ref.* This reference contains comprehensive indexes, routine lists, and a glossary for *Inside Macintosh* and other Macintosh programming books.

Other books that may be helpful include the following, which are available from the Apple Programmer's and Developer's Association (APDA™).

- *Macintosh Programmer's Workshop Reference:* A guide to the Macintosh Programmer's Workshop (MPW™) Shell and utilities, including the resource editor (ResEdit), the resource compiler (Rez), the Linker, the Make facility, and the debugger.
- *MPW C Reference.* This manual describes how to write C programs in MPW C.

Documents related to 3270 API

The following publications are useful for anyone writing 3270 API applications:

- *IBM 3174/3274 Control Unit to Device Product Attachment Information* (Oct 16, 1986).
- *IBM 3270 Information Display System Character Set Reference* (GA27-2837).
- *IBM 3270 High Level Language Application Program Interface Programming Guide* (59X9474).

Conventions used in this document

In this document, terms are printed in **boldface** when they are introduced. These terms are also included in the glossary.

Terms that are taken from the C programming language are shown in `Courier`.

For an explanation of the conventions used to document each API call, see the beginning of Chapter 3.

Chapter 1

The Apple 3270 API Architecture

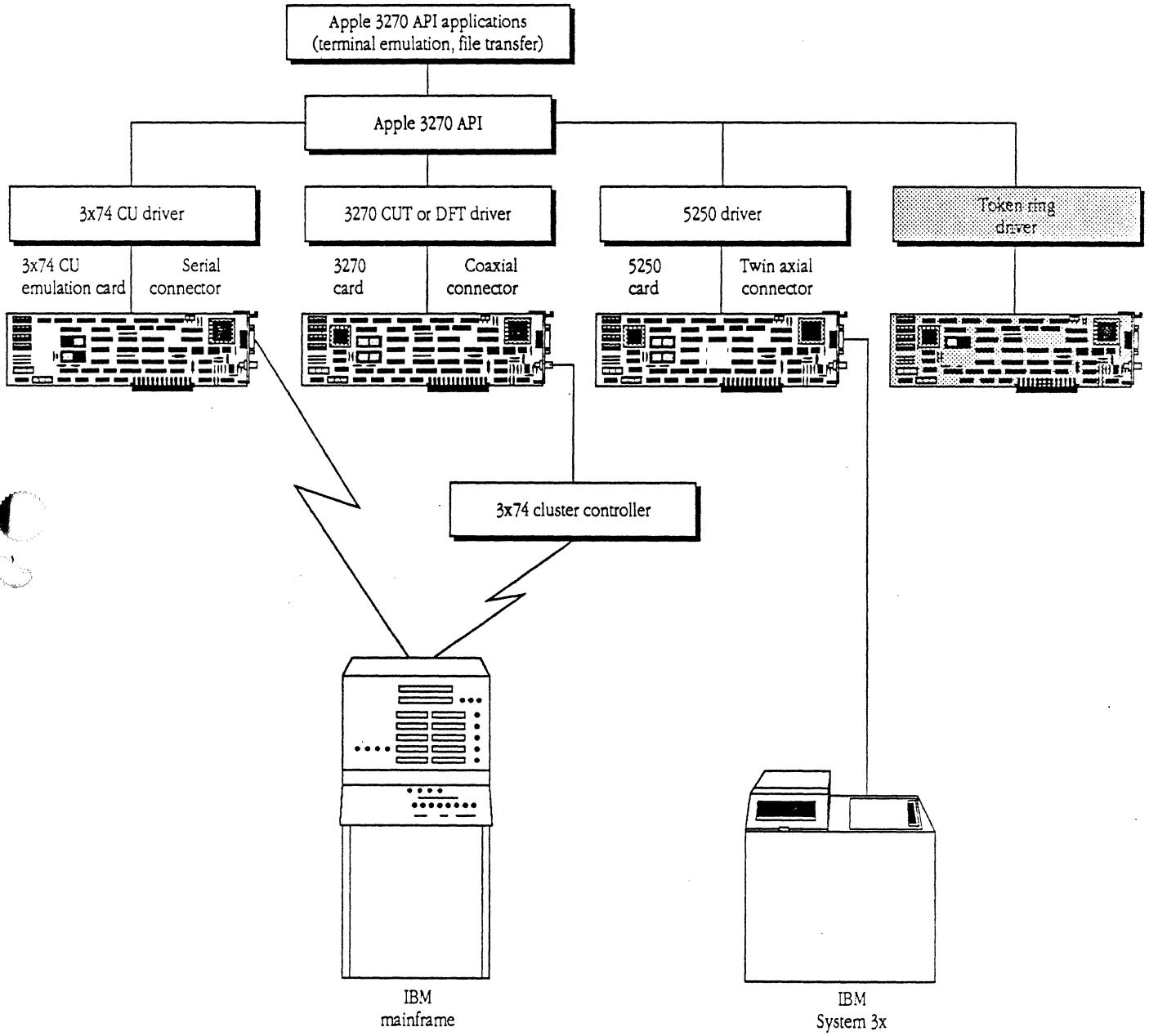
A Macintosh can communicate to an IBM mainframe or System 3x by using the following:

- an Apple 3270 Application Programming Interface application
- the Apple 3270 API
- a 3x74, 3270, or 5250 device driver
- a NuBus™ API card, such as the Apple 3270 API card

Figure 1-1 illustrates the logical structure of the Apple 3270 architecture.

(figure is on next page)

Figure 1-1
Logical 3270 API Architecture


Apple 3270 API applications
(terminal emulation, file transfer)
Apple 3270 API
3x74 CU driver
3270 CUT or DFT driver
5250 driver
Token ring
driver
3x74 CU
emulation card
Serial
connector
3270
card
Coaxial
connector
5250
card
Twin axial
connector
3x74 cluster controller
IBM
mainframe
IBM
System 3x

As the figure indicates, Apple Computer or third-party vendors could add enhancements that support other 3270-type devices.

Your Apple 3270 API application will typically be one of the following:

- a 3270 terminal-emulation application with file-transfer capability
- an implementation of IBM's Enhanced Connectivity Facilities

The Apple 3270 API allows you to write your application with some degree of device independence. As shown in Figure 1-1, the Apple 3270 API separates a 3270 API application from the underlying device driver and API card. Thus, the following are possible:

- An Apple 3270 API application can be used with any device driver that adheres to the API call specification.
- New drivers can be installed without change to the application.
- New cards can be designed that take advantage of existing drivers.

A 3x74, 3270, or 5250 driver (referred to as the *driver* in this document) is a system or application resource that contains object code to be downloaded to an API card.

Many drivers support CUT or DFT-CU devices. **CUT** stands for *Control Unit Technology*. Devices that fit this class are 328x printers or compatible printers, and the classic "dumb" terminals such as the 3178 and the 3278. With this type of technology, the burden of the processing is shifted to the control unit, and the device is limited to providing physical display of the data and input to the controller. CUT devices can support only one logical terminal per device.

DFT stands for *Distributed Function Technology*. As the name implies,devices of this type are used in networks that distribute the processing among the members of a network. Devices that fit this class are the 3270 PC and other PC-based workstations. With this type of technology, the burden of the processing is shared between the host and the terminal. As a result, DFT devices can support up to five separate logical terminals with one or more hosts at once.

The driver also provides the NuBus interface between the Macintosh application and the card. When a user restarts the Macintosh, object code from the driver is loaded into the system memory of the Macintosh. Many different drivers can reside in the system heap and be available to the application.

The driver is usually a system resource installed by the user into the System Folder, or can be a temporary driver available only for the life of the application. See Chapter 4 for more information on how to construct a driver.

A 3270 API-type card, of which the Apple 3270 API card is an example, supplies the hardware support and physical connection to the host. Note that Figure 1-1 illustrates the connections as existing on separate cards; while this structure is logically true, the functions can be combined on one physical card. For the specifications of a particular 3270 API card, see its hardware manual. However, in most circumstances, you won't have to worry about the particular hardware being used; in fact, that's the concern of the API.

What the 3270 API supports

The Apple 3270 API supports the following:

- establish and terminate connections to a host
- position the cursor in the 3270 Presentation Space
- examine and change fields in the 3270 Presentation Space
- send 3270 keystrokes to the host
- wait for the host to update the 3270 screen
- send and receive raw data to and from the host
- host-initiated printing, including SCS and DSC
- maintain multiple host sessions
- the Macintosh user interface
- the ability to suspend a 3270 application and switch to another Macintosh application (when running under MultiFinder)

The Apple 3270 API, at the time of publication of this guide, does not support the following:

- explicit partitions
- double byte coded character sets (DBCS), such as that for Kanjii
- entry assist
- programmed symbol sets in CUT mode
- printer emulation support in CUT mode
- IPDS

IBM and Apple display buffers

The Apple 3270 API defines four buffers; each buffer, if it is used, must be the same size as the screen being emulated. The buffers are as follows:

- **Presentation space (PS),** which contains the displayed data and the field attribute bytes
- **Extended attributes buffer (EAB),** which contains the extended field attributes and the character attributes
- **Display attributes buffer (DAB),** which contains a composite definition for each displayed character that indicates the highlighting, color, and intensity of the character
- **Extended display attributes buffer (DABE),** which contains a composite definition for each displayed character that indicates the character set, the Modified Data Tag (MDT), and some format details

The PS and the EAB are buffers defined by IBM; the DAB and the DABE are buffers defined by Apple Computer. Each of the buffers is described in more detail in the following sections.

The IBM attribute buffers

The PS and the EAB are buffers defined by IBM. The following sections describe these buffers in more detail.

Presentation space

Regardless of the physical connection used, 3270 API applications copy data to and from a logical equivalent of the 3270 screen. This logical equivalent of the screen is called the **presentation space** (PS), and is the main buffer that a 3x74 CU writes or that a DFT terminal maintains. The presentation space contains the data and the field attribute bytes, and is illustrated in Figure 1-2.

(figure is on next page)

Figure 1-2
A view of the presentation space

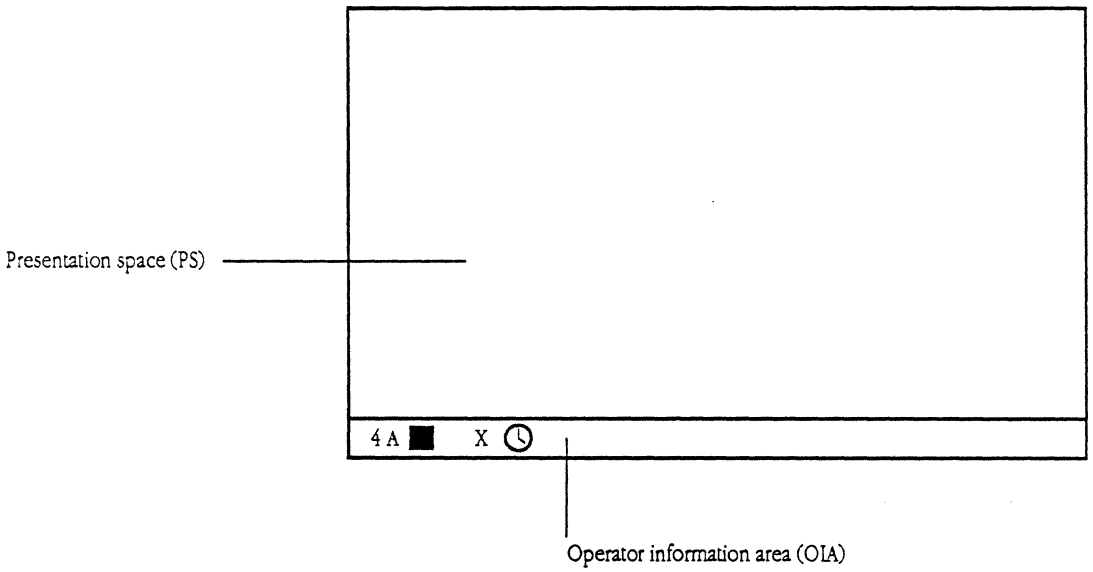
The presentation space is considered to be **unformatted** if it does not contain any fields, or considered to be **formatted** if it does contain fields.

The Operation Information Area (OIA), on row 25 at the bottom of the screen in Figure 1-2, is a status line. For example, if an **input-inhibited condition** exists, it indicates that keyboard input is not allowed. An X followed by a string of symbols appears in the OIA to indicate this condition.

Extended attribute buffer

The **extended attribute buffer** (EAB) is the secondary buffer to which the 3x74 control unit writes if the 3270 device is able to support extended attributes. In this buffer, each field starts with an extended field attribute byte that has additional information about how the field is to be displayed. Also, each individual data byte has a character attribute byte that may specify whether the the extended field attribute is to be overridden for that byte.

Many applications, including most terminal emulation packages, don't need to use this buffer. Instead, you can use the Apple-defined display attribute buffer, as described in the next section.


Presentation space (PS)
4 A
X
Operator information area (OIA)

The Apple attribute buffers

The Apple attribute buffers allow your application to read the relevant information for each byte from its own attribute byte, rather than decoding the information from several scattered attribute bytes. The two Apple attribute buffers are the Display Attribute Buffer (DAB) and the Extended Display Attribute Buffer (DABE). The following sections describe these buffers in more detail.

Display attribute buffer

The **display attribute buffer** (DAB) is a composite buffer derived from the PS and the EAB, and is intended to support a terminal-emulation application. For each byte in the PS passed to the application, there is a corresponding byte in the DAB This **basic DAB byte** provides the highlighting and color information associated with the PS byte.

Most applications, especially text-only terminal emulation application, require only the use of the DAB byte. The format of the basic DAB byte is shown in Figure 1-3.

(figure is on next page)

Figure 1-3
Basic DAB byte format

❖ *Note:* The values for the color bits in the basic DAB byte have been assigned by IBM. However, there is nothing to prevent an application from assigning other color values.

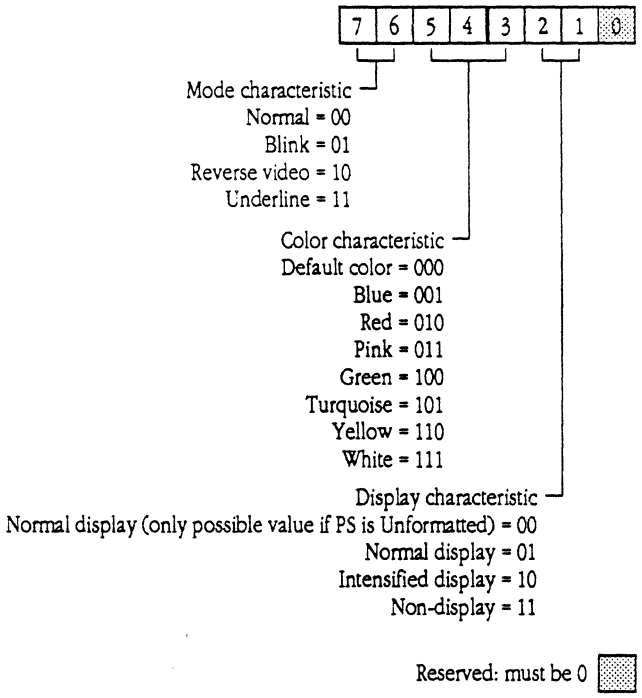


Fig 1-3 1-2-B (L02) Basic DAB byte format
Zoro

Apple Computer, Inc.
Deborah Dennis
Illustrator 88

Extended display attribute buffer

Applications that support APL or programmed symbol sets, or applications that need detailed attribute information, use the **display attribute buffer extended byte** (DABE byte) in addition to the basic DAB byte. When combined with the corresponding byte in the DAB, the DABE byte supplies the rest of the information to fully describe the highlighting, color, attribute, and symbol set information associated with the PS byte.

The DABE byte immediately follows each basic DAB byte in the destination buffer. The format of the DABE byte is shown in Figure 1-4.

(figure is on next page)

Figure 1-4
Extended DAB byte

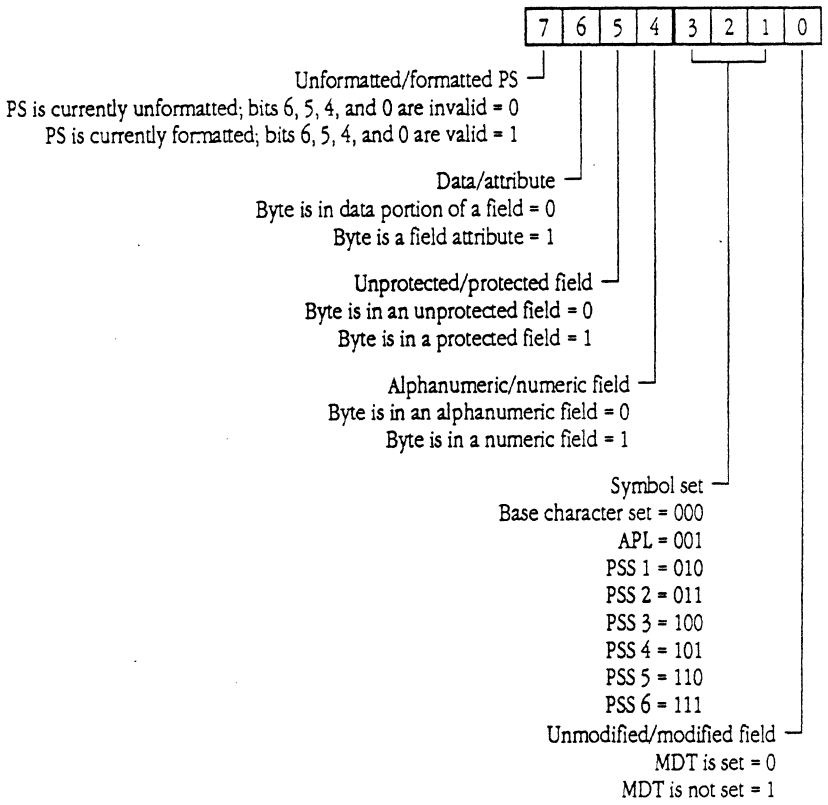


Fig. 1-4 1-3-B (L03) Extended DAB byte
Zoro

Apple Computer, Inc.
Deborah Dennis
Illustrator 88

Using the Apple 3270 API

This section describes how you use the Apple 3270 API to build an application.

The Apple 3270 API request block

The Apple 3270 API request block stores information about the PS and the session. The block either can be nonrelocatable memory in the application's stack or global area, or can be a block of memory that you obtain from the Memory Manager and lock down.

Important Some drivers act upon the full 32 bits of an address. If your application passes a 24-bit address, the application must clear to 0 the high-order 8 bits of the pointer to the request block. This same rule applies to all pointers passed in API requests.

The following type definition shows the structure of the request block in C:

```
typedef struct api_req_fmt
{
    Handle api_vars;
    LONG q_link;
    BYTE req_code;
    AddrBlock net_addr;
    BYTE conn_id;
    BYTE port_id;
    BYTE ps_id;
    WORD result;
    LONG ref_con;
    ProcPtr io_compl;
    union
    {
    OPEN_HOST_CONNECTION        open_host_connection;
    CLOSE_HOST_CONNECTION       close_host_connection;
    GET_HOST_CONNECTION_INFO    get_host_connection_info;
    CONNECT_TO_PS               connect_to_ps;
    DISCONNECT_FROM_PS          disconnect_from_ps;
    SEND_KEYS                   send_keys;
    COPY_TO_PS                  copy_to_ps;
    COPY_FROM_BUFFER            copy_from_buffer;
```

```
        COPY_TO_FIELD               copy_to_field;
        COPY_FROM_FIELD             copy_from_field;
        COPY_OIA                    copy_oia;
        SEARCH_STRING               search_string;
        FIND_FIELD                  find_field;
        GET_UPDATE                  get_update;
        GET_CURSOR                  get_cursor;
        SET_CURSOR                  set_cursor;
        SET_COLOR_SUPPORT           set_color_support;
        SEND_PASSTHRU_DATA          send_passthru_data;
        GET_PASSTHRU_DATA           get_passthru_data;
        POST_PASSTHRU_REPLY         post_passthru_reply;
        DO_SPECIAL_FUNC             do_special_func;
        ACTIVATE_PRT_SESS           activate_prt_sess;
        DEACTIVATE_PRT_SESS         deactivate_prt_sess;
        GET_DSC_PRT_DATA            get_dsc_prt_data;
        GET_LU1_PRT_DATA            get_lu1_prt_data;
        POST_PRT_REPLY              post_prt_reply;
        SEND_PRT_CONTROL            send_prt_control;
        CHECK_SESSION_BIND          check_session_bind;

    } req;

} API_REQ;
```

The definition includes the following:

- a header with the request parameters that must accompany every call
- a union of structures, each of which specifies the parameter values for a call

❖ *Note:* For the parameter values, you can use the exact wording, or the shorter names shown in the section "C Interface and the API Routines" in this chapter, or names that you create.

The parameters in the API request block are as follows (for many 3270 API calls, your application must fill in the values for those parameters shown in **boldface**):

`api_vars` This parameter is the handle returned by the Init_3270_API call. All other API calls should include this value.

`q_link` This parameter is set by drivers that support the queuing of API requests.

`req_code`	This parameter is the request code associated with an API call (set automatically by the API interface routines). The driver examines this field to determine the type of request received from the interface routines.
`net_addr`	This parameter specifies an AppleTalk internet address. Set the aNode field within the address block to 0 if the request is to be processed locally. (At the time of publication of this guide, this parameter was ignored.)
`conn_id`	This parameter identifies the driver or connection method as returned by the Open_Host_Connection call. All other API calls referring to the same connection must include this value.
`port_id`	This parameter indicates the logical address of a physcial device; for example, it can indicate a slot or serial port assigned to a particular session. Data transmitted into and out of a presentation space is routed through the port or slot assigned this ID. All API calls should include this ID except Open_Host_Connection.
`ps_id`	This parameter returns the presentation space identification from the Connect_To_PS call, or a printer session ID from an Activate_Prt_Sess call. All API calls should include this ID except Open_Host_Connection and Close_Host_Connection.
`result`	This parameter is set by the driver. Your application must examine this parameter to verify that a call was processed successfully.
`ref_con`	This parameter is for optional use by the application.

`io_compl` This parameter is a pointer to a routine called by a driver that is capable of receiving an interrupt when an API request completes. The application defines this I/O completion routine; see "Using a Custom I/O Completion Routine" in this chapter. Set this parameter to 0 if you're not going to use a custom I/O completion routine.

The `conn_id`, `port_id`, and `ps_id` parameters work together as illustrated in Figure 1-5.

(figure is on next page)

Figure 1-5
conn_id, port_id, ps_id

Specifying API configuration information

Your 3270 API application must know what slots and what type of driver are being used, along with other information about the driver. You specify that information in a data structure and then supply a pointer to that data structure in the Open_Host_Connection call.

The following sections define the structures and the values for the configuration information for the DFT and CUT drivers produced by Apple.

- *Note:* The values for drivers developed by third parties should be listed in their documentation.

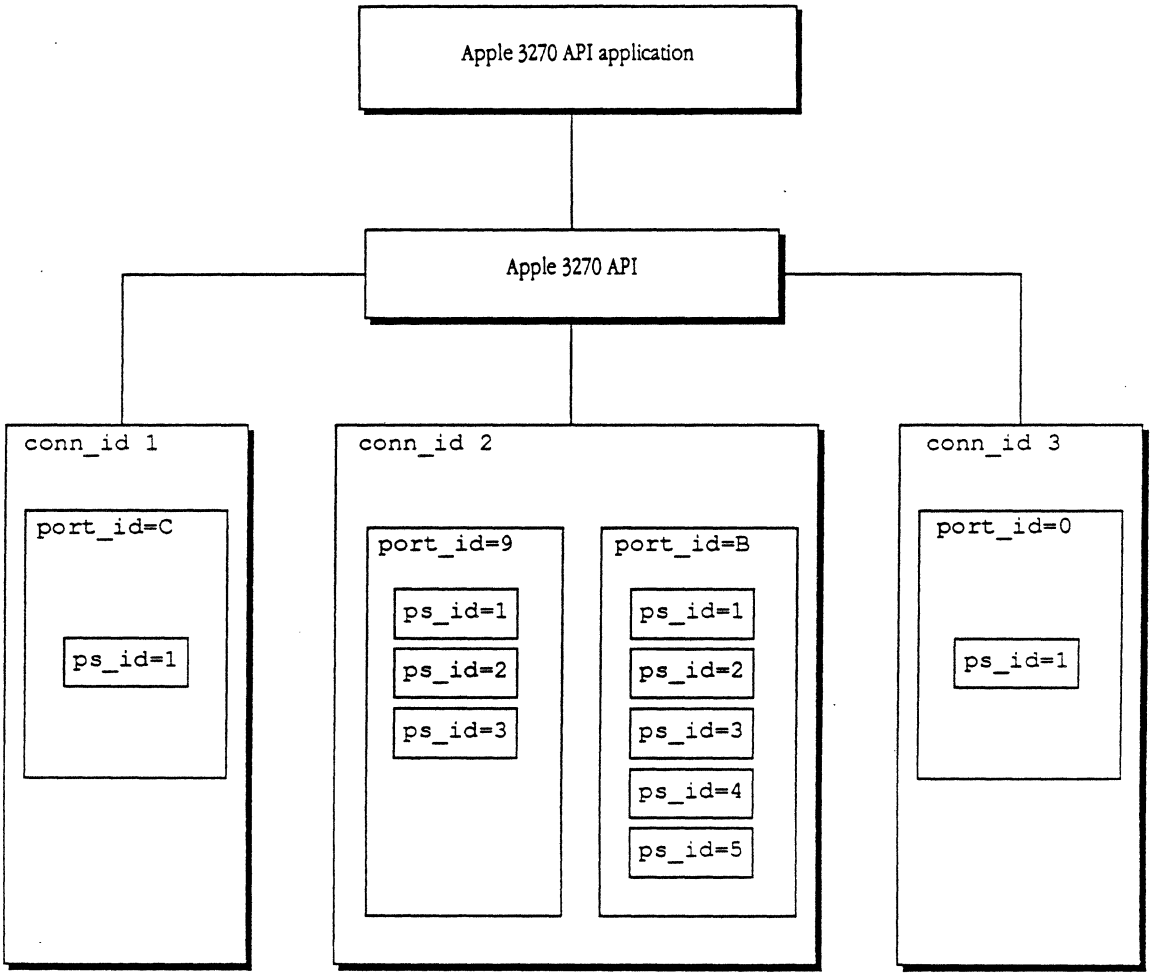


Fig. 1-5 1-6-Comp (L06) conno_id, port_id, ps_id
Zoro

Apple Computer, Inc.
Deborah Dennis
Illustrator 88

Apple 3270 DFT configuration

The configuration information in the Apple 3270 DFT driver supplies the following:

- slot numbers of the slots controlled by the driver, and the status of the card in the slot
- information on the LU types and presentation space characteristics supported by each active slot

The data structure that supplies this information for the Apple DFT driver is as follows:

```
typedef struct apple_dft_config_info

{
WORD slot_map;                              /* passed */
WORD slot_status[NUM_PORTS];                /* returned */
struct
{
  BYTE lu_type[MAX_DFT_SESS];;              /* passed */
  BYTE ps_status[MAX_DFT_SESS];             /* returned */
} slot_info[NUM_PORTS];
} APPLE_DFT_CONFIG_INFO;
```

The parameters for this data structure are described in the following paragraphs.

slot_map: This parameter is a bitmap specifying the slots the driver will control. Bit 0 corresponds to slot 0, bit 1 to slot 1, and so on. For each bit set, the driver downloads code to the card. Subsequent API requests are passed to the slot or card as directed by a request's `port_id`.

slot_status: This parameter returns a value indicating the status of the slot. If the card is brought up successfully, `slot_status` is equal to `NO_ERR` (0x0000); if the card is not brought up successfully, the driver returns an error code indicating the reason for failure. A special value of 0xFFFF indicates that the slot was not specified in `slot_map`. If your application attempts to send a call to a card with a failed or unused slot status, the driver rejects the attempt. Valid slots for the Macintosh II are 9 through E.

slot_info: This parameter is an array with each element corresponding to a slot. Active slots are identified in `slot_map`. Each slot can support up to five PS/SNA sessions.

Important An application should check the `result` field in the API request block before checking returned values in the `slot_info` array. Returned values are invalid if `result` is nonzero.

Within the `slot_info` array are the following fields:

`lu_type` This field is an array whose elements correspond to the five underlying logical terminals. Valid values for the elements are as follows:

`ADFT_LU_TYPE_1`	LU type 1 (printer)
`ADFT_LU_TYPE_2`	LU type 2 (display)
`ADFT_LU_TYPE_3`	LU type 3 (printer)

`ps_status` This array returns a value indicating what presentation spaces the DFT software will support, as follows:

`ADFT_PS_SUPP`	This value (1) indicates that the PS is supported.
`ADFT_PS_UNSUPP`	This value (0) indicates that the PS is unsupported.

If your application attempts to access an unsupported PS, the driver returns `PS_UNSUPP_ERR`.

Apple 3270 CUT configuration

The configuration information in the Apple 3270 CUT driver supplies the satus and the slot numbers of each slot controlled by the driver.

The data structure that supplies this information for the Apple CUT driver is as follows:

```
typedef struct apple_cut_config_info

{

    WORD slot_map;

    WORD slot_status[NUM_PORTS];

    BYTE term_id[NUM_PORTS][5];

} APPLE_CUT_CONFIG_INFO;
```

The parameters for this data structure are described in the following paragraphs.

slot_map: This parameter is a bitmap specifying the slots the driver will control. Bit 0 corresponds to slot 0, bit 1 to slot 1, and so on. For each bit set, the driver downloads code to the card. Subsequent API requests are passed to the slot or card as directed by a request's `port_id`.

slot_status: This parameter returns an array, with each element in the array indicating the status of a card. If a card has been brought up successfully, its corresponding element is equal to `NO_ERR` (0x0000); if the card is not brought up successfully, the driver returns an error code indicating the reason for failure. A special value of 0xFFFF indicates that the slot was not specified in `slot_map`. If your application attempts to send a call to a card with a failed or unused slot status, the driver rejects the attempt. Valid slots for the Macintosh II are 9 through E.

term_id: This parameter is a 5-element array. Each of the 16 possible slots where a card can reside has an associated `term_id` array. Byte 0 of `term_id` is sent by the card to the control unit when the control unit issues a Read Terminal ID command. (Keyboard type and PS size information are present in this byte.)

Bytes 1 through 4 of `term_id` are returned in response to a Read Extended Terminal ID command. (The driver ignores these bytes if byte 0 indicates that the control unit should not issue a Read Extended Terminal ID.) Refer to the *IBM 3174/3274 Control Unit to Device Product Attachment Information* specification for a description of the terminal ID byte and the extended terminal ID bytes.

The driver will extract keyboard type and PS size information from `term_id`. The driver returns errors for invalid values.

Checking for a completed request

You can make most API calls either synchronously or asynchronously by setting the `asyncFlag` parameter in the call to `ASYNC` or `SYNC`.

If you set `asyncFlag` to `SYNC`, your application doesn't regain control until the request is completed.

Important Be aware that, if you set asyncFlag to SYNC, your application can't issue a WaitNextEvent call. That call supports cooperative processing in the MultiFinder environment; thus, issuing a synchronous call also prevents all other applications from executing until the request is completed.

If you set `asyncFlag` to `ASYNC`, your application regains control immediately with the result of the operation set to zero if the API code accepted the call or nonzero if the code did not accept the call. Your application can then proceed with other processing if the operation result is 0 or it can handle the error if the operation result is nonzero.

❖ *Note:* When an API error occurs, the API also sets the `result` field in the request block to the same value as the error. Thus, your application could check `result` later instead of immediately checking the result of the operation. The disadvantage of using this technique is that the application doesn't immediately detect interface routine errors.

Before forwarding the request to the driver, the API code sets the `result` field in the API request block to `RSP_PENDING`. Your application can then periodically check `result` to see if it has changed; when it has, the request has been completed.

Issuing a 3270 API call

After you have allocated memory for the API request block, take the following steps each time you make an API call:

1. Fill in the required fields, if any, in the header portion of the API request block.
2. Provide values for the parameters associated with the particular API request.
3. Make the API call using the following format:

   API_Call_Name `(&req_blk, asyncFlag);`

   Use the call names as listed in this guide for the *API_Call_Name*. If the call needs to access the request block, include the `&req.blk` parameter. You can set the `asyncFlag` parameter to `SYNC` or `ASYNC`.

4. If you set the `asyncFlag` parameter to `ASYNC`, periodically check the `result` field in the API request block for a change to determine when the request actually completes. See "Checking for a Completed Request" in this chapter for more information.
5. After you issue the request, do not modify the contents of the API request block until a response is returned.

The following code fragment shows a typical API calll:

```
API_REQ     api_blk;
BYTE        keys_buf[2];
BYTE        saved_conn_id;
BYTE        saved_ps_id;
WORD        err;

api_blk.conn_id = saved_conn_id;
api_blk.port_id = 0x0E;
api_blk.ps_id = saved_ps_id;

api_blk.req.send_keys.num_keys_to_send = 1;
api_blk.req.send_keys.keys_bufp = &keys_buf;

keys_buf[0] = NO_KEY_MODS;
keys_buf[1] = 0x72;
err = Send_Keys (&api_blk, ASYNC);

if (err)
{
          . . . API glue rejected the call . . .
}
else
{
          {Check result code periodically.}

          while (api_blk.result == RSP_PENDING)
          {
                        ...attend to other matters like
                           servicing the event loop ...
          }
          {Call completed.}

          if (api_blk.result == NO_ERROR)
          {             ...call succeeded...
          }
          else
          {             ...call failed...
```

```
        }
}
```

Building a 3270 API application

Using the API calls, your application can pass data between the Apple 3270 application and the 3270 presentation services of the device driver you are using. The basic API calls that establish and terminate an API application are shown in Figure 1-6.

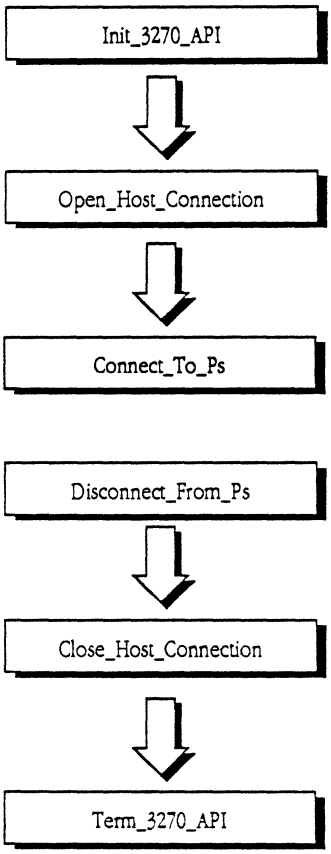
(figure is on next page)

Figure 1-6
The basic API calls

In more detail, to establish a session with the host, your application must take the following steps:

1. Include the API header file.
2. Initialize the Apple 3270 API by using an Init_3270_API call, which returns a handle that you must save and use in subsequent API calls.
3. Allocate memory for the API request block. Either reserve nonrelocatable memory in the application's stack or global area, or obtain a block of memory from the Memory Manager and lock it down.

   The structure and function of the request block is shown in "The Apple 3270 API Request Block," earlier in this chapter.


Init_3270_API
Open_Host_Connection
Connect_To_Ps
Disconnect_From_Ps
Close_Host_Connection
Term_3270_API

4. Allocate memory for the configuration block. The configuration block tells the application what values to use for the driver.
5. Specify the API configuration information for the appropriate driver by defining the appropriate data structure. See the section "Specifying API Configuration Information" earlier in this chapter.
6. Assign the address of the configuration block to a pointer.
7. Make the Open_Host_Connection API call. This call downloads configuration and routing routines from the device driver to the API card, and requests that a physical connection be established with the host. The host specifies the characteristics of the presentation spaces for each session.
8. Save the connection ID returned by the Open_Host_Connection call and place it in the appropriate API request block for use by subsequent API calls.
9. Fill in the required fields in the header portion of the API request block; in this case, supply the `ps_id` that was returned by the Open_Host_Connection call.
10. Make the Connect_To_PS API call, which reserves a presentation space for a particular host session by either specifying or requesting a presentation space ID. Specify the call as either asynchronous or synchronous, and use the technique described in "Checking for a Completed Request" in this chapter to determine when the API request actually completes.

❖ *Note:* Alternatively, you could set up an I/O completion routine to post an event in the event queue when the request has been completed. See "Using I/O Completion Routines" in this chapter for more information.

11. Save the PS ID returned by the call; any other API call using the same presentation space needs the PS ID.
12. Continue to make API calls, filling in request header values and allocating space for the call parameter blocks when necessary.
13. If you need to activate a print session, take the steps listed in "Issuing a Print Request" earlier in this chapter.
14. If you need to activate more than one session, see "Multiple outstanding API requests" later in this chapter.
15. To terminate your application, make the Disconnect_From_PS call, which deallocates the session ID and thus breaks the logical connection to a presentation space.

16. Make the Close_Host_Connection API call, which terminates a connection by sending an "LU offline" message to the host and stopping all tasks running on the Apple 3270 API card.
17. Make a Term_3270_API call, and supply the handle that was returned by the Init_3270_API call. Doing this shuts down the API.

A skeleton application that issues the basic API calls is shown in Chapter 2.

C interface and the API routines

The `api3270.h` header file contains several bit definitions. If you are not familiar with C, here are two ways you can use these definitions:

To set a bit or a group of bits, you can use the bitwise inclusive OR operator (|). For example, the following constants exist for the Get_Update call:

```
#define GU_IGNORE_PS      0x0001
#define GU_IGNORE_CURSOR  0x0002
#define GU_IGNORE_OIA     0x0004
```

To set 3 bits at once, you could use a statement like this:

```
api_blk.req.get_update.modifiers = GU_IGNORE_PS |
GU_IGNORE_CURSOR | GU_IGNORE_OIA;
```

To determine if a bit is set, use the bitwise AND operator (&), as shown here:

```
#define GI_PSS 0x00000008
if (api_blk.getinfo.dev_feats_supp & GI_PSS)
{          PSS is supported
}
else
{          PSS is unsupported
}
```

The API header file also provides alternate definitions that allow you to use fewer characters when you access a field within a particular request. For example, using the alternate definitions, the statement:

```
blk.openhc.open_type = OC_WARM;
```

is equivalent to

```
blk.req.open_host_connection.open_type = OC_WARM;
```

You can also add your own definitions to shorten other names.

The alternate definitions are as follows:

```
#define openhc      req.open_host_connection
#define closehc     req.close_host_connection
#define getinfo     req.get_host_connection_info
#define connps      req.connect_to_ps
#define discps      req.disconnect_from_ps
#define sendkey     req.send_keys
#define cpytops     req.copy_to_ps
#define cpyfbuf     req.copy_from_buffer
#define cpytfld     req.copy_to_field
#define cpyffld     req.copy_from_field
#define cpyoia      req.copy_oia
#define srchstr     req.search_string
#define findfld     req.find_field
#define getupd      req.get_update
#define getcurs     req.get_cursor
#define setcurs     req.set_cursor
#define setcolor    req.set_color_support
#define sndpdata    req.send_passthru_data
#define getpdata    req.get_passthru_data
#define postpass    req.post_passthru_reply
#define spec        req.do_special_func
#define actprt      req.activate_prt_sess
#define dactprt     req.deactivate_prt_sess
#define getdsc      req.get_dsc_prt_data
#define getlul      req.get_lul_prt_data
#define postprt     req.post_prt_reply
#define sndpctl     req.send_prt_control
#define chkbind     req.check_session_bind
```

The API calls and API support

The API calls have designed to support various 3270 features. The following sections introduce you to some of the features of the API and indicates what calls support what features.

About sessions

There are 3 session types supported by the API: LU 1, 2, and 3. LUs 1 and 3 are printer LUs, while LU 2 is display-oriented. Besides supporting PS-oriented data, LU 2 also supports higher level non-PS data destined for applications such as the IND$FILE file transfer program, SRPI, and so on.

EBCDIC, DBC, ASCII, and scan codes

Communication in the 3270 world occurs in several "languages." For the application to succeed, it usually has to translate from one language to another, as discussed in the following sections.

EBCDIC and DBC

EBCDIC is the language of the IBM mainframe world. If your application is using a DFT or CU driver, the application must supply translation tables that perform the translation from EBCDIC-to-DBC and from DBC-to-EBCDIC when the host and the presentation space communicate.

Your application points to the translation tables in the Connect_To_PS call. The format of the tables, and more details about how to use them, is presented in the description of the Connect_To_PS call in Chapter 3.

3270 Device Buffer Code and ASCII format

All connection methods maintain an image of the presentation space in 3270 device buffer code (DBC) format. This allows your application to issue API calls in the same manner regardless of the underlying connection method.

All calls that interact with the presentation space pass or receive data in DBC format. An application is responsible for mapping device buffer code to the appropriate format for display; usually the format is ASCII unless APL/Text and programmed symbol sets (PSS) are supported. Your application points to the translation tables in the Connect_To_PS call. The format of the tables, and more details about how to use them, is presented in the description of the Connect_To_PS call in Chapter 3.

You can also use the various `NO_TRANS` constants in the `modifiers` parameter of appropriate calls to specify that the call should not perform any translation.

To copy data to the PS, an application should map the data to DBC format. The API calls that map the data provide pointer parameters that point to translation tables that you define. Sample DBC-to-ASCII and ASCII-to-DBC tables have been provided in the sample application in Chapter 2, and can be modified to suit the application.

To distinguish between a normal, APL/Text, or PSS character in the presentation space, an application that supports APL/Text or PSS should examine the DABE for the associated character set value. (If an application doesn't support APL/Text or PSS the application can simply map each DBC value to a displayable ASCII value.) The values are as follows:

0 Indicates the base character set

1 Indicates APL/Text

2–7 Specifies PSS sets 1 through 6

Checking the value of the character in the PS is incorrect because APL/Text and PSS characters occupy the same range of values in the PS as the default character set used for normal display.

For the DBC values of APL/Text characters, refer to the APL Device Buffer Code table in the *IBM 3174/3274 Control Unit to Device Product Attachment Information.*

Dead key and dead key terminator scan codes

On certain keyboards (for example French AZERTY), using the accent characters causes individual accents (such as circumflex, grave, diaresis) to appear on the display, but the cursor does not move. These accent functions are referred to as dead keys. A subsequent character that receives the accent must be keyed next. If the subsequent character is valid, a unique composite character is formed. You use the `descriptor` type in the `*ktab_rec` of the Connect_To_PS call to support the use of dead keys.

See *IBM 3270 Information Display System Character Set Reference* (GA27-2837) for further information.

Color support

The API supports the following color modes:

- No color, with the DAB color bits always set to 000
- Two base colors, without extended colors
- Four base colors, without extended colors
- Two base colors, with extended colors
- Four base colors, with extended colors

Your application originally defines its color supportt in the Connect_To_PS call, and may change the color support while the application is running by issuing a Set_Color_Support call. For more information about how those calls define the color, see the descriptions of those calls in Chapter 3.

Passthrough data and structured field support

When an application connects to a presentation space, it also implicitly connects to its underlying session. Consequently, non-PS data transmitted over the session can be passed through by the API without having to establish a separate session connection.

Such passthrough data is usually destined for a higher-level application function. The most common passthrough data is structured field data, such as for D0 structured fields (for the INDSFILE file transfer method) or APA structured fields (vector graphics support).

The API supports passthrough data by providing the Get_Passthru_Data, Send_Passthru_Data, and Post_Passthru_Reply calls.

For example, the API issues Send_Passthru_Data and Get_Passthru_Data calls to send and receive structured fields containing requestor Server-Requestor Programming Interface (SRPI) data and control information to establish a SRPI connection. Use of SRPI on a session does not prevent an application from issuing concurrent API requests on other sessions.

Printer support

To print using the 3270 API, you need to use either the LU1 or the LU3 print data streams. Both of the data streams use structured fields to accomplish the sending of print data; thus, CUT drivers cannot support printing through the API.

The calls that provide 3270 printer support are as follows:

Activate_Prt_Sess
Deactivate_Prt_Sess
Get_DSC_Prt_Data
Get_LU1_Prt_Data
Post_Prt_Reply
Send_Prt_Control
Check_Session_Bind

Until you begin the print sequence with the Activate_Prt_Session call, attempts by the host to establish contact with the session are rejected with a "device unavailable" error.

Your application would typically issue the calls in the sequence shown in the following pseudocode:

```
Activate_Prt_Sess;          {allocate a session to the application}
Check_Session_Bind;         {wait for host application to establish
                             contact}
if lu_type is equal to LU type 1
    while result is not equal to NO_HOST_SESS_ERR
        Get_LU1_Prt_Data;
        {validate the print data}
        if data_end is equal to GLP_END_REPLY
            Post_Prt_Reply;
```

```
        endwhile;
else                                {lu_type must be LU type 3}
        while result is not equal to NO_HOST_SESS_ERR
                Get_DSC_Prt_Data;
        endwhile;
Deactivate_Prt_Sess;                {deallocate session}
```

Certain LU1 host applications may require a PA1 signal or a PA2 signal from the printer. The Send_Prt_Control call is used for this purpose. The call is not typically part of the data acquisition and reply loop.

Alternate screen size support

If the application is emulating a Model 3, 4, or 5 display, the host program or the operator can change the screen size. The driver notifies the application by returning a `result` of `CHG_TO_DEFAULT_SCR_ERR` or `CHG_TO_ALT_SCR_ERR` to the next request that deals with the affected PS. These requests are: Send_Keys, Copy_To_PS, Copy_From_Buffer, Copy_To_Field, Copy_From_Field, Search_String, Find_Field, Get_Update, Get_Cursor, and Set_Cursor.

If a Get_Update call is outstanding when a screen size change occurs, it completes immediately with a screen size change error. When the application receives notification of a change in screen size, it hould adjust its representation of the PS. However, a change isn't necessary if the terminal emulation is already in the screen size specified by the error.

After performing any necessary changes, the application may re-issue the request if desired.

- *CUT note:* CUT drivers return a notification of a screen size change only if the screen column width changes from 80 to 132 or vice versa. Applications never receive such notifications for Models 3 and 4 because the column width is the same for both alternate and default screen sizes. Thus, the application should assume that the larger alternate screen is always in effect and issue calls accordingly.

SNA considerations

Certain calls and parameters have been defined to address the specific requirements of DFT and CU environments. These calls and parameters have an SNA orientation. Some of these calls and parameters have no meaning in the non-SNA environment; they are ignored or re-interpreted by a non-SNA driver.

An example of this is the `sense_code` parameter passed in the Post_Passthru_Reply call. The driver substitutes an Op-Check sense for a non-zero sense code. Another example is the Check_Session_Bind call. For an SNA attachment, it indicates if the host has bound the session, and if so, also returns the session type. For a non-SNA attachment, the call indicates if the session (logical device) has been selected and received data, and if so, also returns the data type.

Multiple outstanding API requests

Given the hierarchical arrangement of the conn_id, port_id, and ps_id—as illustrated in Figure 1-2 earlier in this chapter—an application may have multiple outstanding API requests. What happens to each different type of multiple request is discussed in the following sections.

❖ *Note:* Most applications won't need to use multiple requests.

Requests to different conn_ids

Since different `conn_ids` are independent of each other, multiple requests to different `conn_ids` may be processed in parallel fashion.

Requests to the same conn_id, different port_ids

These requests are processed independently of each other, and may be processed in a parallel fashion.

Requests to the same port_id, different ps_ids

Requests destined for the same port will, in all cases, be processed in a serial fashion. There is relatively little advantage to stacking requests. If a preceding request completes in error, it might affect the processing of stacked requests. It's far safer for an application to issue just a single request at a time, check `result`, and then issue the next request.

Requests to the same ps_id

In most cases, the driver completes a request before it deals with the next request to the same port. The exceptions to this rule are requests such as Get_Update, Get_Passthru_Data, and Wait_Session_Bind which wait for data or an event in order to complete. Such requests are held by the driver if it cannot satisfy them immediately.

While requests are being held, other requests to the port may be sent by the application. They are processed to completion in the usual fashion or may become held requests. The driver rejects a request with the error `REQ_OUTSTANDING_ERR` if the request's `req_code` and `ps_id` are similar to any of the requests currently held; that is, for a particular presentation space, only one of each kind of request may be held.

Using a custom I/O completion routine

Your application can define a custom I/O completion routine and point to it by using the `io_compl` parameter in the API request block.

An I/O completion routine is executed at the interrupt level; thus, all the guidelines for creating Macintosh interrupt-level routines apply, including the following:

- Make sure that the routine executes quickly.
- Don't make Memory Manager calls, either directly, or indirectly by making Macintosh Toolbox calls that issue such calls.
- Save registers on entry and restore them on exit.
- Call SetUpA5 and RestoreA5 to access the application's globals (although, at the time of this guide's publication, MultiFinder doesn't provide a way to access globals from an interrupt-level routine).

For more information about writing interrupt-level routines for the Macintosh, see the Device Manager chapter and the descriptions of SetUpA5 and Restore A5 in *Inside Macintosh*.

For example, an application-defined completion event could post an event into the event queue when the request has been completed. Thus, instead of periodically examining `result`, the application could wait for the completion event. The drawback of this approach is that events can be discarded(and thus the application may not be notified of the I/O completion), if the application and desk accessories don't handle events quickly enough.

If you expect your application to run under MultiFinder, your I/O completion routine must recognize whether the application is running in the background. If the application isn't currently executing, another application may intercept any posted events.

Chapter 2

3270 API Application Guidelines

Besides the 3270 issues that you will have to deal with as you develop your application, you'll need to know how to program in the Macintosh environment. As usual, you should consider volumes I through V of *Inside Macintosh* to be your major source of information on how to write Macintosh applications.

This chapter presents some general guidelines and specific tips that should help you develop 3270 applications.

Writing 3270 applications

As you begin to write your 3270 API application, one important thing to remember is that Macintosh applications are normally **event-driven;** such applications center around a main event loop that waits for the user to do something. When the user causes an event, the main event loop takes action to service the event, and then returns to waiting for something to happen.

If you're not familiar with event-driven programming, read the Event Manager chapter in *Inside Macintosh*.

Writing a 3270 API terminal emulation application

Apple has designed the DAB so that, for many terminal emulation applications, you only need to use it and the PS, thus sparing your application from dealing with the EAB and the DABE.

Transferring files

Terminal-emulation applications written using the Apple 3270 API can also be used to transfer files to and from the host.

DFT file transfer

To accomplish DFT file transfer, you can use the passthrough data mechanism to send and receive structured fields. For example, you can use Send_Passthru_Data and Get_Passthru_Data to send and receive D0 structured fields with the INDSFILE 3270 PC file transfer method **(anything else???)**.

CUT file transfer

CUT drivers cannot support structured fields. If you want to accomplish this type of file transfer, you can send and receive unformatted presentation spaces to and from the host **(anything else???)**.

Sample 3270 application

This section presents the DFTerm application. This application is essentially a skeleton that sets up one working DFT session.

The application is contained in the three files contained in this section, as follows:

DFTerm.c	Contains the main part of the application
Term.c	Contains the routines that actually support the 3270 terminal operations
Translate.c	Contains the tables that handle the key translation from MAC II keyboards to 3270 keys

The DFTerm.c file

This file contains the main part of the DFTerm sample application.

```
 * in TERM.C. *
 */
/*
 * File DFTerm.c
 *
 * Copyright Apple Computer, Inc. 1985-1987
 * All rights reserved.
 *

 * This program exersizes the 3270 DFT terminal capabilities of the APPLE 3270 API
 * interface. Only one session is developed.  The reader should understand that only
 * the following API  calls are used:
 *        Open_Host_Connection            Close_Host_Connection
 *        Connect_to_PS                   Disconnect_From_PS
 *        Send_Keys                       Get_Update
 *        Init_3270_API                   Term_3270_API
 *
 * The key translation from MAC II Keyboards to 3270 keys is handled in
 *  TRANSLATE.C.
 * The template for this program is an extension of "Sample.C" that is
 * distributed with the MPW C release from APPLE.
 * The routines that actually support the 3270 terminal operations are contained
```

```
 * in TERM.C.  When launched, the code in DoSlotPick.C,presents a dialog
 * box requesting slot information.When the ZORRO ROMs are completed,
 * the support routine FindServers will accomplish this function.
 */

#include  "dfterm.h"
#include  "Trantab.c"  /* The default keyboard mappings */


/*-------------------------------------------------------------------------------------------*/
/* Global Defines */
/*-------------------------------------------------------------------------------------------*/
/*
 * Resource ID constants.
 */
#define   appleID          128                  /* This is a resource ID */
#define   fileID           129                  /* ditto */
#define   editID           130                  /* ditto */

#define   appleMenu        0                     /* MyMenus[] array indexes */
#define   aboutMeCommand   1

#define   fileMenu         1
#define   quitCommand      1

#define   editMenu         2
#define   undoCommand      1
#define   cutCommand       3
#define   copyCommand      4
#define   pasteCommand     5
#define   clearCommand     6

#define menuCount          3
/*
 * For the one and only text window
 */
#define windowID           128
/*
 * For the About ... DLOG
 */
#define   aboutMeDLOG      128
#define   okButton         1
#define   authorItem       2                    /* For SetIText */
#define   languageItem     3                    /* For SetIText */

#define SETRECT(rectp, _left, _top, _right, _bottom) \
          (rectp)->left = (_left), (rectp)->top = (_top),  \
          (rectp)->right = (_right), (rectp)->bottom = (_bottom)

/*
 * HIWORD and LOWORD macros, for readability.
 */
#define HIWORD(aLong)  (((aLong) >> 16) & 0xFFFF)
#define LOWORD(aLong)  ((aLong) & 0xFFFF)
```

```
/*
 * HIBYTE and LOBYTE macros, for readability.
 */
#define HIBYTE(aWord)  (((aWord) >> 8) & 0x00FF)
#define LOBYTE(aWord)  ((aWord) & 0x00FF)

#define T75  0
#define   T102        1

/*-----------------------------------------------------------------------------------------------*/
/* Global Data objects */
/*-----------------------------------------------------------------------------------------------*/
extern char cursorActive;   /* in TERM.C */
extern _DataInit();
MenuHandle            MyMenus[menuCount];  /* The menu handles
Boolean               DoneFlag;             /* Becomes TRUE when File/Quit chosen
Rect                   dragRect;            /* limit rect for dragging rectangles on the screen
Rect                   growRect;            /* limit rect for growing rectangles
struct dft_session {                        /* for each dft lu-lu terminal session
          WindowPtr    myWindow;            /* Macintosh window for this session
          BYTE         ps_id;               /* presentation space id
          BYTE         last_request;  /* last open/close/send keys request                    */
          UPD80_REC    ps[25];              /* Presentation Space Buffer
          UPD80_REC    dab[25];             /* DAB Buffer
          API_REQ      req_blk;             /* Request block for open/close/send keys          */
          API_REQ      Gps_blk;             /* Requst Block for Get_Update
} *dft[MAX_SESSIONS];
Handle    api_vars;                         /* Heap Memory needed by interface */
BYTE      session;                          /* lu-lu session number, 1-5 */
char      *stat = "-------------------------------------------------------------------------------";
BYTE      kbtype;
BYTE      Slot;
BYTE      num_sessions;                     /* determined by user in DLOG */
short     err;
BYTE      saved_conn_id;
APPLE_DFT_CONFIG_INFO
          DFT_CFG;                          /* DFT config structure */

BYTE kbuf_toggle = 0;                       /* indicates which key buffer is current */
BYTE key_q_index = 0;
#define KEY_BUF_SIZE 64                     /* nine is the max seen in practice */
union keys {                                /* single key buffer; and overflow key q buffer *
    BYTE key[2];
    WORD key_buf;
    } kbuf_q[2][KEY_BUF_SIZE];  /* double buffering overflow keystrokes */

union keys *kbuf_current;

BYTE sessions_started = 0;   /* num sessions started */

/*-----------------------------------------------------------------------------------------------*
/* Init_API
 *      issues the Open_Host_Connection
 */
```

```
/*-----------------------------------------------------------------------------*
char Init_API()
{
    WORD relSlot;
    BYTE sess_num;

    relSlot = Slot;
/*  First get some memory from application heap for the API */
    api_vars = Init_3270_API();
/*  Issue an Open_Host_Connection, which returns immediately */
    dft[0]->req_blk.api_vars = api_vars;
    dft[0]->req_blk.net_addr.aNode = 0;
    dft[0]->req_blk.port_id = Slot;
    dft[0]->req_blk.io_compl = nil;
    dft[0]->req_blk.openhc.conn_type = OC_APPLE_DFT;
    dft[0]->req_blk.openhc.open_type = OC_COLD;
    dft[0]->req_blk.openhc.config_infop = &DFT_CFG;
    dft[0]->req_blk.openhc.config_info_len = sizeof(DFT_CFG);
    DFT_CFG.slot_map = 1;
    DFT_CFG.slot_map <<= Slot;

    for (sess_num = 0; sess_num < 5; sess_num++)
        DFT_CFG.slot_info[relSlot].lu_type[sess_num] = 0;
    for (sess_num = 0; sess_num < num_sessions; sess_num++)
        DFT_CFG.slot_info[relSlot].lu_type[sess_num] = ADFT_LU_TYPE_2;

    if (err = Open_Host_Connection(&(dft[0]->req_blk))) {
        ErrorMessage("Open_Host_Connection Error",err);
        Term_3270_API(api_vars);
        return 0;
        }
    if ((err = DFT_CFG.slot_status[relSlot]) != NO_ERR) {
        ErrorMessage("Slot Status Non-Zero",err);
        Term_3270_API(api_vars);
        return 0;
        }
    if ((err = DFT_CFG.slot_info[relSlot].ps_status[0]) != ADFT_PS_SUPP) {
        ErrorMessage("Apple DFT Not Supported",err);
        Term_3270_API(api_vars);
        return 0;
        }
    saved_conn_id = dft[0]->req_blk.openhc.ret_conn_id;

    return TRUE;
}
/*-----------------------------------------------------------------------------*
/* Init_Connect
 *      issues the API Connect_To_PS call
 */
/*-----------------------------------------------------------------------------*

char Init_Connect(session)
BYTE session;
{
```

```
unsigned int junk;

/*  Issue a Connect_To_PS.  Test for complete in the main loop */
    dft[session]->req_blk.api_vars = api_vars;
    dft[session]->req_blk.net_addr.aNode = 0;
    dft[session]->req_blk.port_id = Slot;
    dft[session]->req_blk.io_compl = nil;
    dft[session]->req_blk.conn_id = saved_conn_id;
    dft[session]->req_blk.ps_id = 0xFF;
    dft[session]->req_blk.connps.keybd_tabp = ktab;
    dft[session]->req_blk.connps.keybd_tab_len = sizeof(ktab);
    dft[session]->req_blk.connps.dbc_ebc_tabp = dbc_ebc;
    dft[session]->req_blk.connps.ebc_dbc_tabp = ebc_dbc;
    dft[session]->req_blk.connps.dbc_asc_tabp = dbc_asc;
    dft[session]->req_blk.connps.asc_dbc_tabp = asc_dbc;
    dft[session]->req_blk.connps.color_supp = CP_4_COLOR_EXT;
    dft[session]->req_blk.connps.num_lock = FALSE;
    dft[session]->req_blk.connps.scrn_emul = CP_MOD_2;
    dft[session]->req_blk.connps.query_reply_len = 0;
    dft[session]->req_blk.connps.query_replyp = nil;
    dft[session]->req_blk.connps.type_pass_data_len = 0;
    dft[session]->req_blk.connps.type_pass_datap = nil;
    dft[session]->req_blk.connps.modifiers = NO_MODS;

    /* ErrorMessage("Connect_To_PS session",session); */
    /* junk = &(dft[session]->req_blk); */
    /* Debugger();*/

    if (err = Connect_To_PS(&(dft[session]->req_blk),ASYNC)) {
        ErrorMessage("Connect_To_PS Error",err);
        Close_Host_Connection(&dft[session]->req_blk);
        Term_3270_API(api_vars);
        return FALSE;
        }
    dft[session]->last_request = RC_CONNECT_TO_PS;
    return TRUE;
}
/*-----------------------------------------------------------------------------------------------*/
/*
 * showBuf()
 *  calls the routines in TERM.C to display the buffer
 *  returned by Get_PS_Update
 */
/*-----------------------------------------------------------------------------------------------*/
void showBuf(session,display)
BYTE session;
BYTE display; /* write to the current graph port ? */
{
    int     j;
    WORD    i;
    BYTE    row,
            col;
    i = (WORD) &dft[session]->Gps_blk.getupd; /* non functional statement for debugging */
#ifndef DEBUG_ME
```

```
    for (i=0; i< dft[session]->Gps_blk.getupd.num_dab_recs; i++) {
        if (dft[session]->dab[i].row  < 0 || dft[session]->dab[i].row  > 24) {
            ErrorMessage("DAB row",dft[session]->dab[i].row+1);
            j=(int)&dft[session]->Gps_blk.getupd;
            Debugger();
        }
        else if (dft[session]->dab[i].col+1 < 0 || dft[session]->dab[i].col > 80) {
            ErrorMessage("DAB col",dft[session]->dab[i].col+1);
            j=(int)&dft[session]->Gps_blk.getupd;
            Debugger();
        }
        else if (dft[session]->dab[i].len < 0 || dft[session]->dab[i].len > 80) {
            ErrorMessage("DAB len",dft[session]->dab[i].len);
            j=(int)&dft[session]->Gps_blk.getupd;
            Debugger();
        }
        else cpyAttr(dft[session]->dab[i].row + 1, dft[session]->dab[i].col + 1,
            &(dft[session]->dab[i].data[0]), dft[session]->dab[i].len,session);
        }
#else
    for (i=0; i < dft[session]->Gps_blk.getupd.num_dab_recs; i++) {
        if(dft[session]->dab[i].row >= 0 || dft[session]->dab[i].row < 25)
            cpyAttr(dft[session]->dab[i].row + 1, dft[session]->dab[i].col + 1,
                &(dft[session]->dab[i].data[0]), dft[session]->dab[i].len,session);
    }

#endif DEBUG_ME
    for (i=0; i< dft[session]->Gps_blk.getupd.num_ps_recs; i++) {
        if (dft[session]->ps[i].row == 0xff)
            dft[session]->ps[i].row = 25;
        showLine(dft[session]->ps[i].row + 1, dft[session]->ps[i].col + 1,
                &(dft[session]->ps[i].data[0]), dft[session]->ps[i].len,UPDATE,display,session);
        }
    row = dft[session]->Gps_blk.getupd.cursor_row + 1; col = dft[session]->Gps_blk.getupd.cursor_col-
    cursor_position(row, col,session);
}

/*-------------------------------------------------------------------------------------------------------
/*
 * setGet
 *  sets up and issues the Get_Update request
 */
/*-------------------------------------------------------------------------------------------------------
Boolean setGet(session)
BYTE session;
{


    /* ErrorMessage("Setget session",session); */
    /* ErrorMessage("Setget ps_id",dft[session]->ps_id); */

    dft[session]->Gps_blk.net_addr.aNode = 0;
    dft[session]->Gps_blk.api_vars = api_vars;
    dft[session]->Gps_blk.port_id = Slot;
```

```
    dft[session]->Gps_blk.conn_id = saved_conn_id;
    dft[session]->Gps_blk.ps_id = dft[session]->ps_id;
    dft[session]->Gps_blk.getupd.wait_time = 0xFFFF;
    dft[session]->Gps_blk.getupd.ps_recp = &(dft[session]->ps[0]);
    dft[session]->Gps_blk.getupd.dab_recp = &(dft[session]->dab[0]);
    dft[session]->Gps_blk.getupd.dabe_recp = 0;
    dft[session]->Gps_blk.getupd.eab_recp = 0;
    dft[session]->Gps_blk.getupd.modifiers = NO_MODS;
    if (err = Get_Update(&dft[session]->Gps_blk,ASYNC)) {
        ErrorMessage("Glue Get_Update Error",err);
        return FALSE;
        }
    return TRUE;
}

/*-----------------------------------------------------------------------------------------------------*/
/* ClearConnect
 *  does a close & term
 */
/*-----------------------------------------------------------------------------------------------------*/
void    ClearConnect()
{
    Close_Host_Connection(&dft[session]->req_blk);
    Term_3270_API(api_vars);
}

/*-----------------------------------------------------------------------------------------------------*/
/* MAIN */
/*-----------------------------------------------------------------------------------------------------*/
int main()
{
    /* local variables */
    GrafPtr                 tmpWindow;
    Rect                    dragRect;
    long                    newSize;        /* new window size returned by GrowWindow() */
    EventRecord             myEvent;
    WindowPtr               theActiveWindow;
    WindowPtr               whichWindow;
    extern void             setupMenus();
    extern void             doCommand();
    char                    ch;
    BYTE                    i;
    int                     j;
    OSErr                   rtnErr;
    struct SysEnvRec        world;
    struct SysEnvRec        *theWorld;
    char * tmp;

    /*
     * Initialization traps
     */
    UnloadSeg(_DataInit);
    InitGraf(&qd.thePort);
    InitFonts();
```

```
FlushEvents(everyEvent, 0);
InitWindows();
InitMenus();
TEInit();
InitDialogs(nil);
InitCursor();
/*
 * setupMenus is execute-once code, so we can unload it now.
 */
setupMenus();  /* Local procedure, below */
UnloadSeg(setupMenus);


num_sessions = 0x03;                    /* set default as 3 sessions */
if (!DoNumSessions(&num_sessions))    /* Display the dialog */
    return FALSE;

/* get non relocatable memory from the application heap for each session */
for (session = 0; session < num_sessions; session++) {
    dft[session] = (struct dft_session *) NewPtr(sizeof(struct dft_session));
    if (dft[session] == NULL) {
        ErrorMessage("No Applic Heap Memory",dft[session]);
        return 0;
    }
    /* clear the heap screen buffer */
    tmp = (char *) dft[session];
    for (j=0; j < sizeof(struct dft_session); j++)
        *(tmp + j) = 0;
}


theWorld = &world;                      /* Determine MAC II keyboard type */
rtnErr = SysEnvirons(1,theWorld);
if (theWorld->keyBoardType == 4) {
    kbtype = T102;
    }
else
    {
    kbtype = T75;
    }
Slot = 0xCC;                            /* set default slot as c */
if (!DoSlotPick(&Slot))                 /* Display the dialog */
    return FALSE;

if (!Init_API())                        /* Open host connection */
    return FALSE;

/* open windows for each session, initailize  */
for (session = 0; session < num_sessions; session++) {
    dft[session]->myWindow = GetNewWindow(windowID+session, 0, (WindowPtr) -1);
    SetPort(dft[session]->myWindow);
    SetRect(&dragRect,qd.screenBits.bounds.left+4,
                qd.screenBits.bounds.top+24,
                qd.screenBits.bounds.right-4,
```

```
                    qd.screenBits.bounds.bottom-4);

/*
 * growRect will be used in GrowWindow() to limit a window's size during growing
 * top is min height, left is min width
 * bottom is max height, right is max width
 */

    SetRect(&growRect,100,
                    100,
                    qd.screenBits.bounds.right,
                    qd.screenBits.bounds.bottom);

    /* setup the terminal
     * and initialize the interface
     */

    dft[session]->ps_id = 0xFF;                          /* first available session */

    InitPage(session);                                   /* in TERM.C */
    showLine(25,1,stat,80,UPDATE,session);      /* display the bar */


    /* make it work first time through the main for loop */
    dft[session]->Gps_blk.result = RSP_PENDING;

    if (!Init_Connect(session))                          /* connect to each presentation space */
        return FALSE;                                    /* SHUTDOWN SHOULD BE CLEANED UP HERE */

}

kbuf_current = kbuf_q[kbuf_toggle]; /* initialize key buffer pointer */


/*
 * Ready to go.
 * Start with a clean event slate, and cycle the main event loop
 * until the File/Quit menu item sets DoneFlag.
 *
 * It would not be good practice for the doCommand() routine to
 * simply ExitToShell() when it saw the QuitItem -- to ensure
 * orderly shutdown, satellite routines should set global state,
 * and let the main event loop handle program control.
 */
DoneFlag = false;
for ( ;; ) {
    if (DoneFlag) {
        break;          /* from main event loop */
        }
    /*
     * Main Event tasks:
     */
    SystemTask();
```

```
theActiveWindow = FrontWindow(); /* Used often, avoid repeated calls */
for (session = 0; session < num_sessions; session++ ) {
    if (theActiveWindow == dft[session]->myWindow) {
        if ((dft[session]->last_request) && (dft[session]->req_blk.result != RSP_PENDING)) {
            switch (dft[session]->last_request) {
                case RC_CONNECT_TO_PS:
                    if (dft[session]->req_blk.result != NO_ERR) {
                        ErrorMessage("Rslt Connect_to_PC Error",dft[session]->req_blk.result);
                        DoneFlag = TRUE;
                        ClearConnect();
                        return 0;
                    }

                    /* bounds check  ps_id received from card */
                    if ((dft[session]->req_blk.connps.ret_ps_id < 1)
                        || (dft[session]->req_blk.connps.ret_ps_id > 5)) {
                        ErrorMessage("Invalid Session ID  = ",dft[session]->req_blk.connps.ret_p
                        DoneFlag = TRUE;
                        ClearConnect();
                        return 0;
                    }
                    /* save the returned ps_id */
                    dft[session]->ps_id = dft[session]->req_blk.connps.ret_ps_id;
                    dft[session]->req_blk.ps_id = dft[session]->ps_id;

                    if (!setGet(session)) {  /* post get_update on this session */
                        DoneFlag = TRUE;
                        ClearConnect();
                        return 0;
                    }
                    dft[session]->last_request = 0;
                    break;
                case RC_SEND_KEYS:
                    if (dft[session]->req_blk.result != NO_ERR) {

                        ErrorMessage("SendKey Return Error",dft[session]->req_blk.result);
                        SysBeep(1);
                    }

                    if (key_q_index) { /* keys strokes are buffered */

                        dft[session]->req_blk.sendkey.num_keys_to_send = key_q_index;
                        key_q_index = 0;
                        dft[session]->req_blk.sendkey.keys_bufp = (BYTE *) kbuf_current;

                        if (err = Send_Keys(&(dft[session]->req_blk),ASYNC)) {
                                ErrorMessage("GLUE Send Keys Error",err);
                                ClearConnect();
                                return 0;
                        }
                        kbuf_current = kbuf_q[kbuf_toggle ^= 1]; /* switch buffers */

                    }
                    else { /* key strokes not buffered */
```

```
                        dft[session]->last_request = 0;
                    }
                    break;
                case RC_DISCONNECT_FROM_PS:
                    if (dft[session]->req_blk.result != NO_ERR) {
                        ErrorMessage("Rtn Disconnect Error",dft[session]->req_blk.result);
                    }
                    if (err = Close_Host_Connection(&dft[session]->req_blk)) {
                        ErrorMessage("Close Host Error",err);
                    }
                    DoneFlag = TRUE;
                    Term_3270_API(api_vars); /*  release memory */
                    return 0;                       /* Exit_to_Shell */
                    break;
            }   /* switch case API request completion code */
        } /* if the last request just completed */

        if (dft[session]->Gps_blk.result != RSP_PENDING) {
            /* ErrorMessage("Get_Update Completed",0); */
            if (dft[session]->Gps_blk.result != NO_ERR) {
                ErrorMessage("Get_Update Error",dft[session]->Gps_blk.result);
                ClearConnect();
                DoneFlag = TRUE;
                return FALSE;
            }
            GetPort(&tmpWindow);
            /* Only draw text on the currently active window */
            if ((WindowPtr)tmpWindow == dft[session]->myWindow)
                showBuf(session,DISPLAY);
            else
                showBuf(session,NO_DISPLAY);
            if (!setGet(session)) {
                DoneFlag = TRUE;
                ClearConnect();
                return 0;
            }
        }
        break; /* active window found, break out of "for each dft window" loop */
    }

} /* end for each dft window loop */
if (!GetNextEvent(everyEvent, &myEvent)) { /* null event */
    }
switch (myEvent.what) {
    case mouseDown:
        switch (FindWindow(&myEvent.where, &whichWindow)) {
            case inSysWindow:
                SystemClick(&myEvent, whichWindow);
                break;

            case inMenuBar:
                doCommand(MenuSelect(&myEvent.where));
                break;
```

```
            case inDrag:
                DragWindow(whichWindow, &myEvent.where, &dragRect);
                break;

            case inGrow:
                /* There is no grow box. (Fall through) */
                /* no, let's grow the window */
                newSize = GrowWindow(whichWindow, &myEvent.where, &growRect);
                SizeWindow(whichWindow, (short) LOWORD(newSize), (short) HIWORD(newSize), TRUE)

            case inContent:
                if (whichWindow != theActiveWindow) {
                    SelectWindow(whichWindow);

                }
                break;

            default:
                break;
        }/*endsw FindWindow*/
        break;

    case keyDown:
    case autoKey:
        for (session = 0; session < num_sessions; session++) {
            if (dft[session]->myWindow == theActiveWindow) {
                if (myEvent.modifiers & cmdKey) {
                    doCommand(MenuKey(myEvent.message & charCodeMask));
                }
                else {
                    ch = (myEvent.message & keyCodeMask) >> 8;
                    if (Map_Key(kbtype, ch, myEvent.modifiers) == 0x0000) { /* char maps to open
                        SysBeep(1);
                        break;
                    }
                    if (!dft[session]->last_request) { /* no keys buffered */
                        kbuf_current->key_buf = Map_Key(kbtype, ch, myEvent.modifiers);
                        dft[session]->last_request = RC_SEND_KEYS;
                        dft[session]->req_blk.sendkey.num_keys_to_send = 1;
                        key_q_index = 0;
                        dft[session]->req_blk.sendkey.keys_bufp = (BYTE *) kbuf_current;

                        if (err = Send_Keys(&dft[session]->req_blk,ASYNC)) {
                                ErrorMessage("GLUE Send Keys Error",err);
                                ClearConnect();
                                return 0;
                        }
                        kbuf_current = kbuf_q[kbuf_toggle ^= 1]; /* switch buffers */
                    }
                    else {      /* buffer keystrokes */
                        (kbuf_current + key_q_index)->key_buf = Map_Key(kbtype, ch, myEvent.mod
                        ++key_q_index;
                        /*      SysBeep(1); */
                    }
```

```
                }
            break;
            } /* endif myWindow */
        }   /* end for each session */
        break;

    case activateEvt:
        whichWindow = (WindowPtr) myEvent.message;
        for (session=0; session < num_sessions; session++ ) {
            if (whichWindow == dft[session]->myWindow) {
                if (myEvent.modifiers & activeFlag) {
                    SetPort(whichWindow);   /* make SURE drawing works */
                    DisableItem(MyMenus[editMenu], 0);
                    DrawMenuBar();      /* and redraw the menu bar */
                    if (!cursorActive)
                        start_cursor();                 /* crank up the cursor */
                } else {
                    EnableItem(MyMenus[editMenu], 0);
                    DrawMenuBar();
                    if (cursorActive)
                        stop_cursor();        /* stop the cursor */
                }
                break; /* window found and [de]activated, exit for loop */
            }
        }
        break;

    case updateEvt:
        whichWindow = (WindowPtr) myEvent.message;
        for (session=0; session < num_sessions; session++ ) {
            if (whichWindow  == dft[session]->myWindow) {
                GetPort(&tmpWindow);
                SetPort(whichWindow); /* set port */
                BeginUpdate(whichWindow);
                    term_redraw(session);
                EndUpdate(whichWindow);
                SetPort(tmpWindow); /* restore previous port */
                break;
            }
        }
        break;

    default:
        break;

  }/*endsw myEvent.what*/

}/*endfor Main Event loop*/
/*
 * Cleanup here.
 */
for (session=0; session < num_sessions; session++ ) {
    CloseWindow(dft[session]->myWindow);
    DisposPtr(dft[session]);
```

```
    }
    return 0;          /* Return from main() to allow C runtime cleanup */
}
/*--------------------------------------------------------------------------------------------------*/
/*-----------------------------------------SEGMENT--------------------------------------------------*/
/*--------------------------------------------------------------------------------------------------*/

/*
 * Demonstration of the segmenting facility:
 *
 * This code is execute-once, so we toss it in the "Initialize"
 * segment so that main() can unload it after it's called.
 *
 * There really isn't much here, but it demonstrates the segmenting facility.
 */
/*
 * Set the segment to Initialize.  BEWARE: leading and trailing white space
 * would be part of the segment name!
 */
#define __SEG__ Initialize

/*--------------------------------------------------------------------------------------------------*/
/* setupMenus
/*--------------------------------------------------------------------------------------------------*/
/*
 * Set up the Apple, File, and Edit menus.
 * If the MENU resources are missing, we die.
 */
void setupMenus()
{
    extern     MenuHandle    MyMenus[];
    register   MenuHandle    *pMenu;

    /*
     * Set up the desk accessories menu.
     * The "About Sample..." item, followed by a grey line,
     * is presumed to be already in the resource.  We then
     * append the desk accessory names from the 'DRVR' resources.
     */
    MyMenus[appleMenu] = GetMenu(appleID);
    AddResMenu(MyMenus[appleMenu], (ResType) 'DRVR');
    /*
     * Now the File and Edit menus.
     */
    MyMenus[fileMenu] = GetMenu(fileID);
    MyMenus[editMenu] = GetMenu(editID);
    /*
     * Now insert all of the application menus in the menu bar.
     *
     * "Real" C programmers never use array indexes
     * unless they're constants :-)
     */
    for (pMenu = &MyMenus[0]; pMenu < &MyMenus[menuCount]; ++pMenu) {
        InsertMenu(*pMenu, 0);
```

```
    }

    DrawMenuBar();

    return;
}
/*--------------------------------------------------------------------------------------*/
/*-----------------------------------SEGMENT--------------------------------------------*/
/*--------------------------------------------------------------------------------------*/

/*
 * Back to the Main segment.
 */
#define __SEG__ Main

/*--------------------------------------------------------------------------------------*/
/* showAboutMeDialog */
/*--------------------------------------------------------------------------------------*/
/*
 * Display the Sample Application dialog.
 * We insert two static text items in the DLOG:
 *     The author name
 *     The source language
 * Then wait until the OK button is clicked before returning.
 */
void showAboutMeDialog()
{
    GrafPtr             savePort;
    DialogPtr           theDialog;
    short               itemType;
    Handle              itemHdl;
    Rect                itemRect;
    short               itemHit;

    GetPort(&savePort);
    theDialog = GetNewDialog(aboutMeDLOG, nil, (WindowPtr) -1);
    SetPort(theDialog);

    GetDItem(theDialog, authorItem, &itemType, &itemHdl, &itemRect);
    SetIText(itemHdl, "Gerry A. Brown");
    GetDItem(theDialog, languageItem, &itemType, &itemHdl, &itemRect);
    SetIText(itemHdl, "MPW C");

    do {
        ModalDialog(nil, &itemHit);
    } while (itemHit != okButton);

    CloseDialog(theDialog);

    SetPort(savePort);
    return;
}
/*--------------------------------------------------------------------------------------*/
/* doCommand */
```

```
/*-----------------------------------------------------------------------------------------------
/*
 * Process mouse clicks in menu bar
 */
void doCommand(mResult)
    long mResult;
{
    int             theMenu, theItem;
    char            daName[256];
    GrafPtr     savePort;
    extern MenuHandle MyMenus[];
    extern Boolean  DoneFlag;
    extern TEHandle  TextH;
    extern void   showAboutMeDialog();


    theItem = LOWORD(mResult);
    theMenu = HIWORD(mResult);              /* This is the resource ID */

    switch (theMenu) {
        case appleID:
            if (theItem == aboutMeCommand) {
                showAboutMeDialog();
            } else {
                GetItem(MyMenus[appleMenu], theItem, daName);
                GetPort(&savePort);
                (void) OpenDeskAcc(daName);
                SetPort(savePort);
            }
            break;

        case fileID:
            switch (theItem) {
                case quitCommand:
                    if (!dft[session]->last_request) {
                        dft[session]->req_blk.discps.modifiers = NO_MODS;
/*
ErrorMessage("Disconnecting session",session);
ErrorMessage("Disconnecting ps_id is",dft[session]->req_blk.ps_id);
*/                      if (err = Disconnect_From_PS(&dft[session]->req_blk,ASYNC)) {
                            ErrorMessage("Glue Disc_PS Error",err);
                            DoneFlag = TRUE;
                            ClearConnect();
                            }
                    }
                    dft[session]->last_request = RC_DISCONNECT_FROM_PS;
                    break;
                default:
                    break;
            }
            break;

        case editID:
            SystemEdit(theItem-1);
```

```
            }
            break;

        case editID:
            SystemEdit(theItem-1);
            break;
    }/*endsw  theMenu*/

    HiliteMenu(0);

    return;
}
```

The Term.c file

This file contains the routines that actually support the 3270 terminal operations.

```
/*
 * term.c
 * Responsible for maintaining the 3270 display screen
*/


#include "dfterm.h"

#define False               0
#define True                1
#define FALSE               0
#define TRUE                1
#define MAXLIN             26
#define MAXCOL             80
#define LINEHEIGHT         11
#define CHARWIDTH           6
#define TOPMARGIN           3                    /* Terminal display constants */
#define BOTTOMMARGIN  (LINEHEIGHT * MAXLIN + TOPMARGIN)
#define LEFTMARGIN          3
#define RIGHTMARGIN   (CHARWIDTH * MAXCOL + LEFTMARGIN)
#define LINEADJ             3                    /* Amount of char below base line */
#define CR                0x0d
static Rect penRect;
static FontInfo fontstuff;



/* cursor variables */
char cursorActive;                /* Global - referenced by main */

int topmargin=TOPMARGIN,          /* Edges of adjustable window */
    bottommargin=BOTTOMMARGIN,
    textstyle=0;
```

```
    BYTE curlin, curcol;                    /* Cursor position */
    BYTE savcol, savlin;                    /* Cursor save variables */
} screen[MAX_SESSIONS];

/*************************************************************************/
/* cursor stuff */
void getPenRect(r)
  Rect *r;
 { Point pt;
   GetPen(&pt);
   r->top =  pt.v;
   r->bottom = pt.v + fontstuff.descent;
   r->left = pt.h;
   r->right = pt.h + fontstuff.widMax ;
  }
void start_cursor()
   {
     cursorActive = True;
    getPenRect(&penRect);
    ForeColor(blackColor);
    InvertRect(&penRect);
    }

void stop_cursor()
   {
     cursorActive = False;
    getPenRect(&penRect);
    InvertRect(&penRect);
    }

/*************************************************************************/

/* Connect support routines */

InitPage(session)
BYTE session;

{

    TextMode(srcCopy);
    TextFont(86);
    TextSize(9);
    GetFontInfo(&fontstuff);
    init_term(session);  /* Set up some terminal variables */
    home_cursor(session);  /* Go to the upper left */
    cursor_save(session);  /* Save this position */
    start_cursor();
}

home_cursor(session)
BYTE session;
{
    absmove(0,0,session);
}
```

```
cursor_save(session)
BYTE session;
{
    screen[session].savcol = screen[session].curcol;  /* Save the current line and column */
    screen[session].savlin = screen[session].curlin;
}

cursor_restore(session)
BYTE session;
{
    absmove(screen[session].savcol,screen[session].savlin,session); /* Move to old cursor position */
}

/*
 * Move to absolute position hor char and ver line.
 *
 */

absmove(hor,ver,session)
BYTE hor,ver,session;
{
    MoveTo(hor*CHARWIDTH+LEFTMARGIN,(ver+1)*LINEHEIGHT+TOPMARGIN-LINEADJ);
    screen[session].curcol = hor;
    screen[session].curlin = ver;
}

cursor_position(line,col,session)
BYTE line;
BYTE col;
BYTE session;
{
    if (line > 24)
        return;
    line--;
    col--;
    stop_cursor();
    absmove(col,line,session);
    cursor_save(session);
    start_cursor();
}

term_redraw(session)
BYTE session;
{
    BYTE i;
    BYTE *astr;

#ifdef DEBUG
    DebugStr("Entering term_redraw");
#endif
    for (i=0; i<MAXLIN; i++) {
        astr = &(screen[session].scr[i][0]);
        showLine(i+1,1,astr,80,NO_UPDATE,DISPLAY,session);
```

```
    }
#ifdef DEBUG
    DebugStr("Exiting term_redraw");
#endif
}

init_term(session)
BYTE session;
{
    int i;
    int j;

    for (i=0; i<MAXLIN; i++) {
        for (j=0; j<MAXCOL; j++) {
            screen[session].scr[i][j] = ' ';
            screen[session].attr[i][j] = 0;
            }
        screen[session].scr[i][MAXCOL] = '\0'; /* Terminate the lines as strings */
    }
}
/*-------------------------------------------------------------------------------------------------*/
/*
 *  setAttr - sets the screen attributes sent by the controller
 */
/*-------------------------------------------------------------------------------------------------*/
setAttr(pattr)
BYTE pattr;
{
    BYTE temp;

    /*
     *  Set Display Characteristics
     */
    temp = (pattr >> 1) & 0x03;
    ShowPen();
    if ((temp == 0) || (temp == 1)) {       /* Normal display */
        TextFont(86);
        }
    else if (temp == 2) {                   /* Intensified display */
        TextFont(87);
        }
    else {                                  /* non - display handled by not showing */
        }
    /*
     *  Set Mode Characteristics
     */
    temp = (pattr >> 6) & 0x03;
    TextMode(srcCopy);
    TextFace(normal);
    BackColor(whiteColor);
    if (temp == 0) {                        /* Normal video */
        TextFace(normal);
        }
    else if (temp == 1) {                   /* Blink video */
```

```
        BackColor(yellowColor);
        }
    else if (temp == 2) {                   /* Reverse video */
        TextMode(notSrcCopy);
        }
    else {                                  /* underline video */
        TextFace(underline);
        }
    /*
     *  Set Color Characteristics
     */
    temp = (pattr >> 3) & 0x07;
    switch (temp) {
        case 0:
            ForeColor(blackColor);
            break;
        case 1:
            ForeColor(blueColor);
            break;
        case 2:
            ForeColor(redColor);
            break;
        case 3:
            ForeColor(magentaColor);
            break;
        case 4:
            ForeColor(greenColor);
            break;
        case 5:
            ForeColor(cyanColor);
            break;
        case 6:
            ForeColor(yellowColor);
            break;
        case 7:
            ForeColor(blackColor)           ; /* Should be white */
            break;
        }
}
/*-------------------------------------------------------------------------------------------------
/*
 *  showLine - called from 3270 to output the string to the terminal
 *  and actually does a drawscreen to the screen.  The screen coordinates
 *   are externally rows 1 - 24; and columns 1 - 80;  The OIA is supported
 *   on the screen at row 26.  Main displays a dashed-line in row 25.
 */
/*-------------------------------------------------------------------------------------------------*/
showLine(line,col,astr,len,update,display,session)
BYTE          line;                     /* line number to modify     */
BYTE          col;                      /* starting colume                    */
BYTE          *astr;                    /* 1 to 80 BYTEs to dsiplay  */
BYTE          len;                      /*  how many BYTEs to do     */
BYTE          update;                   /*  update the scr buffer?   */
BYTE          display;                  /*  display in the current graphport ? */
```

```
BYTE          session;                  /* which screen ?                  */
{
    BYTE          tcol;
    BYTE          scol;
    BYTE          tlen;
    BYTE          i;
    BYTE          *tstr;
    Boolean       cursorWasActive;
    BYTE          tattr;

#ifdef DEBUG
    DebugStr("Entering showLine");
#endif

    line--;                               /* Make line & col 0 rel */
    col--;
    cursor_save(session);
    if (cursorWasActive = cursorActive)
        stop_cursor();
    if (line == 25 && update)             /* if the OIA, translate */
        StXlate(astr,len);
    tstr = astr;                          /* Copy string to scr buffer */
    if (update)
        for (i=0; i < len; i++) {
            screen[session].scr[line][col+i] = *tstr++;
        }
/*  if (line == 25) DebugStr(&(screen[session].scr[line][col])); */
    tlen = 0;
    scol = col;
    while (len) {
        scol = scol + tlen;
        tattr = screen[session].attr[line][scol];
        /* cDebugStr("\004CALL"); */
        setAttr(tattr);
        tlen = 1;
        len--;
        tcol = scol + 1;
        while (len && (tattr == screen[session].attr[line][tcol])) {
            tlen++;
            tcol++;
            len--;
            }
        if (display)
            if ((tattr & 0x06) != 0x06) {
                absmove(scol,line,session);
                DrawText(astr,(scol-col),tlen);
            }
    }
    cursor_restore(session);
    if (cursorWasActive) start_cursor();

#ifdef DEBUG
    DebugStr("Exiting showLine");
#endif
```

```
}

/*-----------------------------------------------------------------------------------------------*/
/* cpyAttr */
/*-----------------------------------------------------------------------------------------------*/
void cpyAttr(row,start,bfr,len,session)
BYTE row;
BYTE start;
BYTE *bfr;
WORD len;
{                                          /* copy attributes to our array */
    row--;
    start--;
    while (len) {
        screen[session].attr[row][start++] = *bfr++;
        len--;
        }
}
```

Term.c Contains the routines that actually support the 3270 terminal operations

The Translate.c file

This file contains the tables that handle the key translation from MAC II keyboards to 3270 keys.

```
/*
 *Translate.c
 *  Does the actual mapping from MAC II keyboards (regular and enhanced) to
 *the 3270 keyboard mappings.
 *  Also does the mapping of the inbound OIA.
 */
#include <types.h>
#include "api3270.h"
#include <events.h>
/* MAC II KEYBOARD MAPPING
 *
 *  maps to a type 87 keyboard
 *
 */
#define T75 0
static WORD M2KB[256] =
{0x0060, 0x0072, 0x0063, 0x0065, 0x0067, 0x0066, 0x0079, 0x0077, /* 00 - 0F */
 0x0062, 0x0075, 0x0000, 0x0061, 0x0070, 0x0076, 0x0064, 0x0071,
 0x0078, 0x0073, 0x0021, 0x0022, 0x0023, 0x0024, 0x0026, 0x0025, /* 10 - 1F */
 0x0011, 0x0029, 0x0027, 0x0030, 0x0028, 0x0020, 0x0015, 0x006E,
 0x0074, 0x001B, 0x0068, 0x006F, 0x0008, 0x006B, 0x0069, 0x0012, /* 20 - 2F */
 0x006A, 0x007E, 0x0018, 0x0033, 0x0014, 0x006D, 0x006C, 0x0032,
 0x0036, 0x0010, 0x0000, 0x0031, 0x0000, 0x003D, 0x0000, 0x0000, /* 30 - 3F */
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
 0x0000, 0x005E, 0x0000, 0x000C, 0x0000, 0x000D, 0x0000, 0x0040, /* 40 - 4F */
```

```
 0x0000, 0x0000, 0x0000, 0x0042, 0x0018, 0x0000, 0x0000, 0x0000,
 0x0000, 0x0041, 0x0000, 0x0049, 0x004A, 0x004B, 0x0046, 0x0047, /* 50 - 5F */
 0x0048, 0x0043, 0x0000, 0x0044, 0x004C, 0x0000, 0x0000, 0x0000,
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, /* 60 - 6f */
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, /* 70 - 7f */
 0x0000, 0x0000, 0x0000, 0x0016, 0x001A, 0x0013, 0x000E, 0x0000,
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, /* 80 - 8F */
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, /* 90 - 9F */
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, /* A0 - AF */
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, /* B0 - BF */
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, /* C0 - CF */
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, /* D0 - DF */
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, /* E0 - EF */
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, /* F0 - FF */
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000};

/*
 * MAC II ENHANCED KEYBOARD
 *
 *      maps to a type 87 keyboard
 */
#define T102 1
static WORD MACENC[512] =
{0x0060, 0x0072, 0x0063, 0x0065, 0x0067, 0x0066, 0x0079, 0x0077, /* 00 - 0F */
 0x0062, 0x0075, 0x0000, 0x0061, 0x0070, 0x0076, 0x0064, 0x0071,
 0x0078, 0x0073, 0x0021, 0x0022, 0x0023, 0x0024, 0x0026, 0x0025, /* 10 - 1F */
 0x0011, 0x0029, 0x0027, 0x0030, 0x0028, 0x0020, 0x0015, 0x006E,
 0x0074, 0x001B, 0x0068, 0x006F, 0x0000, 0x006B, 0x0069, 0x0012, /* 20 - 2F */
 0x006A, 0x007E, 0x0035, 0x0033, 0x0014, 0x006D, 0x006C, 0x0032,
 0x0036, 0x0010, 0x003D, 0x0031, 0x0000, 0x0000, 0x0000, 0x0000, /* 30 - 3F */
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0040, /* 40 - 4F */
 0x0000, 0x0000, 0x0000, 0x0042, 0x0018, 0x0000, 0x0000, 0x0000,
 0x0000, 0x0041, 0x0000, 0x0049, 0x004A, 0x004B, 0x0046, 0x0047, /* 50 - 5F */
 0x0048, 0x0043, 0x0000, 0x0044, 0x0045, 0x0000, 0x0000, 0x0000,
 0x0825, 0x0826, 0x0827, 0x0823, 0x0828, 0x0829, 0x0000, 0x0830, /* 60 - 6F */
 0x0000, 0x0040, 0x0000, 0x0041, 0x0000, 0x0820, 0x0000, 0x0811,
 0x0000, 0x0042, 0x0050, 0x005F, 0x0052, 0x000C, 0x0824, 0x000D, /* 70 - 7F */
 0x0822, 0x0034, 0x0821, 0x0016, 0x0013, 0x001A, 0x000E, 0x0000,
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, /* 80 - 8F */
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, /* 90 - 9F */
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, /* A0 - AF */
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, /* B0 - BF */
```

```
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, /* C0 - CF */
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, /* D0 - DF */
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, /* E0 - EF */
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, /* F0 - FF */
 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000};

                          /* status line xlate table */
 static BYTE sl_tbl[] =
/*                                   Second digit
    */
/*          0    1    2    3    4    5    6    7    8    9   a    b    c    d    e    f */
{
/* 0 */   0x20,0x20,0x0c,0x0a,0x20,0x0d,0x20,0x20,0x3e,0x3c,0x5b,0x5d,0x29,0x28,0x7d,0x7b,
/* 1 */   0x20,0x3d,0x27,0x22,0x2f,0x5c 0x7c,0x7c,0x3f,0x21,0x24,0xa2,0xa3,0xb4,0xa5,0xa5,
/* 2 */   0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39,0xa7,0xc4,0x23,0x40,0x25,0x5f,
/* 3 */   0x26,0x2d,0x2e,0x2c,0x3a,0x2b,0xc2,0xd1,0xa1,0x20,0x5e,0x7e,0xac,0x60,0xab,0xa5,
/* 4 */   0x88,0x8f,0x93,0x98,0x9d,0x8b,0x9b,0x79,0x88,0x8f,0x8e,0x93,0x98,0x9c,0x9f,0x8d,
/* 5 */   0x8a,0x91,0x95,0x9a,0x9f,0x89,0x90,0x94,0x99,0x9e,0x87,0x8e,0x92,0x97,0x9c,0x96,
/* 6 */   0xcb,0x83,0x49,0x4f,0x55,0xcc,0xcd,0x59,0x41,0x45,0x45,0x49,0x4f,0x55,0x59,0x43,
/* 7 */   0x80,0x45,0x49,0x85,0x86,0xcb,0x83,0x49,0xcd,0x86,0x81,0x83,0x49,0xcd,0x86,0x84,
/* 8 */   0x61,0x62,0x63,0x64,0x65,0x66,0x67,0x68,0x69,0x6a,0x6b,0x6c,0x6d,0x6e,0x6f,0x70,
/* 9 */   0x71,0x72,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7a,0xbe,0xbf,0x8c,0x8d,0x3b,0x2a,
/* a */   0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4a,0x4b,0x4c,0x4d,0x4e,0x4f,0x50,
/* b */   0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58,0x59,0x5a,0xae,0xaf,0x81,0x8d,0x3b,0x2a,
/* c */   0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,
/* d */   0xe0,0xe1,0xe2,0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0xea,0xeb,0xec,0xed,0xee,0xef,
/* e */   0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,
/* f */   0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff
};

/*-------------------------------------------------------------------------------------------*/
/* Map_key
 *  Depending on keyboard, maps to 3270 Typewriter Scan Codes.
 */
/*-------------------------------------------------------------------------------------------*/

WORD Map_Key(kbtype,code,modifier)
    BYTE kbtype;
    BYTE code;
    WORD modifier;
{
    WORD outp;

    if (kbtype == T75) {   /* map MAC II -> 87 keyboard */
        if (modifier & controlKey) {
            if (code == 0x00) {   /* CTL-A -> ATTN key */
                outp = 0x0050;
                if (modifier & optionKey)
                    outp |= (ALT_SHIFT << 8);
                return outp;
```

```
            }
        if (code == 0x0F) {   /* CTL-R -> RESET key */
            outp = 0x0034;
            if (modifier & optionKey)
                outp |= (ALT_SHIFT << 8);
            return outp;
            }
        }
    outp = M2KB[code];    /* key -> 327x key */
    }
else {                              /* map ENHANCED -> 87 keyboard */
    outp = MACENC[code];
    }
if (!outp) {                /* handle modifiers */
    if (modifier & optionKey)
        outp |= (ALT_SHIFT << 8);
    if ((modifier & shiftKey) || (modifier & alphaLock))
        outp |= (UP_SHIFT << 8);
    }
return outp;
}

/*-------------------------------------------------------------------------------------------------*
/* StXlate
 * translates the OIA characters for output.
 */
/*-------------------------------------------------------------------------------------------------*/
void StXlate(bptr,len)
    BYTE *bptr;
    BYTE len;
{
    BYTE i;

    for (i=0; i<len; i++)
        *(bptr+i) = sl_tbl[(*(bptr+i))];
}
```

Chapter 3

API Service Requests

Documentation format of each API call

This chapter contains the specifications for all of the API calls. The calls are described in alphabetical order by call name, with each call beginning on a new page with the name of the call at the top.

❖ *Note*: For a functional grouping of the calls, see Table 1-X in Chapter 1 of this manual.

The description of each call contains the following categories in the following format:

Purpose This section defines the intent of the call.

Format This section shows the structure of the call in the following format:

```
API_call (&req_block, asyncFlag);
```

Parameters This section lists the parameters and their types, and includes a comment about whether the parameter is passed, returned, or passed and returned, as shown here:

```
TYPE    *firstParm              /* passed */
TYPE    second_parm;            /* returned */
```

Some calls use a `modifiers` parameter. Such calls, if you don't specify any of the supplied constants, have a default mode of operation. If the default mode doesn't suit the needs of the application, the application can use the supplied constants to tailor a call to its requirements.

For example, Disconnect_From_PS normally breaks the logical connection between the application and a presentation space and terminates the underlying host session that supports the presentation space. However, if you supply the `DP_KEEP_SESSION` constant to the `modifiers` parameter, the driver breaks the logical connection but does not terminate the underlying host session.

A particular driver may support none, some, or all of the modifier values. If a request has an unsupported modifier set, the driver returns `MOD_UNSUPP_ERR`.

❖ *Note*: The API Get_Host_Connection_Info call returns information about the calls and modifiers supported by a particular connection method.

Definitions This section defines each of the parameters.

Description This section provides any additional description not covered by the purpose (some calls do not require additional description).

Example This section, when present, provides an example of the call in the following format:

```
req_blk.api_req.ref_con = API_REQ_CONNECT_SESS;
last_request = API_REQ_CONNECT_SESS;
req_blk.api_req.sess_id = -1;
if (err = Connect_Session(&req_blk)) {
      ErrorMessage("Connect_Session Error",err);
      return 0;
      }
Do_Get = FALSE;
Do_Disp = FALSE;
return 1;
```

The example is taken from the sample terminal-emulation application given in Chapter 2.

Errors This section lists the most notable errors that the interface code or the driver can return, in the following format:

NAME_OF_ERR Description of the error as it applies to the call

Errors that can occur for all calls—such as invalid connection ID, PS ID, or parameter values—aren't listed.

Activate_Prt_Sess

Purpose

This 3270 API call instructs the driver to allocate a connection to a printer session and make that session available to the host.

❖ *CUT note:* A CUT driver cannot support this call.

If an underlying host session exists at the time this call is issued, and the `APS_KEEP_SESSION` modifier is specified, the driver ignores all of the passed parameters except `ret_ps_id` and retains the printer session parameters in effect prior to the call.

Format

```
Activate_Prt_Sess (&req_block, asyncflag);
```

Parameters

```
BYTE    prt_type;              /* passed */
WORD    prtbuf_size;           /* passed */
BYTE    *query_replyp;         /* passed */
WORD    query_reply_len;       /* passed */
BYTE    *ebc_dbc_tabp;         /* passed */
BYTE    modifiers;             /* passed */

BYTE    ret_ps_id;             /* returned */

/* modifiers */
#define APS_KEEP_SESSION       0x01
```

Definitions

asyncFlag

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC — Specify this constant to return control immediately to the caller.

SYNC — Specify this constant to prevent control from returning until the request completes.

ps_id

For this call, this parameter in the request header specifies the printer session ID rather than the presentation space ID.

If you want your application to connect to any available printer session rather than to a specific session, place 0xFF in `ps_id`. The driver returns the ID—in the `ret_ps_id` parameter—of a specific printer session that can support at least one of the types of printer data specified in the `prt_type` parameter.

To connect to a particular printer session, place a specific printer session's ID in `ps_id`. If the call is successful, the driver duplicates the printer session ID in `ret_ps_id`.

`prt_type`

This BYTE normally specifies the type of printer data that the application can support, as follows:

`APS_DSC`	DSC support
`APS_LU1`	LU type 1 support (SCS or SCS/IPDS support)
`APS_DSC_OR_LU1`	Both DSC and LU 1 support.

However, if a driver is employing the SNA host protocol, `prt_type` specifies what type of Bind the driver will accept, as follows:

`APS_DSC`	LU type 3 bind
`APS_LU1`	LU type 1 bind
`APS_DSC_OR_LU1`	Either LU type 1 or LU type 3 bind

`prtbuf_size`

For DSC mode, this WORD specifies the size of the print buffer that the application can support.

For SCS mode, this WORD specifies to the driver the size of the buffer the driver should create to manage incoming data from the host. For this mode, the buffer size affects the maximum RU size and pacing count that can be accepted by the driver in the Bind.

For both modes, the buffer size should be at least 4K to allow full emulation of a 3287printer or a 3289 printer. The maximum size that the application can specify is limited by the driver's buffer capacity returned in the `max_prtbuf_size` parameter of a Get_Host_Connection_Info call.

`*query_replyp`

This POINTER points to a buffer that contains Query Reply structured fields indicating the features the session can support. The Query Reply structured fields must be contiguous in the buffer. The driver supplies Query Reply (Null), Query Reply (Summary), or both as needed when a Read Partition (Query) structured field is received. If your application uses Query Reply fields, it must supply all other Query Replys.

If you do not want to provide Query Reply fields, pass a NIL pointer for this parameter. The driver then formats Query Reply fields for Color, Highlighting, Implicit Partition, Usable Area, and Character Set for DSC sessions only. No Query Reply fields are formatted for LU1 sessions.

If your application supports SCS or IPDA data, the application must format the appropriate query replys and send them to the host by way of a Send_Prt_Data call.

query_reply_len

This WORD specifies the length, in bytes, of the Query Reply structured fields.

*ebc_dbc_tabp

This POINTER points to a table that translates EBCDIC values to 3270 device buffer codes. This parameter applies only when the Prt_Type parameter is set to one of the constants that indicate that DSC is supported.

The table is used to translate EBCDIC code received from the host application to device buffer codes. The driver uses this table to maintain an image of the printer buffer in DBC format.

The 256-byte array is indexed by an EBCDIC value. Each array element contains a 3270 device buffer code point falling in the range 0x00 through 0xBF. (The range 0xC0 through 0xFF is reserved for attributes.)

modifiers

This BYTE contains the modifiers for the Activate_Prt_Sess call, as follows:

APS_KEEP_SESSION
: Specify this constant to instruct the driver to retain the current host session if it exists and to ignore all other parameters specified in this call. If you don't specify this constant, the host session is terminated (if it exists) before being re-established.

ret_ps_id

This BYTE returns the specific printer session ID.

Description

For the types and formats of the Query Reply s to be sent to the host, see the *3270 Data Stream Programmer's Reference* (GA23-0059) published by IBM.

If the driver is employing the SNA host protocol, the following notes apply:

For a DSC Bind,you can specify a print buffer larger than 4K. Such a buffer is useful for supporting host applications that are not display-oriented and can take full advantage of a larger buffer size to increase throughput. For more details about the relationship between the bind and print buffer use, see the *3274 Description and Programmer's Guide* (GA23-0061) published by IBM.

For an SCS Bind, the smaller the buffer, fewer and/or smaller RUs that can be stored and the longer the driver must wait to send a pacing response. A 4K buffer is recommended as the minimum; a 32K buffer is more than necessary. Refer to the *IBM 3174 Functional Description (GA#???)* for more information.

Errors

None

Check_Session_Bind

Purpose

This 3270 API call allows an application to check whether a host session exists, and to wait for the session to be established if the session doesn't exist.

Important Check_Session_Bind applies only to SNA connections.

If the LU session is type 2 or 3, the call also indicates the sizes of the default and the alternate presentation spaces.

Format

```
Check_Session_Bind (&req_block, asyncFlag);
```

Parameters

```
LONG    wait_time;              /* passed */

BYTE    lu_type;                /* returned */
WORD    default_ps_size;        /* returned */
WORD    alt_ps_size;            /* returned */
```

Definitions

asyncFlag

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC Specify this constant to return control immediately to the caller.

SYNC Specify this constant to prevent control from returning until the request completes.

ps_id

This parameter in the request header specifies the PS or the printer session ID.

wait_time

This LONG parameter specifies the maximum amount of time that the driver should wait for PLU-SLU session establishment. If the timeout period expires, the driver returns TIMEOUT_ERR.

The value passed represents a number of 100-millisecond ticks. There are two special values, as follows:

0xFFFFFFFF	Specify this value to instruct the driver to wait forever; that is, the driver will never return a `TIMEOUT_ERR`.
0	Specify this value to instruct the driver to return `TIMEOUT_ERR` immediately if the PLU-SLU session doesn't exist at the time of the call.

lu_type

This BYTE returns the LU type of the current session as follows:

`WSB_LU1`	Indicates that the current session is an LU type 1 session.
`WSB_LU2`	Indicates that the current session is an LU type 2 session.
`WSB_LU3`	Indicates that the current session is an LU type 3 session.

default_ps_size

This WORD IS valid only if `lu_type` is set to the `WSB_LU2` or `WSB_LU3` constant. The parameter contains the default presentation space dimensions passed when the session was bound. The number of rows is in the high-order byte; the number of columns is in the low-order byte.

If `lu_type` is set to `WSB_LU3`, and this parameter and the `alt_ps_size` parameter are both 0, the host is indicating that the printer (that is, the application) should use the full capacity of its print buffer. The capacity of the print buffer is specified in the `prtbut_size` parameter of the Activate_Prt_Sess call.

alt_ps_size

This WORD IS valid only if `lu_type` is set to the `WSB_LU2` or `WSB_LU3` constant. The parameter contains the alternate presentation-space dimensions passed in the Bind. The number of rows is in the high-order byte; the number of columns is in the low-order byte. If `lu_type` is set to `WSB_LU3`, and this parameter and the `default_ps_size` parameter are both 0, the host is indicating that the printer (that is, the application) should use the full capacity of its print buffer. The capacity of the print buffer is specified in the `prtbut_size` parameter of the Activate_Prt_Sess call.

Description

If the underlying host protocol is SNA, an application can issue this call to verify that a host session exists before it issues calls to retrieve printer data.

Errors

`TIMEOUT_ERR`	This error indicates that the time specified in `wait_time` was exceeded before the session was established.

Close_Host_Connection

Purpose

This 3270 API call closes a connection method. Issue the call when your application no longer requires the services of a particular connection method.

This call immediately terminates the connection method, and any held requests—such as Get_Update—are discarded.

Format

```
Close_Host_Connection (&req_block, asyncFlag);
```

Parameters

None

Definitions

`asyncFlag`

This flag is ignored for this call.

Description

The system reads the `conn_id` parameter in the specified `&req_block` to determine which driver and connection method to close.

Example

The following example, taken from the sample application presented in Chapter 2 of this guide, shows a statement that closes the host connection if an error is made in the Connect_To_PS call.

```
if (err = Connect_To_PS(&(dft[session]->req_blk),ASYNC)) {
   ErrorMessage("Connect_To_PS Error",err);
   Close_Host_Connection(&dft[session]->req_blk);
   Term_3270_API(api_vars);
   return FALSE;
   }
```

Errors

None

Connect_To_PS

Purpose

This 3270 API call requests a presentation-space ID so that an application can establish a logical connection to a presentation space. The call also provides the application with the option to retain an existing underlying host session and its parameters.

Presentation space IDs (`ps_id`) range from 1 to the maximum number of presentation spaces supported by a particular connection method. You can get the IDs of the available presentation spaces by using an Open_Host_Connection call or a Get_Host_Connection_Info call.

❖ *DFT-CU note:* If the host connection method is DFT-based or CU-based, you must pass additional translation tables as parameters to configure the underlying presentation-services component. These parameters allow an application written for a CUT emulation to be used transparently in a DFT- or CU-emulation environment.

If an underlying host session exists at the time this call is issued and the `modifiers` parameter specifies the `CP_KEEP_SESSION` constant, all the passed parameters except the `ps_id` parameter are ignored. The session parameters in effect prior to the call are retained.

Format

```
Connect_To_PS (&req_blk, asyncFlag);
```

Parameters

```
BYTE    *asc_dbc_tabp;              /* passed */
BYTE    *dbc_asc_tabp;              /* passed */
BYTE    color_supp;                 /* passed */
BYTE    modifiers;                  /* passed */

/* The following passed parameters apply only to DFT or CU */
struct  ktab_rec *keybd_tabp;       /* passed - DFT or CU only*/
WORD    *keybd_tab_len;             /* passed - DFT or CU only */
BYTE    *dbc_ebc_tabp;              /* passed - DFT or CU only */
BYTE    *ebc_dbc_tabp;              /* passed - DFT or CU only */
BYTE    *query-replyp;              /* passed - DFT or CU only */
WORD    query-reply_len;            /* passed - DFT or CU only */
BYTE    *type_pass_datap;           /* passed - DFT or CU only  */
WORD    type_pass__data_len;        /* passed - DFT or CU only */
BYTE    scrn_emul;                  /* passed - DFT or CU only */
BYTE    num_lock;                   /* passed - DFT or CU only */

BYTE    ret_ps_id;                  /* returned */
```

```
/* modifiers */
#define CP_KEEP_SESSION          0x01

struct ktab_rec
{
        BYTE descriptor;
        BYTE shift_code;
        BYTE scan_code;
        BYTE value;
};
```

Definitions

conn_id

This field appears in the request header, and specifies the connection method to be used for this connection.

port_id

This field appears in the request header, and specifies the ID of the port to be used for this connection.

ps_id

This field appears in the request header. If you want your application to connect to any of the available presentation spaces, place 0xFF in ps_id. Connect_To_PS then returns a specific presentation-space ID in ret_ps_id.

If you want your application to connect to a specific presentation space, place the ID of that space in ps_id. Presentation-space IDs (ps_id) can be from 1 to the maximum number of presentation spaces supported by a particular connection method. If the call is successful, Connect_To_PS then duplicates the specified presentation-space ID in ret_ps_id.

asyncFlag

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC Specify this constant to return control immediately to the caller.

SYNC Specify this constant to prevent control from returning until the request completes.

*asc_dbc_tabp

This POINTER points to a table that translates Macintosh ASCII into 3270 device-buffer codes. The driver uses this table to map Macintosh ASCII codes received in an API request. The 256-byte array is indexed by an ASCII value. Each array element contains a 3270 device- buffer code point in the range 0x00 through 0xBF. If you want your application to send only device-buffer codes to the driver, set this pointer to NIL to indicate that no translation is required.

*dbc_asc_tabp

This POINTER points to a table that the driver uses to convert 3270 device-buffer codes into Macintosh ASCII codes before presenting the codes to the application. The 256-byte array is indexed by a device buffer code (index values 0x00 through 0xBF are data code points; 0xC0 through 0xFF are attribute code points). Each array element contains an ASCII code point. If you want your application to send only device buffer codes to the driver, set this pointer to NIL to indicate that no translation is required.

color_supp

This BYTE specifies the type of color support that the application needs. The type of color support affects how the driver sets the color bits in the DAB.

❖ *Note:* If the presentation space is unformatted, the color returned by the driver is always green unless the CP_NO_COLOR constant is specified.

You can specify the following color support modes:

CP_NO_COLOR Specify this constant to always set the DAB color bits so that they do not support color (x'000').

CP_2_COLOR Specify this constant for two base colors and no extended colors. The driver examines only the field attribute to determine the color setting for the DAB. The driver returns CK_WHITE if the intensified bit is set in the field attribute; if the bit is not set, the driver returns CK_GREEN.

CP_4_COLOR Specify this constant for four base colors and no extended colors. The driver examines only the field attribute to determine the color setting for the DAB, and returns one of the following colors:

CK_GREEN	Unprotected, normal intensity
CK_RED	Unprotected, intensified
CK_BLUE	Protected, normal intensity
CK_WHITE	Protected, intensified

CP_2_COLOR_EXT

Specify this constant to support extended colors with two base colors.

The color setting in the DAB is a copy of the EAB color setting with this exception: When extended color is in effect (that is, when the base color override bit is set to 1) and when the color bits in the EAB are set to the default values, the driver examines the field attribute and sets the DAB to either white for intensified fields or green for non-intensified fields. When base color is in effect (that is, when the base color override bit has been reset to 0), the driver ignores the EAB and sets only white and green, in the same fashion as for CP_2_COLOR.

CP_4_COLOR_EXT

Specify this constant to support extended colors with four base colors. Doing this causes much of the same behavior as CP_2_COLOR_EXT except that, when base color is in effect (that is, when the base color override bit has been reset), colors are set in the same fashion as for CP_4_COLOR.

modifiers

This BYTE contains the modifiers for the Connect_To_PS call, as follows:

CP_KEEP_SESSION

Specify this constant to instruct the driver to retain the current host session if it exists and to ignore all parameters except ps_id in the next Connect _To_PS call. If you don't specify this constant, the host session is terminated (if it exists) before being re-established.

***keybd_tab**

This POINTER points to a keyboard translation array.

❖ *CUT note:* CUT drivers ignore this parameter.

Each element in the variable-length array must contain a value specifying the following information:

descriptor This BYTE identifies the type of key as one of the following:

KT_REGULAR	0x00	Text or numeric character
KT_CONTROL	0x01	Control key
KT_DEAD_KEY	0x02	Dead key

KT_DEAD_KEY_TERM	0x03	Dead-key terminator
KT_APL_TEXT	0x04	APL key

Specify KT_DEAD_KEY or KT_DEAD_KEY_TERM only if the keyboard type being emulated supports dead keys. See the description of this call for more details about the descriptor records.

shift_code This BYTE can be one of the following:

KT_NO_SHIFT	0x00
KT_UP_SHIFT	0x01
KT_ALT_SHIFT	0x02

scan_code This BYTE contains the scan code for the keyboard type being emulated. Do not define scan codes for the shift keys Shift Lock (Caps Lock), Left Shift, Right Shift, and Alt Shift; shift_code provides the scan code definitions.

value In the case of KT_REGULAR, KT_DEAD_KEY, and KT_DEAD_KEY_TERM descriptor records, this BYTE is a displayable EBCDIC code value. In the case of a KT_CONTROL descriptor record, value identifies a particular 3270 control key. See Appendix B in this guide for a table of control-key values.

keybd_tab_len

This WORD specifies the length, in bytes, of keybd_tab.

❖ *CUT note:* CUT drivers ignore this parameter.

***dbc_ebc_tabp**

This POINTER points to a table that translates 3270 device-buffer codes to EBCDIC values.

❖ *CUT note:* CUT drivers ignore this parameter.

If the driver is maintaining the PS in EBCDIC, the driver uses the specified table to translate device-buffer codes received from an application request or through the ASCII-to-DBC table. If the driver is maintaining the PS in DBC format, the driver uses the specified table when data is transmitted to the host. The 192-byte array is indexed by a 3270 device-buffer code point (0x00 through 0xBF). Each array element contains an EBCDIC code point.

*ebc_dbc_tabp

This POINTER points to a table that translates EBCDIC values into 3270 device buffer codes.

❖ *CUT note:* CUT drivers ignore this parameter.

The driver uses the table to translate EBCDIC code received from the host and keyboard scan codes received from an application into device-buffer codes. The 256-byte array is indexed by an EBCDIC value. Each array element contains a 3270 device buffer code point falling in the range 0x00 through 0xBF. (The range 0xC0 through 0xFF is reserved for attributes.)

*query_replyp

This POINTER points to a buffer containing Query Reply structured fields that indicate the featues the application can support.

❖ *CUT note:* CUT drivers ignore this parameter.

The Query Reply structured fields must be contiguous in the buffer. If your application uses Query Reply fields, the driver supplies Query Reply (Null), Query Reply (Summary), or both as needed when a Read Partition (Query) structured field is received. Your application must supply all other Query Reply fields.

If you do not want to provide Query Reply fields, pass a NIL pointer for this parameter. This technique allows the driver to use configuration information from the Open_Host_Connection call and other information in this call to format Query Reply fields for color highlighting, implicit partition, reply mode, usable area, and character set.

query_reply_len

This WORD specifies the length, in bytes, of the Query Reply structured fields.

❖ *CUT note:* CUT drivers ignore this parameter.

*type_pass_datap

The POINTER points to a number of structured field descriptors, with each descriptor structured as follows:

Byte 2 reply/no-reply flag

Byte 1 High-order byte of structured field ID or 0xFF

Byte 0 Low-order byte of structured field ID

❖ *CUT note:* CUT drivers ignore this parameter.

This POINTER applies only if the driver supports the Get_Passthru_Data call. If the driver doesn't support the Get_Passthru_Data call, it ignores the pointer. Set this parameter to NIL if your application doesn't issue Get_Passthru_Data calls.

The array is terminated with an 0xFFFF. Structured field IDs that are 2 bytes long occupy bytes 1 and 0. Structured field IDs that are 1 byte long have 0xFF in byte 1 with the ID in byte 0.

Byte 2 indicates if a response is required from the application to the structured field. Certain structured fields do not require a response at the host communication level; any detected errors are dealt with at the application level (for example, D0 structured fields). If so, byte 2 should be set to the value of the constant `CP_NO_REPLY`.

Other structured fields requiring a response at the the host communication level (that is, Load PS) should have the byte set to the `CP_REPLY` constant. Based upon this byte, the driver sets the `data_end` parameter to `GPD_END` or `GPD_END_REPLY` in the Get_Passthru_Data call.

When the driver receives a structured field with an ID that matches one of the IDs in the array, it buffers the structured field. The structured field is posted to the application when it issues a Get_Passthru_Data call or passed immediately if a Get_Passthru_Data call is outstanding. If the structured field requires a reply, the driver is suspended from any further processing on the session until the application issues a Post_Passthru_Reply.

You should specify those structured fields in this array which the driver cannot process, but which are the application's responsibility. Thus, the array is intended to provide your application access to special structured fields, such as D0 structured fields (to support IND$FILE file transfer) or APA structured fields (to support vector graphics support). Normally, your application should entrust the processing of the other structured fields to the driver. Refer to the driver's documentation to find out what structured fields it can support.

If an application specifies the destination/origin structured field (0x0F02) in the array, the following applies:

- When the driver receives a destination/origin structured field whose ID is other than 0 (the value associated with the display), it forwards the destination/origin structured field to the application and all structured fields that follow, whether or not their types are specified in the array, until the destination reverts to the display. At that time, only specified structured fields types are again forwarded to the application.
- To allow access to the INCTRL field, all destination/origin structured fields will be passed to the application, whether the ID field is set to the base display (0x0000) or to a particular destination.

`type_pass_data_len`

This WORD specifies the length, in bytes, of the data pointed to by the `*type_pass_datap` parameter. You can obtain the number of structured field descriptors by dividing this parameter by three.

❖ *CUT note:* CUT drivers ignore this parameter.

`scrn_emul`

This BYTE specifies the screen size of the 3278 terminal that the application is emulating, as follows:

Model	Constant	Default screen size	Alternate screen size
Model 2	`CP_MOD_2`	1920	1920
Model 3	`CP_MOD_3`	1920	2560
Model 4	`CP_MOD_4`	1920	3440
Model 5	`CP_MOD_5`	1920	3564

❖ *CUT note:* CUT drivers ignore this parameter.

`num_lock`

If set to TRUE, this BYTE instructs the driver to support **numeric lock.**

❖ *CUT note:* CUT drivers ignore this parameter.

If `num_lock` is TRUE, and the application attempts to write data other than the characters 0 through 9, decimal sign, minus sign, or DUP into a numeric field, the driver displays an input-inhibited condition of X-NUM in the OIA.

If set to FALSE, this BYTE disables numeric lock.

`ret_ps_id`

This returned BYTE identifies the PS reserved for a particular host session.

Description

Scan codes associated with regular, dead key, and dead key terminator descriptors should map to displayable EBCDIC code points. You can use regular scan codes for normal text and numeric characters.

❖ *DFT note:* For a DFT connection, the presentation-space ID (`ps_id`) returned by the Connect_To_PS call always maps to the same underlying logical terminal; that is, PS ID 1 maps to logical terminal 1, PS ID 2 maps to logical terminal 2, and so on.

Two descriptor types—dead key and dead-key terminator—support the use of dead keys. The dead key descriptor record identifies the dead key scan code and its EBCDIC value. Immediately following this record is at least one dead-key terminator record. The dead-key terminator records inform the driver of each legal scan code that can be combined with the preceding dead-key scan code. Each dead-key terminator value gives the EBCDIC value of the composite character. See *IBM 3270 Information Display System Character Set Reference* (GA27-2837) for further information.

Scan codes associated with control keys map to special encoded values which identify 3270 control keys. The scan codes and keys associated with them are shown in Appendix B of this guide.

Example

The following example, taken from the sample application presented in Chapter 2 of this guide, sets up the API request block and issues the Connect_To_PS call.

```
dft[session]->req_blk.api_vars = api_vars;
dft[session]->req_blk.net_addr.aNode = 0;
dft[session]->req_blk.port_id = Slot;
dft[session]->req_blk.io_compl = nil;
dft[session]->req_blk.conn_id = saved_conn_id;
dft[session]->req_blk.ps_id = 0xFF;
dft[session]->req_blk.connps.keybd_tabp = ktab;
dft[session]->req_blk.connps.keybd_tab_len = sizeof(ktab);
dft[session]->req_blk.connps.dbc_ebc_tabp = dbc_ebc;
dft[session]->req_blk.connps.ebc_dbc_tabp = ebc_dbc;
dft[session]->req_blk.connps.dbc_asc_tabp = dbc_asc;
dft[session]->req_blk.connps.asc_dbc_tabp = asc_dbc;
dft[session]->req_blk.connps.color_supp = CP_4_COLOR_EXT;
dft[session]->req_blk.connps.num_lock = FALSE;
dft[session]->req_blk.connps.scrn_emul = CP_MOD_2;
dft[session]->req_blk.connps.query_reply_len = 0;
dft[session]->req_blk.connps.query_replyp = nil;
dft[session]->req_blk.connps.type_pass_data_len = 0;
dft[session]->req_blk.connps.type_pass_datap = nil;
dft[session]->req_blk.connps.modifiers = NO_MODS;

/* ErrorMessage("Connect_To_PS session",session); */
/* junk = &(dft[session]->req_blk); */
/* Debugger();*/

if (err = Connect_To_PS(&(dft[session]->req_blk),ASYNC)) {
   ErrorMessage("Connect_To_PS Error",err);
   Close_Host_Connection(&dft[session]->req_blk);
   Term_3270_API(api_vars);
   return FALSE;
   }
dft[session]->last_request = RC_CONNECT_TO_PS;
return TRUE;
```

Errors

`PS_UNSUPP_ERR`	This error indicates that a logical terminal does not support the presentation space or that the specified `ps_id` was not in the range of valid IDs.
`PS_UNAVAIL_ERR`	If 0xFF was passed in `ps_id`, this error indicates that no more presentation spaces are available. If a specific ID was passed, this error indicates that the caller (or another application) has already established a connection with that PS.

Copy_From_Buffer

Purpose

This 3270 API call copies all or a portion of the PS, DAB, DABE, or EAB into the application's corresponding destination buffers. Characters from the PS are normally presented to the application in ASCII format; however, you have the option of receiving them in DBC format by using the `CFB_NO_TRANS` constant in the `modifiers` parameter.

Format

```
Copy_From_Buffer (&req_blk, asyncFlag);
```

Parameters

```
BYTE     *ps_bufp;              /* passed */
BYTE     *dab_bufp;             /* passed */
BYTE     *dabe_bufp;            /* passed */
BYTE     *eab_bufp;             /* passed */
WORD     dest_offset;           /* passed */
WORD     num_bytes_to_copy;     /* passed */
WORD     src_offset;            /* passed */
BYTE     modifiers;             /* passed */

WORD     num_bytes_copied;      /* returned*/

/* modifiers */
#define CFB_WRAP                0x01
#define CFB_NO_TRANS            0x02
```

Definitions

`asyncFlag`

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

`ASYNC` Specify this constant to return control immediately to the caller.

`SYNC` Specify this constant to prevent control from returning until the request completes.

`*ps_bufp`

This POINTER points to an application buffer designated to receive data from the PS. Set this parameter to NIL if you do not intend to copy the PS.

***dab_bufp**

This POINTER points to an application buffer designated to receive data from the Display Attribute Buffer. Set this parameter to NIL if you do not intend to copy the DAB.

***dabe_bufp**

This POINTER points to an application buffer designated to receive data from the Extended Display Attribute Buffer. Set this parameter to NIL if you do not intend to copy the DABE.

***eab_bufp**

This POINTER points to an application buffer designated to receive data from the Extended Attribute Buffer. Set this parameter to NIL if you do not intend to copy the EAB.

dest_offset

This WORD specifies the offset for the application buffers pointed to by the `*ps_bufp`, `*dab_bufp`, `*dabe_bufp`, and `*eab_bufp` parameters. The driver begins writing data to these destination buffers at the location indicated by the offset.

src_offset

This WORD specifies the offset in the source buffer at which point the driver begins transferring data. This value cannot exceed the size of the PS minus 1.

num_bytes_to_copy

This WORD specifies the number of bytes to be copied into one or more of the application's buffers. This number applies to each buffer for which a pointer is supplied; that is, the driver will copy the same number of bytes into each destination buffer specified. This number cannot exceed the size of the PS or, for that matter, the DAB or the EAB, which are the same size as the PS.

modifiers

This BYTE contains the modifiers for the Copy_From_Buffer call, as follows:

`CFB_WRAP`	Specify this value to cause the copy operation to wrap to the beginning of the PS if the operation encounters the end of the PS before it finishes copying bytes from the source buffer.

`CFB_NO_TRANS` Normally, the driver translates 3270 device-buffer codes into ASCII characters using the table pointed to by the `*dbc_asc_tabp` parameter of the Connect_To_PS call. However, if you specify `CFB_NO_TRANS`, the driver does not perform the translation. This option allows your application to receive device-buffer codes in a PS destination buffer. This applies only to the destination PS buffer and has no bearing upon the destination DAB or EAB buffers.

`num_bytes_copied`

This WORD returns the number of bytes that were copied into each buffer.

Errors

`INP_INHIBITED_ERR`	This error indicates that the copy operation was completed but that an input-inhibited condition was present.

Copy_From_Field

Purpose

This 3270 API call allows an application to copy a field from the PS to an application-defined data area. Characters from the PS are normally presented to the application in ASCII format; however, you have the option of receiving them in DBC format by using the `CFF_NO_TRANS` constant in the `modifiers` parameter.

Format

```
Copy_From_Field (&req_blk, asyncFlag);
```

Parameters

```
WORD     ps_offset;              /* passed */
BYTE     *dest_bufp;             /* passed */
WORD     max_bytes_to_copy;      /* passed */
BYTE     *modifiers;             /* passed */

WORD     num_bytes_copied;       /* returned */

/* modifiers */
#define CFF_NO_TRANS            0x01
```

Definitions

`asyncFlag`

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

`ASYNC` Specify this constant to return control immediately to the caller.

`SYNC` Specify this constant to prevent control from returning to the caller until the request completes.

`ps_offset`

This WORD specifies a location in the PS where the driver begins to transfer data.

`*dest_bufp`

This POINTER points to a buffer that receives the field copied from the PS.

`max_bytes_to_copy`

This WORD specifies the maximum number of bytes that the driver can copy into the application's buffer. If the length of the field exceeds the value of this parameter, the driver returns the error `DATA_XFER_TRUNC_ERR`.

modifiers

This BYTE contains the modifiers for the Copy_From_Field call, as follows:

CFF_NO_TRANS Normally, the driver translates 3270 device buffer codes into ASCII characters using the table pointed to by the *dbc_asc_tabp parameter of the Connect_To_PS call. However, if you specify CFF_NO_TRANS, the driver does not perform the translation. This option allows your application to receive device buffer codes in a PS destination buffer. This applies only to the destination PS buffer and has no bearing upon the destination DAB or EAB buffers.

num_bytes_copied

This WORD returns the number of bytes that the driver copied before the call terminated.

Description

The copy begins at the location specified in ps_offset and stops at the end of the field. If ps_offset is positioned on an attribute byte, the driver copies the attribute byte through the end of the field. The driver returns an error message if it reaches the end of the PS or the end of the destination buffer before it finishes copying a field. Issue a Find_Field call to ascertain a field's starting point and length.

If the copy operation terminates normally at the end of the field, the driver returns NO_ERR.

Errors

DATA_XFER_TRUNC_ERR	This error indicates that the driver encountered the end of the application's destination buffer before the driver finished copying a field from the PS.
END_OF_PS_ERR	This error indicates that the driver encountered the end of the PS before it finished copying a field.
INP_INHIBITED_ERR	This error indicates that the copy operation completed, but that the input-inhibited condition was present.
PS_UNFMT_ERR	This error indicates that the PS is currently unformatted, so no fields exist.

Copy_OIA

Purpose

This 3270 API call obtains an untranslated copy of the operator information area (OIA). The call can also obtain the corresponding EAB image of the OIA.

Format

```
Copy_OIA (&req_blk, asyncFlag);
```

Parameters

```
BYTE     *oia_bufp;              /* passed */
BYTE     *oia_eabp;              /* passed */
BYTE     modifiers;              /* passed */

/* modifiers */
#define CO_GET_GRP_INDS          0x01
```

Definitions

asyncFlag

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC Specify this constant to return control immediately to the caller.

SYNC Specify this constant to prevent control from returning to the caller until the request completes.

***oia_bufp**

This POINTER points to the buffer designated to receive the image of the OIA. This buffer must be at least 80 bytes long. If you also want the OIA group indicators to be returned, the buffer must be 122 bytes long. Set this pointer to NIL to prevent the copying of the OIA (and optional group indicators). See "The Presentation Space" in Chapter 1 for more information about group indicators.

***oia_eabp**

This POINTER points to the buffer designated to receive the EAB image of the OIA. This buffer must be 80 bytes long. Set this pointer to 0 to NIL to prevent the copying of the EAB image.

modifiers

This BYTE contains the modifiers for the Copy_To_OIA call, as follows:

CO_GET_GRP_INDS
: Specify this option to copy the OIA group indicators. The driver returns the indicators in 42 bytes that immediately follow the 80-byte OIA image pointed to by the `*oia_bufp` parameter.

Errors None

Copy_To_Field

Purpose

This 3270 API call copies a string of data into a field in the PS. ASCII characters supplied by the application are normally translated into DBC format; however, if you prefer, you can also write DBC codes directly by setting the `CTF_NO_TRANS` constant as described under the `modifiers` parameter.

Format

```
Copy_To_Field (&req_blk, asyncFlag);
```

Parameters

```
BYTE    *strp;                  /* passed */
WORD    num_bytes_to_copy;      /* passed */
WORD    ps_offset;              /* passed */
BYTE    *modifiers;             /* passed */

WORD    num_bytes_copied;       /* returned */

/* modifiers */
#define CTF_COPY_MULT           0x01
#define CTF_NO_TRANS            0x02
```

Definitions

`asyncFlag`

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC — Specify this constant to return control immediately to the caller.

SYNC — Specify this constant to prevent control from returning to the caller until the request completes.

`*strp`

This POINTER points to the source buffer containing the string that the driver will copy into the PS.

`num_bytes_to_copy`

This WORD specifies the number of bytes to be copied from the application's source buffer.

`ps_offset`

This WORD specifies a location in the PS where the driver begins writing data. The location is where the copy should begin; that position cannot be part of the attribute byte.

modifiers

This BYTE contains the modifiers for the Copy_To_Field call, as follows:

CTF_COPY_MULT
: This modifier allows a string to be dispersed into a number of unprotected fields, beginning with the current field. The driver copies bytes from the string buffer into contiguous unprotected fields in the PS. The driver ignores autoskip fields (protected and numeric bits set) in the midst of these unprotected fields.

  The initial field to which `ps_offset` is positioned must be an unprotected field or the driver will immediately return the error `WRITE_PROT_FLD_ERR`. The copy operation terminates when string data runs out or when the driver encounters a non-autoskip protected field, in which casethe driver returns the error `WRITE_PROT_FLD_ERR`.

CTF_NO_TRANS
: Normally, the driver translates characters in the source buffer to 3270 DBC format by using the translation table pointed to by the `*asc_dbc_tabp` parameter in the Connect_To_PS call. By setting the `modifiers` parameter to the value of `CTF_NO_TRANS`, you instruct the driver to not translate the codes. This allows your application to copy codes in DBC format directly into fields.

num_bytes_copied

This WORD returns the number of bytes copied to the PS before the call terminated.

Description

The copy operation begins at the location specified in the `ps_offset` parameter and stops at the end of the field. The driver returns an error message if it reaches either the end of the source buffer or the end of the PS before it finishes copying or writing a field. Attempts to write data into a protected field or the wrong type of data into a field also generate errors.

The control unit must receive an AID key before it will transmit changes in the PS to the host. Use the Send_Keys request to send an AID scan code.

The passed offset identifies where the copy operation should begin in the field. The offset cannot be positioned on the attribute byte.

Changes to unprotected fields cause the Modified Data Tag (MDT) to be set.

Copy_To_Field allows only nonattribute data values to be written to a field. Use a Copy_To_PS call with the `modifier` parameter set to the `CTP_NO_CHECK` constant to copy data with values that fall into the range of attributes.

If the copy operation terminates normally at the end of the field, the driver returns `NO_ERR`.

Errors

`DATA_XFER_TRUNC_ERR`	This error indicates that the driver encountered the end of the application's source buffer before it finished copying a field.
`DATA_ERR`	This error indicates that the data copied from the source buffer contained an attribute value (0xC0 through 0xFF) or that there was an attempt to copy nonnumeric data into a numeric field.
`END_OF_PS_ERR`	This error indicates that the driver encountered the end of the PS before it finished overwriting a field.
`INP_INHIBITED_ERR`	This error indicates that the copy operation completed, but that the input-inhibited condition was present.
`PS_UNFMT_ERR`	This error indicates that the PS was unformatted.
`WRITE_ATTR_ERR`	This error indicates that the passed offset was positioned on an attribute.
`WRITE_PROT_FLD_ERR`	This error indicates that the application attempted to write data into a protected field in the PS.

Copy_To_PS

Purpose

This 3270 API call permits an application to copy data directly into a presentation space. ASCII characters supplied by the application are normally translated into DBC format; however, if you prefer, you can also write DBC codes directly by setting the `modifiers` parameter to the `CTP_NO_TRANS` constant.

Data from the source buffer specified by the application overlays some or all of the PS. The driver preserves attributes and protected fields and checks data integrity as part of the normal copy operation. You can override this feature, and write data to any portion of the PS, by setting the `CTP_NO_CHECK` constant as described under the definition of the `modifiers` parameter.

Changes to unprotected fields cause the Modified Data Tag (MDT) to be set, unless the `modifiers` parameter is set to the `CTP_NO_CHECK` constant. Writing into a protected field is not allowed unless the `modifiers` parameter is set to the `CTP_NO_CHECK` constant.

Important Use the CTP_NO_CHECK constant with care. For example, an application supporting file transfer can use this value to write any data value into any position in the PS, and the API won't protect the application if it does something wrong.

Format

```
Copy_To_Ps (&req_blk, asyncFlag);
```

Parameters

```
BYTE    *src_bufp;              /* passed */
WORD    src_offset;             /* passed */
WORD    num_bytes;              /* passed */
WORD    ps_offset;              /* passed */
BYTE    modifiers;              /* passed */

/* modifiers */
#define CTP_NO_CHECK            0x01
#define CTP_WRAP                0x02
#define CTP_NO_TRANS            0x04
```

Definitions

`asyncFlag`

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC Specify this constant to return control immediately to the caller.

SYNC Specify this constant to prevent control from returning to the caller until the request completes.

***src_bufp**

This POINTER points to a buffer whose contents are to be copied to the PS.

src_offset

This WORD specifies the offset into the source buffer from which the copy operation should begin.

num_bytes

This WORD specifies the number of bytes to copy from the source buffer to the PS. The number of bytes cannot exceed the size of the PS.

ps_offset

This WORD specifies the offset in the PS where the copy operation should begin. The number of bytes cannot exceed the size of the PS minus 1.

modifiers

This BYTE contains the modifiers for the Copy_To_PS call, as follows:

CTP_NO_CHECK Normally, the driver validates data that is written to the PS, as follows:

- Bytes in the source buffer that have attribute counterparts in an unprotected field in the PS must match; if they do not, the driver returns the error WRITE_ATTR_ERR.
- Bytes in the source buffer having nonattribute counterparts in the PS must have data values that fall in the non-attribute range, 0x00 through 0xBF; if they do not, the driver returns the error DATA_ERR is returned. (If numeric lock is in effect, attempting to write nonnumeric data into a numeric field also causes the driver to return a DATA_ERR.)

Setting this modifier suppresses data validation; it allows an application to write any data value into any position in the PS.

CTP_WRAP If this constant is specified, and the driver encounters the end of the PS as it copies bytes from the source buffer, the copy operation wraps to the beginning of the PS.

CTP_NO_TRANS	Normally, the driver translates characters in the source buffer to 3270 DBC format by using the translation table pointed to by the *asc_dbc_tabp parameter in the Connect_To_PS call. By setting the modifiers parameter to the CTP_NO_TRANS constant, you instruct the driver to not translate the codes. This allows your application to copy codes in DBC format directly into the presentation space.

Description

The Copy_To_PS request enables your application to copy data into a presentation space. Data from the application's source buffer can overlay all or a portion of the PS. The task that performs this operation preserves attributes and protected fields and checks data integrity unless you specify modifier options in the call. For more information about writing data to specific fields in a PS, see the Copy_To_Field call.

Changes to the PS are not transmitted to the CU until the application sends an AID key by way of a Send_Keys call.

This request does not affect the DAB, EAB, and cursor position.

If a Get_Update call is outstanding when this call is issued, changes to the PS caused by a Copy_To_PS will be returned to the application.

Errors

DATA_XFER_TRUNC_ERR	This error indicates that the driver encountered the end of the application's source buffer before it finished copying to the PS.
DATA_ERR	This error indicates that there was an attempt to write an attribute value (0xC0 through 0xFF) into a nonattribute position.
INP_INHIBITED_ERR	This error indicates that the copy operation was aborted because an input-inhibited condition was present.
WRITE_ATTR_ERR	This error indicates that an attempt was made to overwrite an attribute with a different value.
WRITE_PROT_FLD_ERR	This error indicates that the application attempted to write data into a protected field in the PS.

Deactivate_Prt_Sess

Purpose

This 3270 API call instructs the driver to immediately deallocate a printer session, dispose of any buffered data, and discard any held requests (such as a Get_Update request). The driver also makes the session unavailable to the host unless the `DPS_KEEP_SESSION` constant is specified for the `modifiers` parameter.

❖ *CUT note:* CUT drivers cannot support this call.

After the application issues this call, attempts by the host to communicate with the session will be rejected with a *device unavailable* error.

Format

```
Deactivate_Prt_Sess (&req_block, asyncFlag);
```

Parameters

```
BYTE    *modifiers;             /* passed */

/* modifiers */
#define DPS_KEEP_SESSION        0x01
```

Definitions

asyncFlag

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC Specify this constant to return control immediately to the caller.

SYNC Specify this constant to prevent control from returning until the request completes.

ps_id

For this call, this parameter in the request header specifies the printer session the application wants to deactivate.

modifiers

This BYTE allows you to select options that control the way data in the presentation space or the related application buffer are manipulated. These options are:

DPS_KEEP_SESSION
This option instructs the driver to not terminate the host session supporting the PS.

Errors

PS_INACTIVE_ERR	This error indicates that the specified printer session was never activated.

Disconnect_From_PS

Purpose

This 3270 API call instructs the driver to immediately break the logical connection to a PS and discard any held requests (such as a Get_Update request). The driver also makes the session unavailable to the host unless the `DC_KEEP_SESSION` constant is specified for the `modifiers` parameter.

Format

```
Disconnect_From_PS (&req_block, asyncFlag);
```

Parameters

```
BYTE    modifiers;              /* passed */

/* modifiers */
#define DC_KEEP_SESSION         0x01
```

Definitions

`asyncFlag`

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

`ASYNC` Specify this constant to return control immediately to the caller.

`SYNC` Specify this constant to prevent control from returning to the caller until the request completes.

`ps_id`

For this call, this parameter in the request header specifies the printer session that the application wants to deactivate.

`modifiers`

This `BYTE` contains the modifiers for the Disconnect_From_PS call, as follows:

`DC_KEEP_SESSION`
This modifier instructs the driver not to terminate the host session supporting the PS.

Example

The following example, taken from the sample application presented in Chapter 2 of this guide, shows a `case` statement that terminates the connection when the user quits the application.

```
case fileID:
```

```
                switch (theItem) {
                  case quitCommand:
                    if (!dft[session]->last_request) {
                      dft[session]->req_blk.discps.modifiers = NO_MODS;
/*
      ErrorMessage("Disconnecting session",session);
      ErrorMessage("Disconnecting ps_id is",dft[session]->req_blk.ps_id);
*/                      if (err = Disconnect_From_PS(&dft[session]->req_blk,ASYNC)) {
                          ErrorMessage("Glue Disc_PS Error",err);
                          DoneFlag = TRUE;
                          ClearConnect();
                          }
                    }
                    dft[session]->last_request = RC_DISCONNECT_FROM_PS;
                    break;
                  default:
                    break;
                }
                break;
```

Errors

PS_INACTIVE_ERR	This error indicates that the application never activated the specified printer session.

Do_Special_Func

Purpose

This 3270 API call allows an application to request that the driver execute a function unique to itself. You could, for example, set operational parameters, initiate a diagnostic, retrieve specific information pertaining to a driver, and so on.

Important Use this call sparingly, if at all. It defeats the purpose of an API if an application has to know many details about a driver.

Format

```
Do_Special_Func(&req_blk, asyncFlag);
```

Parameters

```
BYTE    func_code;          /* passed */
BYTE    *passed_infop;      /* passed */
BYTE    *ret_infop;         /* passed */
```

Definitions

asyncFlag

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC	Specify this constant to return control immediately to the caller.
SYNC	Specify this constant to prevent control from returning to the caller until the request completes.

port_id

For this call, this parameter in the request header specifies a particular port where the function should be executed. If you want the function to apply to all ports, or to the driver in general, pass 0xFF for this parameter.

ps_id

Set this parameter in the request header to one of the following values:

0xFF	For the function to apply to all connected PSs
0x00	For the function to apply to no connected PSs
A specific PS ID	For the function to apply to a particular PS

func_code

This BYTE specifies the function to be performed.

***passed_infop**

This POINTER points to a block of parameters that the routine needs to execute.

***ret_infop**

This POINTER points to a block of memory into which the routine will return function results.

Errors

SPEC_FUNC_FAILED_ERR	This error indicates that the special request failed.

Find_Field

Purpose

This 3270 API call searches for a field within a PS. The call can search the current, next, or previous field. In addition, you can limit the search to protected fields or unprotected fields.

The call returns the following items:

- an offset from the beginning of the PS to the beginning of the data portion of the found field; that is, the offset to the byte immediately following the field attribute
- the field attribute
- the length of the data portion of the field
- an indication if the field wraps

Format

```
Find_Field (&req_blk, asyncFlag);
```

Parameters

```
WORD    ps_offset;              /* passed */
BYTE    srch_type;              /* passed */

WORD    fnd_offset;             /* returned */
WORD    len;                    /* returned */
WORD    wrap_len;               /* returned */
BYTE    attr;                   /* returned */

/* srch_type*/
#define FF_CUR_FLD              1
#define FF_NXT_ANY              2
#define FF_NXT_UNPROT           3
#define FF_NXT_PROT             4
#define FF_PRV_ANY              5
#define FF_PRV_UNPROT           6
#define FF_PRV_PROT             7
```

Definitions

asyncFlag

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC Specify this constant to return control immediately to the caller.

SYNC Specify this constant to prevent control from returning to the caller until the request completes.

ps_offset

This WORD specifies the location in the PS where the search will begin for the type of field specified in the `srch_type` parameter.

srch_type

This BYTE specifies the type of field to be found, as follows:

`FF_CUR_FLD`	Finds the current field (located at the offset)
`FF_NXT_ANY`	Finds the next field regardless of type
`FF_NXT_UNPROT`	Finds the next unprotected field
`FF_NXT_PROT`	Finds the next protected field
`FF_PRV_ANY`	Finds the previous field regardless of type
`FF_PRV_UNPROT`	Finds the previous unprotected field
`FF_PRV_PROT`	Finds the previous protected field.

fnd_offset

This WORD returns the offset from the beginning of the PS to the first byte following the field attribute. If the designated field wasn't found, the driver sets the parameter to 0xFFFF.

len

This WORD returns the length of the field found, not including the length of the attribute byte. If the field wraps to the beginning of the PS, the length returned includes only the number of bytes from the beginning of the data portion of the field to the end of the PS. For example, if an attribute occupies the last position in the PS, `len` would be 0, and the `wrap_len` parameter would contain the length of the data portion of the field.

wrap_len

If this WORD is not 0, it indicates that the field wraps to the beginning of the PS, and the value of `wrap_len` is the number of bytes from the beginning of the PS to the end of the field.

attr

This BYTE returns the attribute associated with the field. See *IBM 3174/3274 Control Unit to Device Product Attachment Information* (October 16, 1986) for an explanation of the attribute-byte format.

Errors

NOT_FOUND_ERR	This error indicates that the driver did not find the specified type of field.
PS_UNFMT_ERR	This error indicates that the PS was not formatted.

Get_Cursor

Purpose This 3270 API call returns the position of the cursor in the PS.

Format

```
Get_Cursor (&req_blk, asyncFlag);
```

Parameters

```
WORD    offset;                 /* returned */
```

Definitions

asyncFlag

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC Specify this constant to return control immediately to the caller.

SYNC Specify this constant to prevent control from returning to the caller until the request completes.

offset

This WORD returns the position of the cursor as an offset from the beginning of the PS.

Errors None

Get_DSC_Prt_Data

Purpose

This request allows an application to retrieve DSC printer session data. The call is held by the driver until data is received from the host. The call completes when the driver detects that the "Start Print" bit in the WCC has been set. The setting of this bit indicates that the host has completed updating the print buffer and is a signal to initiate printing of the buffer contents.

The application can specify whether it wants to copy the PS, DAB, DABE, or EAB buffers. The number of bytes transferred to each buffer is equal to the current size of the driver's buffer as specified by the host application (that is, either the default or alternate size). The size of each of the buffers associated with `ps_bufp`, `dab_bufp`, `dabe_bufp`, and `eab_bufp` should equal (or exceed) the value of the `prtbuf_size` parameter in the Activate_Prt_Sess call.

The presentation-space data is returned in DBC format as specified in the table pointed to by the `*ebc_dbc_tabp` parameter of the Activate_Prt_Sess call.

Format

```
Get_DSC_Prt_Data (&req_blk, asyncFlag);
```

Parameters

```
BYTE    *ps_bufp;              /* passed */
BYTE    *dab_bufp;             /* passed */
BYTE    *dabe_bufp;            /* passed */
BYTE    *eab_bufp;             /* passed */
LONG    wait_time;             /* passed */

WORD    num_bytes_rcved;       /* returned*/
BYTE    buf_size_state;        /* returned*/
BYTE    wcc;                   /* returned*/
BYTE    end_job;               /* returned*/
```

Definitions

`asyncFlag`

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC Specify this constant to return control immediately to the caller.

SYNC Specify this constant to prevent control from returning to the caller until the request completes.

`*ps_bufp`

This POINTER points to a buffer designated to receive data from the PS. Set this pointer to NIL if you do not intend to copy the PS.

***dab_bufp**

This POINTER points to a buffer designated to receive data from the DAB. Set this pointer to NIL if you do not intend to copy the DAB.

***dabe_bufp**

This POINTER points to a buffer designated to receive data from the DABE. Set this pointer to NIL if you do not intend to copy the DABE.

***eab_bufp**

This POINTER points to space designated to receive data from the EAB. Set this pointer to NIL if you do not intend to copy the EAB.

wait_time

This LONG parameter specifies the maximum timeout period that the driver should wait for data. If the timeout period expires, the driver returns `TIMEOUT_ERR`.

The value passed represents a number of 100-millisecond ticks. There are two special values, as follows:

0xFFFFFFFF	Specify this value to instruct the driver to wait forever for data; that is, the driver will never return `TIMEOUT_ERR`.
0	Specify this value to instruct the driver to return `TIMEOUT_ERR` immediately if update data doesn't exist at the time of the call.

num_bytes_rcved

This WORD returns the number of bytes that the driver transferred into one of the application's receive buffers. The number of bytes transferred is the same for all specified receiving buffers.

buf_size_state

This BYTE indicates whether the buffer is currently in its default or alternate size, as follows:

`GDP_DEFAULT_SIZE`	This value indicates that the buffer is currently in its default size.
`GDP_ALT_SIZE`	This value indicates that the buffer is currently in its alternate size.

wcc

This BYTE returns a copy of the Write Control Character (WCC). Bits 2 and 3 (using IBM's numbering scheme) describe the printout format.

end_job

This BYTE, if returned as TRUE, indicates that the driver received notification at the protocol level that the current print job has completed. If the notification has not been received, the parameter is returned as FALSE.

Important This parameter is valid only if the underlying host communications protocol is SNA. If the protocol is not SNA, the parameter always returns FALSE.

Description

Other calls issued by the application to the session, with the exception of Deactivate_Prt_Sess, are rejected while this call is outstanding.

A copy of the WCC is returned to permit the application to examine the printout format bits. Because DSC data is validated by the driver before being conveyed to the application, a DSC print operation can only fail because of external problems such as a printer malfunction, or, for an application acting as a print spooler, for problems such as a disk file becoming full, a disk volume becoming unavailable, and so on. An application should deal with such non-recoverable conditions by sending a Deactivate_Prt_Sess to render the session unavailable to the host.

Once print data is successfully retrieved via this call, it is lost at the driver level; that is, another Get_DSC_Prt_Data does not complete until the host application once again updates the print buffer and sets the "Start Print" bit in the WCC.

To increase throughput, an application may wish to employ a double-buffering scheme. As soon as a Get_DSC_Prt_Data call completes, another one can be issued immediatelyusing a different request block and receive buffers.

If the underlying host protocol is SNA:

- An application can continue to issue Get_DSC_Prt_Data calls until a NO_HOST_SESS_ERR is returned. At that time, the application can issue a Deactivate_Prt_Sess call to either deallocate control of the printer session or retain control of the session and wait for another bind.

- LU type 3 protocol is the same as LU type 2 protocol in that they both deal with a presentation space and receive and process the data stream in an identical manner. The chief difference is that printer orders may be embedded in the presentation space. As with LU type 2, the current size of the PS depends on the Erase/Write and Erase/Write Alternate commands received from the host application and the Bind received. The current size of the PS is reflected in the `buf_size_state` parameter. See the *3274 Description and Programmer's Guide (GA23-0061)* for more details about the relationship between the Bind and Erase/Write and Erase/Write Alternate commands.

Errors

`DAB_UNSUPP_ERR`	This error indicates that the driver does not support the DAB.
`DABE_UNSUPP_ERR`	This error indicates that the driver does not support the extended DAB.
`EAB_UNSUPP_ERR`	This error indicates that that the driver does not support the EAB.
`NO_HOST_SESS_ERR`	This error indicates that the underlying host session no longer exists.
`PS_INACTIVE_ERR`	This error indicates that the specified printer session was never activated.
`TIMEOUT_ERR`	This error indicates that the interval specified in `wait_time` was exceeded.
`SESS_TYPE_ERR`	This error indicates that the application sent the print request to the wrong type of session; the request is valid only for DSC printer sessions.

Get_Host_Connection_Info

Purpose

This 3270 API call returns information about the connection method specified by the conn_id parameter in the request block.

Format

```
Get_Host_Connection_Info (&req_blk, asyncFlag);
```

Parameters

```
CONN_INFO   *conn_infop;          /* passed and returned*/

typedef struct conn_info
{
      struct
         {
              BYTE prod_id[4];
              BYTE version[4];
              BYTE misc[8];
         } vendor;
         BYTE conn_means;
         BYTE drvr_type;
         BYTE io_compl_supp;
         BYTE timeout_supp;
         BYTE eab_supp;
         BYTE dab_supp;
         BYTE dabe_supp;
         LONG dev_feats_supp;
         WORD reqs_supp[NUM_API_REQS];
         WORD port_map;
         struct
         {    struct
              {       BYTE supported;
              {       BYTE lu_type;
              {       BYTE scrn_emul;
                      BYTE connected;
                      WORD max_prtbuf_size;
              } ps[NUM_PS];
         } port_info[NUM_PORTS];

} CONN_INFO;
```

Definitions

conn_id

This field appears in the request header, and specifies the connection method about which information will be returned.

port_id

This field appears in the request header, but is ignored for this call.

ps_id

This field appears in the request header, but is ignored for this call.

asyncFlag

This flag is ignored for this call.

***conn_infop**

This POINTER points to a buffer in which the following information is returned:

vendor — This parameter is a structure containing the following three fields:

prod_id — This field supplies a 4-byte ASCII string containing one of the following constants:

ADFT	Apple DFT
ACUT	Apple CUT
SIMW	Simware
DCAC	DCA CUT
DCAD	DCA DFT
APPL	AppleLine
AVTC	Avatar CUT
AVTD	Avatar DFT
CXIC	CXI CUT
CXID	CXI DFT

version — This field supplies the version of the driver as a 4-byte displayable ASCII string.

misc — This field supplies 8 bytes of driver-specific information.

`conn_means` This BYTE indicates that the emulated underlying connectivity means is one of the following:

`CUT`

`DFT_SNA`

`DFT_LOCAL`

`CU_SNA`

`CU_LOCAL`

`OTHER`

`drvr_type` This BYTE indicates that the type of driver supporting the connection is one of the following:

`GI_TEMP_DRVR`	A temporary driver residing in the application's heap
`GI_PERM_DRVR`	A permanent driver residing in the system heap

`io_compl_supp`
This BYTE is set to TRUE if the driver can support a call to an I/O completion routine at the interrupt level; the byte is set to FALSE if not.

`timeout_supp` This BYTE is set to TRUE if the driver can support timeouts; the byte is set to FALSE if not. If the driver cannot support timeouts, then it ignores timeout values in calls that specify them, such as Get_Update and Get_Passthru_Data. The driver treats such calls as though they were issued with a timeout value of *wait forever* (0xFFFFFFFF).

`eab_supp` This BYTE is set to TRUE if the driver supports an EAB; the byte is set to FALSE if not. If the driver does not support this type of buffer, then it rejects calls that attempt to retrieve data from the EAB with an `EAB_UNSUPP_ERR` error. See "The IBM Attribute Buffers" in Chapter 1 for more information.

`dab_supp` This BYTE is set to TRUE if the driver supports a display attribute buffer, FALSE if not. If the driver does not support this type of buffer, then it rejects calls that attempt to retrieve data from the DAB with a `DAB_UNSUPP_ERR` error. See "Apple Attribute Buffers" in Chapter 1 for more information.

`dabe_supp` This BYTE is set to TRUE if the driver supports extended DAB bytes in the DAB, FALSE if not. If the driver does not support this type of buffer, then it rejects calls that attempt to retrieve data from the DABE with a `DABE_UNSUPP_ERR` error. See "Apple Attribute Buffers" in Chapter 1 for more information. This byte is set to `FALSE` if `dab_supp` is FALSE.

`dev_feats_supp`
This LONG parameter is a bitmap that indicates if a driver can support the following device features:

`GI_APL_TEXT`	APL/Text
`GI_DEAD_KEYS`	Dead keys
`GI_ATTR_SELECTION`	Attribute selection (PSHICO)
`GI_PSS`	Programmed symbol sets (PSS)
Bits 4–31	Undefined at the time of publication

`reqs_supp` This parameter is an array of 16-bit words indicating the API requests and modifiers that the driver can support. The high-order bit (15) of a word is set if the request is supported. If the request supports `keybds_supp` modifiers, bits 0 through 14 indicate the specific modifiers supported. The bit settings match the modifier values.

Your application can access array elements by the symbolic names defined for the request codes, as follows:

Word 0	Open_Host_Connection
Word 1	Close_Host_Connection
Word 2	Get_Host_Connection_Info
Word 3	Connect_To_PS
Word 4	Disconnect_From_PS
Word 5	Send_Keys
Word 6	Copy_To_PS
Word 7	Copy_From_Buffer
Word 8	Copy_To_Field
Word 9	Copy_From_Field
Word 10	Copy_OIA
Word 11	Search_String
Word 12	Find_Field
Word 13	Get_Update
Word 14	Get_Cursor
Word 15	Set_Cursor
Word 16	Set_Color_Support
Word 17	Send_Passthru_Data
Word 18	Get_Passthru_Data
Word 19	Post_Passthru_Reply
Word 20	Do_Special_Func
Word 21	Activate_Prt_Sess
Word 22	Deactivate_Prt_Sess
Word 23	Get_DSC_Prt_Data
Word 24	Get_LU1_Prt_Data
Word 25	Post_Prt_Reply
Word 26	Send_Prt_Control
Word 27	Check_Session_Bind

`port_map` This WORD is a bitmap indicating the specific ports the driver is managing. A value of 1 indicates that driver controls the port; a value of 0 means indicates that the driver does not control the port. For slots, bits 0 through 15 represent slots 0 through 15. For serial ports, bits 0 and 1 represent the modem port and printer port, respectively.

`port_info` This parameter is a 16-element array. An array element is valid only if its corresponding `port_map` bit is set to 1. Each array element contains information about the presentation spaces or the printer sessions that the port supports, as follows:

`supported` This BYTE is TRUE if the driver supports the PS or printer session. If the byte is set to FALSE, the `lu_type byte` in this array is set to `GI_NO_LU` and `connected byte` in this array is set to FALSE.

`connected` This BYTE indicates whether the PS is currently connected by a Connect_To_PS call, or whether the printer session has been activated by an Activate_Prt_Sess call. TRUE indicates the PS is connected or the session is activated; FALSE indicates the PS is not connected or the session is activated. If a PS is unsupported, `connected` is FALSE.

`lu_type` This BYTE specifies the configured session type. Possible values are as follows:

`GI_NO_LU`	Not an SNA connection method
`GI_LU_1`	LU 1
`GI_LU_2`	LU 2
`GI_LU_3`	LU 3
`GI_LU_1_OR_3`	A generic printer that supports both LU 1 and LU 3

`scrn_emul` This BYTE specifies the type of 3278 model that the application is emulating in terms of screen size. This parameter applies only if the `lu_type` byte in this array is `GI_LU_2`. Possible constant values and the screen sizes they represent are as follows:

`GI_MOD_2`	Model 2, screen size is 1920
`GI_MOD_3`	Model 3, screen size is 2560
`GI_MOD_4`	Model 3, screen size is 3440
`GI_MOD_5`	Model 3, screen size is 3564

`max_prt_buf_size`
This WORD specifies the buffer capacity of the driver for handling print data from the host. The field applies only if if `lu_type` in this array is set to handle type 1, or type 3, or both.

Errors

`DAB_UNSUPP_ERR`	This error indicates that the driver does not support the DAB.
`DABE_UNSUPP_ERR`	This error indicates that the driver does not support the DABE.
`EAB_UNSUPP_ERR`	This error indicates that the driver does not support the EAB.

Get_LU1_Prt_Data

Purpose

This 3270 API call allows an application to gain access to SCS or IPDS printer data.The driver holds the call until printer data is received from the host. While this call is outstanding, the driver rejects other calls issued by the application to the session, with the exception of Deactivate_Prt_Sess.

Format

```
Get_LU1_Prt_Data (&req_blk, asyncFlag);
```

Parameters

```
BYTE    *rcv_bufp;              /* passed */
WORD    max_bytes_to_rcv;       /* passed */
LONG    wait_time;              /* passed */

WORD    num_bytes_rcved;        /* returned */
BYTE    data_type;              /* returned */
BYTE    data_end;               /* returned */
BYTE    end_job;                /* returned*/
```

Definitions

asyncFlag

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC Specify this constant to return control immediately to the caller.

SYNC Specify this constant to prevent control from returning to the caller until the request completes.

***rcv_bufp**

This POINTER points to the application buffer where the driver will return data.

max_bytes_to_rcv

This WORD specifies the maximum number of bytes that the driver will transfer to the buffer.

wait_time

This LONG parameter specifies the maximum timeout period that the driver should wait for data. If the timeout period expires,the driver returns the error TIMEOUT_ERR.

The value passed represents a number of 100-millisecond ticks. There are two special values, as follows:

0xFFFFFFFF	Specify this value to instruct the driver to wait forever; that is, the driver will never return `TIMEOUT_ERR`.
0	Specify this value to instruct the driver to return `TIMEOUT_ERR` immediately if no data is present at the time of the call.

num_bytes_rcved

This WORD returns the number of bytes that the driver transferred. A count of 0 is valid.

data_type

This BYTE returns the type of data transferred, as follows:

`GLP_FMH`	This value indicates that the data was FMH1.
`GLP_SF`	This value indicates that the data was structured field data.
`GLP_SCS_DATA`	This value indicates that the data was normal SCS data.

This value will be the same for all portions of print data transferred until the end of the data unit (`data_end` set to either `GLP_END` or `GLP_END_REPLY`).

data_end

If this BYTE returns `GLP_NOT_END`, it indicates the driver has not completed sending the data unit to the application. The application should continue issuing Get_LU1_Prt_Data calls until this flag becomes `GLP_END_REPLY`, at which time the application must issue Post_Prt_Reply before issuing Get_LU1_Prt_Data calls again.

If this parameter is set to `GLP_END`, it indicates that the sending of the data unit has been aborted and that no reply is necessary.

end_job

This BYTE, if returned as TRUE, indicates that the driver received notification at the protocol level that the current print job has completed. If the notification has not been received, the parameter is returned as FALSE.

Description

In addition to SCS data, the driver conveys structured fields and function-management headers (type 1) with attached data unchanged to the application. Your application is responsible for validating function management headers, structured field types, and data. The driver presents only one function-management header with attached data, or one structured field, or one chain of regular SCS data in each series of calls that end with the `data_end` parameter set to the `GLP_END_REPLY` constant.

A driver might have more data buffered or might be waiting for further data from the host. In such cases, the driver sets the `data_end` parameter to `GLP_NOT_END` to inform the application that more data is coming. An application should issue Get_LU1_Prt_Data calls until the `data_end` parameter becomes `GLP_END_REPLY`. The application must then issue a Post_Prt_Reply call to inform the driver of the validity of the previous data received.

After issuing Post_Prt_Reply the application can again begin to issue Get_LU1_Prt_Data calls.

❖ *Note:* Instead of posting a reply, your application can also issue, at any time, a Deactivate_Prt_Sess call to deallocate the printer session and terminate contact with the host.

A larger buffer decreases the number of Get_LU1_Prt_Data calls that the application needs to repeatedly issue to retrieve data. For maximum performance, the application's receive buffer should be as large as the buffer maintained by the driver. Refer to the `max_prtbuf_size` parameter in the description of the Get_Host_Connection_Info call in this chapter to determine the driver's buffer capacity.

To increase throughput, you may want to employ a double-buffering scheme. As soon as a Get_LU1_Prt_Data call completes, issue another one immediately using a different request block and receive buffers.

If the underlying host protocol is SNA, the following are true:

- An application can continue to issue Get_LU1_Prt_Data calls until a `NO_HOST_SESS_ERR` is returned. At that time, the application can issue a Deactivate_Prt_Sess call to either deallocate control of the printer session or retain control of the session and wait for another bind.
- The `data_end` parameter is set to `GLP_END` when the host abnormally termates a chain (for example, Cancel) or if the application issues a Send_Prt_Control call with a Cancel request. The data unit transferred to the application in this case should be considered suspect and incomplete.

- A print job sent to the host is not always terminated by an `end_job` notification or an Unbind. Usually, print jobs are spooled and handled by a host system utility which delimits print jobs within brackets. Some custom host print applications, however, begin a bracket and send multiple print jobs without ever ending the bracket. Therefore, your application should terminate a print job based on both the reception of either a `NO_HOST_SESS_ERR` or an `end_job` notification, and also inactivity for a sufficiently long timeout interval (the timeout can be set in the `wait_time` parameter).

Errors

`LOST_DATA_ERR`	This error indicates that the driver received data and couldn't save it because of insufficient buffer space. No data is transferred when this error occurs. The driver returns the error only once to an application. Subsequent Get_LU1_Prt_Data calls should receive data successfully unless another lost data condition arises.
`NO_HOST_SESS_ERR`	This error indicates that the underlying host session no longer exists. The error also signals that a print job has been completed or aborted.
`PS_INACTIVE_ERR`	This error indicates that the underlying host session no longer exists. This signals either the completion or abortion of a print job in progress.
`TIMEOUT_ERR`	This error indicates that the interval specified in `wait_time` was exceeded.
`SESS_TYPE_ERR`	This error indicates that the print request was sent to the wrong type of session; the request is valid only for LU1 printer sessions.
`STATE_ERR`	This error indicates that the request is inappropriate; the application must reply to the driver with a Post_Prt_Reply call before it can issue another Get_LU1_Prt_Data call.

Get_Passthru_Data

Purpose

This 3270 API call is issued by the application in order to receive data from the host that has not been mapped to the PS task. If a Get_Passthru_Data request is not outstanding when the driver receives data, the driver will buffer this data until the application issues the request.

Important Only a DFT or CU driver can support Get_Passthru_Data.

Format

```
Get_Passthru_Data (&req_block, asyncFlag);
```

Parameters

```
BYTE    *rcv_bufp;              /* passed */
WORD    max_bytes_to_rcv;       /* passed */
WORD    wait_time;              /* passed */

WORD    num_bytes_rcved;        /* returned */
BYTE    data_end;               /* returned */
```

Definitions

asyncFlag

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC Specify this constant to return control immediately to the caller.

SYNC Specify this constant to prevent control from returning to the caller until the request completes.

***rcv_bufp**

This POINTER points to the application's destination buffer that will receive data from the host.

max_bytes_to_rcv

This WORD specifies the number of bytes that the driver can write to the destination buffer.

wait_time

This WORD specifies the maximum amount of time the driver should wait for data. The value passed represents the number of 100-milliseconds ticks. If the timeout period expires, the driver will return TIMEOUT_ERR. A value of 0xFFFF instructs the driver to wait forever for data. A value of 0 instructs the driver to return TIMEOUT_ERR immediately if it finds no data at the time of the call.

num_bytes_rcved

This WORD returns the number of bytes of data that the driver transferred to the destination buffer.

data_end

If this BYTE is set to GPD_NOT_END, it indicates that the driver has not sent the entire structured field to the application. The application should continue issuing Get_Passthru_Data calls until the value becomes GPD_END or GPD_END_REPLY.

Description

You use Get_Passthru_Data primarily to receive structured field data that is not intended for processing by a PS component but by a higher-level function. The structured field types to be monitored and passed to the application by way of this call are specified in the Connect_To_PS call in the type_pass_datap parameter.

If no passthrough data is available at the time the call is received, the driver holds the request and waits for more updates. Other API requests to the PS can be sent by the application while a Get_Passthru_Data request is outstanding.

If a Get_Passthru_Data request is not outstanding at the time data is received, the driver buffers the data until such time as the request is issued. An application should issue Get_Passthru_Data calls in a timely fashion; if ti doesn't, processing of host data could be delayed for the session.

If the specified application buffer is not large enough to receive an entire structured field or if the structured field has not been completely received from the host, the driver sets the data_end flag to GPD_NOT_END in the request. The application should continue issuing Get_Passthru_Data calls until the data_end flag is set to either GPD_END or GPD_END_REPLY.

Only one structured field is conveyed in a series of Get_Passthru_Data calls that terminates with data_end set to GPD_END or GPD_END_REPLY. The structured field header containing the length field and ID will appear in the first buffer in the series.

You can use Get_Passthru_Data and Send_Passthru_Data to send and receive D0 structured fields with the INDSFILE 3270 PC file-transfer method.

Errors

TIMEOUT_ERR	This error indicates that the value specified in wait_time was exceeded.
NO_HOST_SESS_ERR	This error indicates that the underlying host session no longer exists.
LOST_DATA_ERR	This error indicates that the driver received data and could not save it because of insufficient buffer space; no data was transferred. The driver returns the error only once to an application. Subsequent Get_Passthru_Data calls should receive data successfully, unless another lost data condition arises.
NO_PASS_DATA_TYPES_ERR	This error indicates that the application did not specify any structured fields to be received in the Connect_To_PS call.
STATE_ERR	This error indicates that the call was inappropriate; that is, if a previous Get_Passthru_Data call completed with a GPD_END_REPLY notification, the application must issue a Post_Passthru_Reply call before it issues another Get_Passthru_Data call.

Get_Update

Purpose

This 3270 API call returns information to the application describing changes to the PS, DAB, DABE, EAB, OIA, cursor position, or alarm state. Get_Update is designed primarily to support an application acting as a terminal emulator.

An **update record** identifies a row in the PS, DAB, DABE, EAB that was changed. The driver writes update records contiguously and in ascending sequence by row number. Only one update record is returned for each changed row. Gaps between row numbers often occur, however, since the host commonly updates a presentation space in bits and pieces.

Updates to the PS are normally presented to the application in ASCII format; however, you have the option of receiving them in DBC format by using the GU_NO_TRANS constant in the `modifiers` parameter.

Format

```
Get_Update (&req_block, asyncFlag);
```

Parameters

```
WORD          wait_time;          /* passed */
UPD80_REC     *ps_recp;           /* passed */
UPD80_REC     *dab_recp;          /* passed */
UPD80_REC     *dabe_recp;         /* passed */
UPD80_REC     *eab_recp;          /* passed */
BYTE          modifiers;          /* passed */

BYTE          cursor_row;         /* returned */
BYTE          cursor_col;         /* returned */
BYTE          alarm;              /* returned */
BYTE          scrn_width;         /* returned */
BYTE          num_ps_recs;        /* returned */
BYTE          num_dab_recs;       /* returned */
BYTE          num_dabe_recs;      /* returned */
BYTE          num_eab_recs;       /* returned */

/* modifiers */
#define GU_IGNORE_PS            0x0001
#define GU_IGNORE_CURSOR        0x0002
#define GU_IGNORE_OIA           0x0004
#define GU_NO_TRANS             0x0008
#define GU_CHECK_ALL_DABE       0x0010
```

```
typedef struct upd80_rec
{       BYTE row;
        BYTE col;
        WORD len;
        BYTE data[80];
} UPD80_REC;

typedef struct upd132_rec
{       BYTE row;
        BYTE col;
        WORD len;
        BYTE data[132];
} UPD132_REC;

/* alarm */
#define GU_ALARM_ON             1
#define GU_ALARM_OFF            0
```

Definitions

asyncFlag

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC Specify this constant to return control immediately to the caller.

SYNC Specify this constant to prevent control from returning until the request completes.

wait_time

This LONG parameter specifies the maximum timeout period the driver should wait for an update. If the timeout period expires, the driver returns the error TIMEOUT_ERR.

The value passed represents a number of 100-millisecond ticks. There are two special values, as follows:

0xFFFFFFFF Specify this value to instruct the driver to wait forever; that is, the driver will never return TIMEOUT_ERR.

0 Specify this value to instruct the driver to return TIMEOUT_ERR immediately if no update data is present at the time of the call.

***ps_recp**

This POINTER points to a buffer in which the driver returns an array of PS update records. Set this pointer to NIL if you don't want those update records. When the call completes, only `ps_recp[0]` through `ps_recp[num_ps_recs minus 1]` are valid in the buffer.

***dab_recp**

This POINTER points to a buffer in which the driver returns an array of DAB update records. Set this pointer to NIL if you don't want those update records. When the call completes, only `dab_recp[0]` through `dab_recp[num_dab_recs minus 1]` are valid in the buffer.

***dabe_recp**

This POINTER points to a buffer in which the driver returns an array of DABE update records. Normally, update records are sent only if symbol set information (bits 3-1 within a DABE byte) changes. However, you can also use the `GU_CHECK_ALL_DABE` constant in the `modifiers` parameter to cause the driver to send update records for any portion of a DABE byte that changes.

Set this pointer to NIL if you don't want those update records. When the call completes, only `dabe_recp[0]` through `dabe_recp[num_dabe_recs minus 1]` are valid in the buffer.

***eab_recp**

This POINTER points to a buffer in which the driver returns an array of EAB update records. Set this pointer to NIL if you don't want those update records. When the call completes, only `eab_recp[0]` through `eab_recp[num_eab_recs minus 1]` are valid in the buffer.

upd80_rec or upd132_rec update record

These structures each contain three fields and an array:

`row` This BYTE identifies the row in either the PS, the DAB, the EAB, or the OIA where an update occurred. For changes to the PS, DAB, and EAB, the row number ranges from 0 to the number of rows in the PS minus 1. A row-number value of 0xFF identifies updates to the OIA, which has only one row. The OIA update record is placed last in a set of update records. In addition, the driver sends no DAB update record for the OIA.

`col` This BYTE identifies the starting column number in the row containing the update. Column number 0 is the first column in a row.

`len` This WORD contains the number of updated bytes in the row, starting from `col`.

`data` This array contains the updated bytes. The updated bytes start at data[0], not data[`col`].

`modifiers`

This BYTE contains the modifiers for the Get_Update call, as follows:

`GU_IGNORE_PS` This value instructs the driver not to check for updates to the PS. As a result, the driver will not write PS, DAB, or EAB update records to the destination buffer. However, setting this value does not suppress the writing of OIA update records. To accomplish that, use `GU_IGNORE_OIA`.

`GU_IGNORE_CURSOR`
This value instructs the driver not to check for a change in cursor position in the PS.

The cursor position is still presented when the Get_Update request completes, even though a change in cursor position does not cause the completion.

`GU_IGNORE_OIA`
This value instructs the driver not to check for changes to the OIA. Thus, the driver doesn't write an OIA update record (`row` = 0xFFFF) in any of the PS or DAB update buffers.

`GU_NO_TRANS` Normally, the driver translates 3270 device buffer codes into ASCII characters using the table pointed to by the `*dbc_asc_tabp` parameter of the Connect_To_PS call. However, if you specify `GU_NO_TRANS`, the driver does not perform the translation. This option allows your application to receive device buffer codes in a PS update buffer.

`GU_CHECK_ALL_DABE`
This value instructs the driver to send DABE updates if any part of a DABE byte changes. Normally, DABE updates are sent only if bits 3-1 (symbol set information) change.

`cursor_row`

This BYTE returns the current row position of the cursor. The first row begins at 0.

cursor_col

This BYTE returns the current column position of the cursor. The first column begins at 0.

alarm

This BYTE returns the state of the alarm, as follows:

GU_ALARM_ON Alarm on

GU_ALARM_OFF Alarm off

scrn_width

This BYTE returns the current screen width as 80 or 132 columns.

num_ps_recs

This BYTE returns the number of PS update records that the driver wrote.

num_dab_recs

This BYTE returns the number of DAB update records that the driver wrote.

num_dabe_recs

This BYTE returns the number of DABE update records that the driver wrote.

num_eab_recs

This BYTE returns the number of EAB update records that the driver wrote.

Example

The following example, taken from the sample application presented in Chapter 2 of this guide, shows the code that sets up the Get_Update request block and retrieves the update.

```
Boolean setGet(session)
BYTE session;
{
   /* ErrorMessage("Setget session",session); */
   /* ErrorMessage("Setget ps_id",dft[session]->ps_id); */

   dft[session]->Gps_blk.net_addr.aNode = 0;
   dft[session]->Gps_blk.api_vars = api_vars;
   dft[session]->Gps_blk.port_id = Slot;
```

```
    dft[session]->Gps_blk.conn_id = saved_conn_id;
    dft[session]->Gps_blk.ps_id = dft[session]->ps_id;
    dft[session]->Gps_blk.getupd.wait_time = 0xFFFF;
    dft[session]->Gps_blk.getupd.ps_recp = &(dft[session]->ps[0]);
    dft[session]->Gps_blk.getupd.dab_recp = &(dft[session]->dab[0]);
    dft[session]->Gps_blk.getupd.dabe_recp = 0;
    dft[session]->Gps_blk.getupd.eab_recp = 0;
    dft[session]->Gps_blk.getupd.modifiers = NO_MODS;
    if (err = Get_Update(&dft[session]->Gps_blk,ASYNC)) {
        ErrorMessage("Glue Get_Update Error",err);
        return FALSE;
        }
    return TRUE;
}
```

Description

The format of the basic DAB and extended DAB bytes are described in the section "The Apple Attribute Buffers" in Chapter 1.

The type for the `*ps_recp`, `*dab_recp`, `*dabe_recp`, and `*eab_recp` buffer pointers is defined as `UPD80_REC` simply because an 80-column screen is most commonly emulated. If a 132-column display is being emulated (that is, a Model 5 screen has been specified in the `scrn_emul` parameter in the Connect_To_PS call), you should cast the pointers in `UPD132_REC` format. For Model 5 emulations, the driver always formats 132-byte update records even if the current screen size is 24 by 80.

If the driver has not updated PS, DAB, EAB, OIA, cursor position, or alarm state at the time the call is received, the driver holds the request awaiting updates. Other API requests can sent by the application while a Get_Update request is outstanding. (Thus, an application must maintain two request blocks, one for Get_Update calls and another for other API calls.)

The call completes when a timeout occurs or the driver detects a change in one of the following items:

PS (if row 0xFF, then it was a change in the OIA)
DAB
EAB
cursor position
alarm state

The driver returns update information in the buffers specified in the call. The driver decides when update information should be presented to the application by simply setting `result`, as with other requests. The driver may decide either to send update information immediately upon detecting any type of change or to wait until the buffers under surveillance have been been scanned in their entirety for accumulated changes.

Within an application's update buffers, the driver will return a series of update records in ascending sequence by row number. Typically, there will be gaps in the row number of the update records. The last record is an OIA record if a change to the OIA occurred.

Your application must allocate update buffers large enough to accommodate worst-case situations in which the entire screen is updated. For example, to support a 24 x 80 screen, you must allocate a buffer of (25 * size of `(UPD80_REC)`) bytes. (The 25th record is for the OIA.)

By setting `GU_IGNORE_PS` and `GU_IGNORE_CURSOR`, your application can monitor only the OIA for changes, such as an X-Clock or X-System drop. You should set `*ps_recp` to point to a buffer in which the driver returns the OIA record (`row` = 0xFF). Set `*eab_recp` if you also want the EAB image of the OIA.

A DAB update record is never sent for the OIA. Display modes other than normal display (intensified, non-display, highlighted, blinking, and underline modes) are not relevant to the OIA. You can obtain color information for updated OIA bytes from the EAB update record.

The codes sent in the PS update record for the OIA are 3270 device-buffer codes as described in *IBM 3174/3274 Control Unit to Device Product Attachment Information* (October 16, 1986).

Errors

`DAB_UNSUPP_ERR`	This error indicates that the driver does not support the DAB.
`DABE_UNSUPP_ERR`	This error indicates that the driver does not support the extended DAB.
`EAB_UNSUPP_ERR`	This error indicates that the driver does not support the EAB.
`TIMEOUT_ERR`	This error indicates that the interval specified in `wait_time` was exceeded.

Init_3270_API

Purpose This 3270 API call initializes the 3270 API and returns a handle that subsequent API calls must pass as the value of the `api_vars` field in the API request block.

Format

```
Init_3270_API();
```

Parameters none

Example This example from DFTerm.c sets the `api_vars` field of the request block to be equal to the handle that Init_3270_API returns.

```
api_vars = Init_3270_API();
```

Errors none

Open_Host_Connection

Purpose

This 3270 API call opens the driver for the specified connection type, such as for the Apple 3270 CUT or the Apple 3270 DFT connection type, and specifies the configuration information for the driver.

Important Your application must make an Open_Host_Connection call before it makes any other API calls that access that particular host connection.

When this call invokes its corresponding interface routine is invoked by this call, the interface routine attempts to open the driver associated with the specified host connection method. Once initialized, the driver establishes communication with a 3270 host.

Format

```
Open_Host_Connection (&req_blk, asyncFlag);
```

Parameters

```
LONG    conn_type;              /* passed */
BYTE    open_type;              /* passed */
BYTE    *config_infop;          /* passed */
WORD    config_info_len;        /* passed */

BYTE    ret_conn_id;            /* returned */

/* conn_type values */
#define NUM_CONN_TYPES      10

#define OC_APPLE_CUT        0
#define OC_APPLE_DFT        1
#define OC_APPLELINE        2
#define OC_SIMWARE          3
#define OC_AVATAR_CUT       4
#define OC_AVATAR_DFT       5
#define OC_DCA_CUT          6
#define OC_DCA_DFT          7
#define OC_CXI_CUT          8
#define OC_CXI_DFT          9

/* open_type values*/
#define OC_COLD             0
#define OC_WARM             1
#define OC_ATTACH           2
```

Definitions

asyncFlag

This flag is ignored for this call.

conn_type

This LONG parameter indicates the type of 3270 connection desired, as follows:

OC_APPLE_CUT	Apple CUT
OC_APPLE_DFT	Apple DFT
OC_APPLELINE	Apple AppleLine
OC_SIMWARE	Simware
OC_AVATAR_CUT	Avatar CUT
OC_AVATAR_DFT	Avatar DFT
OC_DCA_CUT	DCA CUT
OC_DCA_DFT	DCA DFT
OC_CXI_CUT	CXI CUT
OC_CXI_DFT	CXI DFT

open_type

This BYTE specifies the method for establishing a connection with the driver specified in conn_type, as follows:

OC_COLD	This value loads or reloads a driver.
OC_WARM	This value establishes a connection to a driver that is already loaded. Because OC_WARM does not reset the driver, your application can resume interacting with a presentation space to which it had previously connected. If the driver isn't currently loaded, OC_WARM acts like OC_COLD.
OC_ATTACH	This value causes the same behavior as OC_WARM except that OC_ATTACH does not attempt to load the driver if it isn't already present. You can use this value to connect to an executing driver that has already been passed configuration information.

***config_infop**

This POINTER points to configuration information for the 3270 driver selected in conn_type. The driver ignores the *config_infop if the open_type is OC_ATTACH. All configuration information must be present in the block pointed to by *config_infop, that is, no pointers to other data may be included in the configuration block. See "Configuration Information" in Chapter 1 for more information about the configuration of the Apple DFT and CUT drivers.

config_info_len

This WORD specifies the length of the configuration information block.

ret_conn_id

This BYTE returns the ID for the opened driver. All subsequent API calls to this connection must pass this value in the conn_id request block parameter.

Description

To determine if a driver of a specified connection type exists in the system, an application can issue an Open_Host_Connection call with the conn_type equal to the value of OC_ATTACH without issuingcalls to the connection.

Example

This example from DFTerm.c sets up the request and issues the Open_Host_Connection call.

```
/*  Issue an Open_Host_Connection, which returns immediately */
  dft[0]->req_blk.api_vars = api_vars;
  dft[0]->req_blk.net_addr.aNode = 0;
  dft[0]->req_blk.port_id = Slot;
  dft[0]->req_blk.io_compl = nil;
  dft[0]->req_blk.openhc.conn_type = OC_APPLE_DFT;
  dft[0]->req_blk.openhc.open_type = OC_COLD;
  dft[0]->req_blk.openhc.config_infop = &DFT_CFG;
  dft[0]->req_blk.openhc.config_info_len = sizeof(DFT_CFG);
  DFT_CFG.slot_map = 1;
  DFT_CFG.slot_map <<= Slot;

  for (sess_num = 0; sess_num < 5; sess_num++)
     DFT_CFG.slot_info[relSlot].lu_type[sess_num] = 0;
  for (sess_num = 0; sess_num < num_sessions; sess_num--)
     DFT_CFG.slot_info[relSlot].lu_type[sess_num] = ADFT_LU_TYPE_2;

  if (err = Open_Host_Connection(&(dft[0]->req_blk))) {
     ErrorMessage("Open_Host_Connection Error",err);
     Term_3270_API(api_vars);
     return 0;
     }
  if ((err = DFT_CFG.slot_status[relSlot]) != NO_ERR) {
     ErrorMessage("Slot Status Non-Zero",err);
```

```
    Term_3270_API(api_vars);
    return 0;
    }
if ((err = DFT_CFG.slot_info[relSlot].ps_status[0]) != ADFT_PS_SUPP) {
    ErrorMessage("Apple DFT Not Supported",err);
    Term_3270_API(api_vars);
    return 0;
    }
saved_conn_id = dft[0]->req_blk.openhc.ret_conn_id;
```

Errors

GLU_DRVR_OPEN_ERR	This error indicates that the interface could not establish communication with the driver. Usually, this means that the driver does not exist.
CONFIG_ERR	This error indicates that the configuration information is invalid.
Driver-specific errors	Other errors can be defined and returned by particular drivers.

Post_Passthru_Reply

Purpose This 3270 API call informs the driver of the validity of the received data. The driver then sends an appropriate response to the host. Your application should issue Post_Passthru_Reply when the `data_end` parameter is set to `GPD_END_REPLY` in a Get_Passthru_Data call.

Important Only DFT or CU driver can support Post_Passthru_Reply.

Format

```
Post_Passthru_Reply (&req_blk, asyncFlag);
```

Parameters

```
LONG    sense_code;          /* passed */
BYTE    *pss_infop;          /* passed */
```

Definitions

`asyncFlag`

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

`ASYNC` Specify this constant to return control immediately to the caller.

`SYNC` Specify this constant to prevent control from returning until the request completes.

`sense_code`

This LONG parameter, if 0, indicates that no error should be returned to the host. A nonzero value indicates that the data was in error, and the value is also the specific sense code to be returned to the host.

The high-order 2 bytes of the sense code contains the major sense information and the low-order 2 bytes contains minor sense information. In most cases, minor sense information is set to 0x0000.

The sense codes commonly returned for errors in structured fields are as follows:

0x10010000 RU data error

0x10030000 Function not supported

0x10050000 Parameter error

Non-SNA drivers will send Op-Check if `sense_code` is non-zero.

You can find dditional sense codes in the *3270 Data Stream Programmer's Reference* (GA23-0059).

***pss_infop**

This POINTER points to a 6-byte array containing Local Character Set IDs (LCID) to inform the driver of updates to LCID-to-PSS assignments. Set this parameter can be set to NIL if the application doesn't support programmed symbol sets or if no changes are required to the current LCID-to-PSS assignments. The initial state of the driver is that no LCID-to-PSS assignments are in effect.

The application updates this array when it processes a Load PS structured field. Using this array, the driver maps an LCID—as received in Start Field Extended (SFE), Set Attribute (SA), or Modify Field (MF) order—to a PSS ID value which is then set in the DABE. The first element in the array specifies the LCID associated with PSS 2, the second element specifies the LCID associated with PSS 3, and so on. An LCID value of 0xFF indicates the particular PSS is not assigned.

Errors

STATE_ERR	This error indicates that the application issued the call inappropriately. Either the previous Get_Passthru_Data call specified GPD_END, instead of GPD_END_REPLY, or a Get_Passthru_Data call had not been issued.

Post_Prt_Reply

Purpose

This 3270 API call informs the driver of the validity of received LU type 1 data. A Post_Prt_Reply call should follow all Get_LU1_Prt_Data calls that complete with `data_end` set to `GLP_END_REPLY`.

Your application may post a status of no error or an error status in the form of SNA sense codes. The driver conveys these sense codes to the host application within negative SNA responses.

As an alternative to issuing this call, an application can deal with a nonrecoverable error by issuing a Deactivate_Prt_Sess call.

Format

```
Post_Prt_Reply (&req_blk asyncFlag);
```

Parameters

```
LONG    sense_code;             /* passed */
```

Definitions

`asyncFlag`

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC Specify this constant to return control immediately to the caller.

SYNC Specify this constant to prevent control from returning until the request completes.

`sense_code`

If this LONG parameter is zero, indicates the received data was valid. If `sense_code` is nonzero, the data was in error, and the value is also the specific sense code to be returned to the host.

The high-order 2 bytes of the sense code contains the major sense information, and the low-order 2 bytes contains minor sense information. In most cases, minor sense information is set to 0x0000.

The sense codes commonly returned for errors in printer data are as follows:

0x10010000 RU data error

0x10030000 Function not supported

0x10050000 Parameter error

0x10080000 Invalid function-management header

Non-SNA drivers will send Op-Check and a Sense byte if `sense_code` is nonzero.

You can find additional sense codes in the *3270 Data Stream Programmer's Reference* (GA23-0059).

Errors

`NO_HOST_SESS_ERR`	This error indicates that the underlying host session no longer exists.
`PS_INACTIVE_ERR`	This error indicates that the application never activated the specified printer session.
`SESS_TYPE_ERR`	This error indicates that the application sent the print request to the wrong type of session; the request is valid only for printer sessions.
`STATE_ERR`	This error indicates that the request is inappropriate; the application must issue additional Get_LU1_Prt_Data calls until `data_end` becomes `GLP_END_REPLY`.

Search_String

Purpose

This 3270 API call searches for a string within a field or searches all or part of the PS. The call can perform forward or backward searches from an offset in the PS. A search does not wrap; that is, forward searches terminate at the end of the PS, and backward searches terminate at the beginning. ASCII characters supplied by the application are normally translated into DBC format; however, if you prefer, you can also compare DBC codes directly by using the `SS_NO_TRANS` constant in the `modifiers` parameter.

Format

```
Search_String (&req_blk asyncFlag);
```

Parameters

```
BYTE    *strp;                  /* passed */
WORD    len_or_eos;             /* passed */
WORD    ps_offset;              /* passed */
BYTE    modifiers;              /* passed */

WORD    fnd_offset;             /* returned */

/* modifiers */
#define SS_SRCH_FLD             0x01
#define SS_SRCH_BACK            0x02
#define SS_NO_TRANS             0x04
```

Definitions

`asyncFlag`

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC — Specify this constant to return control immediately to the caller.

SYNC — Specify this constant to prevent control from returning to the caller until the request completes.

`*strp`

This POINTER points to a string in the application's source buffer that will be matched in the PS.

len_or_eos

This WORD specifies, in bytes, the length of the string to be matched. However, if the value in the high-order byte is 0xFF, the contents of the low-order byte is an end-of-string marker. The driver will match the string up to, but not including, the marker in the PS. If a string is too long to be found in the PS with the specified ps_offset value, the driver returns the error PARM_ERR.

ps_offset

This WORD specifies an address in the PS where the search should begin. Use the Find_Field call to determine the starting location of a field.

modifiers

This BYTE allows you to specify the type of search to be performed.

SS_SRCH_FLD Specify this constant to restrict the search to the current field. The search begins at the location specified in ps_offset, and continues to the end of the field, to the end of the PS, or to the length specified in the len_or_eos parameter. If SS_SRCH_BACK is in effect, the search will proceed in the opposite direction. If the PS is unformatted—that is, no field attribute exists in the entire space—the driver returns the error PS_UNFMT_ERR.

SS_SRCH_BACK Specify this constant to cause the search to proceed from the location specified in ps_offset either to the beginning of the field or to the beginning of the PS. The search begins at ps_offset and ends at the beginning of the PS.

SS_NO_TRANS Normally, the driver translates characters in the source buffer to 3270 DBC format by using the translation table pointed to by the *asc_dbc_tabp parameter in the Connect_To_PS call. By setting the modifiers parameter to CTF_NO_TRANS, you instruct the driver to not translate the codes. This allows your application to compare codes in DBC format directly to codes in the PS. For example,this technique could be useful if you wanted to search for a particular field attribute.

fnd_offset

This WORD returns the location of the matching string in the PS. If no match is found, this parameter is set to 0xFFFF.

Errors

NOT_FOUND_ERR This error indicates that the specified string was not found.

PS_UNFMT_ERR	This error indicates that the PS was not formatted.

Send_Keys

Purpose

This 3270 API call sends IBM scan codes to either a 3x74 control unit (for CUT drivers) or a PS component (for DFT and CU drivers). The CU component or the PS component receiving these scan codes maps them to the corresponding character codes for each key.

As a means of conveying data to the host, Send_Keys is relatively slow. It's especially slow if the underlying connectivity means is CUT since the CU must process each keystroke; speed is limited to 10 to 12 keystrokes per second. A much quicker technique is to copy data into the PS (using Copy_To_PS or Copy_To_Field) and then to issue Send_Keys with an AID scan code to prompt the CU to read the PS.

Format

```
Send_Keys (&req_blk, asyncFlag);
```

Parameters

```
WORD     num_keys_to_send;                  /* passed */
BYTE     *keys_bufp;                        /* passed */

WORD     num_keys_sent;                     /* returned */

/* shift key values*/
#define NO_SHIFT                0
#define UP_SHIFT                2
#define ALT_SHIFT               8
```

Definitions

asyncFlag

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC Specify this constant to return control immediately to the caller.

SYNC Specify this constant to prevent control from returning to the caller until the request completes.

num_keys_to_send

This WORD specifies the number of Shift key–scan code pairs that the driver will send to the PS. The number of bytes in the keystroke buffer is twice the value of this parameter.

***keys_bufp**

This POINTER points to the keystroke buffer. Each keystroke contains two bytes of information. The high-order byte specifies a Shift key; the low-order byte indicates a scan code. Use the symbolic values shown in Shift key values in this call to specify the high-order byte.

num_keys_sent

This WORD returns the number of Shift key–scan code pairs successfully sent. The value returned may not equal num_keys_to_send if either an input-inhibited condition occurs or an AID (Attention ID), SYSREQ, or ATTN key precedes the last key in the keystroke buffer. The driver does not send keystrokes following one of these keys.

Shift key values

These values allow you to specify the type of keyboard character, as follows:

NO_SHIFT	Specify this value if you want the scan code is to be sent without a Shift key in effect.
UP_SHIFT	Specify this value if you want the scan code is to be sent with the Up-Shift key in effect.
ALT_SHIFT	Specify this value for you want the scan code to be sent with the ALT Shift key in effect.

Description

Use of IBM scan codes instead of ASCII character codes in the Send_Keys call enables the API to support other languages besides English. The CU component or the PS component maps each scan code to the appropriate character code based upon the customized language.

An application acting in the role of a terminal emulator is responsible for mapping key codes or character codes from the event queue to a Shift key–scan code pair. An application can also generate Shift key–scan code pairs without requiring keyboard input.

The driver checks for an input-inhibited condition before sending the first keystroke and then after each successive keystroke. Should this condition occur, the driver immediately terminates the Send_Keys call and returns INP_INHIBITED_ERR.

To clear most input-inhibited conditions, the initial keystroke should be a RESET. The input-inhibited conditions not cleared by RESET are Time, Printer Busy, Printer Very Busy, and Printer Not Working. Besides RESET, these keys can function under the following input-inhibited conditions:

- Time, SYSREQ, and ATTN are valid. (ATTN, though valid, may still be rejected with an invalid function indication (X–f) in the OIA if the ATTN is inappropriate.)

- Printer Busy, Printer Very Busy, Printer Not Working: DEVICE CANCEL is valid and serves to clear these particular input-inhibited conditions.

The driver does not process keystrokes following an AID in the buffer; the call terminates once the AID is sent.

You don't have to send makeor break scan codes for the modifiers; the driver takes care of the process based upon the setting of the Shift key for each scan code. However, your application can send a scan code for a modifier key by setting the Shift key byte to `NO_SHIFT` and placing the Shift key's scan code in the scan-code byte.

Important Sending scan codes in this manner is appropriate only for a CUT emulation and renders the application incompatible with a DFT or CU emulation.

For information about mapping scan codes to a character set, refer to *IBM 3270 Information Display System Character Set Reference* (GA27-2837) and the *IBM 3174/3274 Control Unit to Device Product Attachment Information* (October16, 1986).

Example The following example, taken from the sample application presented in Chapter 2 of this guide, shows a `case` statement that sends keys to the host.

```
case RC_SEND_KEYS:
    if (dft[session]->req_blk.result != NO_ERR) {

        ErrorMessage("SendKey Return Error",dft[session]->req_blk.result);
        SysBeep(1);
    }

    if (key_q_index) { /* keys strokes are buffered */

        dft[session]->req_blk.sendkey.num_keys_to_send = key_q_index;
        key_q_index = 0;
        dft[session]->req_blk.sendkey.keys_bufp = (BYTE *) kbuf_current;

        if (err = Send_Keys(&(dft[session]->req_blk),ASYNC)) {
              ErrorMessage("GLUE Send Keys Error",err);
              ClearConnect();
              return 0;
        }
        kbuf_current = kbuf_q[kbuf_toggle ^= 1]; /* switch buffers */

    }
    else { /* key strokes not buffered */
              dft[session]->last_request = 0;
    }
    break;
```

Errors

`INP_INHIBITED_ERR`	This error indicates that an input-inhibited condition either existed prior to any keystrokes being sent or developed as they were sent. You can check the number of keys sent by examining the `num_keys_sent` parameter.
`CU_NO_RSP_ERR`	This error indicates that the CU hasn't acknowledged a sent keystroke. The CU has failed, or the connection between the device and CU has broken, or the coax hardwareor firmware failed.

Send_Passthru_Data

Purpose

This 3270 API call enables the application to send structured field data directly to the host, bypassing the PS component in the driver.

Important Only DFT and CU drivers can support Send_Passthru_Data.

The driver does not examine the data to be sent. Your application must create one or more valid structured fields in their entirety in the send buffer.

In the SNA environment, the data passed becomes a request unit (RU) or a series of RUs to which the SNA LU 2 function attaches the appropriate transmission and request headers. The `data_end` parameter is a signal to the driver to mark the last RU formatted from the passed data buffer as a last-in-chain (LIC) RU.

You can use Send_Passthru_Data and Get_Passthru_Data to send and receive D0 structured fields with the IND$FILE 3270 PC file-transfer method.

For an LU type 1 session, you can use this call to send inbound IPDS structured fields (such as an Acknowledge Reply).

Format

```
Send_Passthru_Data (&req_block, asyncFlag);
```

Parameters

```
BYTE    *send_bufp;              /* passed */
WORD    num_bytes_to_send;       /* passed */
BYTE    data_end;                /* passed */
```

Definitions

`asyncFlag`

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC Specify this constant to return control immediately to the caller.

SYNC Specify this constant to prevent control from returning to the caller until the request completes.

`*send_bufp`

This POINTER points to the buffer containing the bytes to be sent.

num_bytes_to_send

This WORD specifies the number of bytes of data to send.

data_end

This BYTE, if set to TRUE, indicates that the buffer to be sent completes transmission of a structured field or a number of structured fields. In either case, the data present in the buffer must contain the last portion of a structured field or one or more complete structured fields.

Errors

DATA_XFER_TRUNC_ERR	This error indicates that the driver did not send the number of bytes specified by the num_bytes_to_send parameter.
NO_HOST_SESS_ERR	This error indicates that the underlying host session no longer exists.
STATE_ERR	This error indicates that the call was inappropriate; that is, if a previous Get_Passthru_Data call completed with a GPD_END_REPLY notification, the application must issue a Post_Passthru_Reply call before it issues a Send_Passthru_Data call.

Send_Prt_Control

Purpose

This 3270 API call enables the application to send the SCS printer controls PA1, PA2, and Cancel.

PA1 and PA2 either signal a host application of the occurrence of an event or act as a prompt for a particular action. The host application defines their meaning. A PA key sent while one is already outstanding is ignored, and the driver does not return an error notification.

Cancel causes the driver to terminate the current chain of data being sent from the host application. The next or currently outstanding Get_LU1_Prt_Data request have `data_end` set to `GLP_END` if the application sends Cancel while receiving a chain. If the application is not receiving a chain when it sends Cancel, Cancel has no effect, and the driver does not return an error notification.

Format

```
Send_Prt_Control(&req_block, asyncFlag);
```

Parameters

```
BYTE    ctrl;                   /* passed */
```

Definitions

`asyncFlag`

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC	Specify this constant to return control immediately to the caller.
SYNC	Specify this constant to prevent control from returning until the request completes.

`ctrl`

This BYTE specifies the type of control to be sent, as follows:

SPC_PA1	Specify this constant to send a PA1 printer control.
SPC_PA2	Specify this constant to send a PA2 printer control.
SPC_Cancel	Specify this constant to send a Cancel printer control.

Errors

NO_HOST_SESS_ERR	This error indicates that the underlying host session no longer exists.
PS_INACTIVE_ERR	This error indicates that the application never activated the specified printer session.

SESS_TYPE_ERR	This error indicates that the application sent the print request to the wrong type of session; the request is valid only for LU 1 printer sessions.
STATE_ERR	This error indicates that a previous Send_Prt_Control call is still outstanding; the driver ignores this request.

Set_Cursor

Purpose

This 3270 API call moves the cursor directly to a specified position in the PS. You can also use this call to select a field by positioning the cursor and then sending a Cursor Select scan code via a Send_Keys call.

❖ *Note:* You can also set the cursor by issuing a Send_Keys call with individual horizontal and vertical keystrokes, although that technique is usually more cumbersome than setting the cursor directly.

Format

```
Set_Cursor (&req_blk, asyncFlag);
```

Parameters

```
WORD    offset;                 /* passed */
```

Definitions

asyncFlag

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC	Specify this constant to return control immediately to the caller.
SYNC	Specify this constant to prevent control from returning to the caller until the request completes.

offset

This WORD specifies the location in the PS to which the cursor will be moved as an offset from the beginning of the PS.

Errors

INP_INHIBITED_ERR	This error indicates that an input-inhibited condition existed; the write operation to the PS was disallowed.
PARM_ERR	This error indicates that the specified offset was out of range for the current screen size.

❖ *CUT note:* For a CUT driver, the PARM_ERR error does not mean that the application made a mistake. CUT drivers don't notify the application the application of a screen size change for Models 3 and 4 when they revert to their default 24 x 80 screen size. If the application attempts to set the cursor outside of the 24 x 80 area when the default screen size is in effect, the driver returns PARM_ERR.

Set_Color_Support

Purpose

This 3270 API call allows your application to change the color support mode for an existing session. The color support mode is for a session is initially specified in the Connect_To_PS call; Set_Color_Support allows your application to change the color mode while the session is in progress.

After your application performs a Set_Color_Support call, all subsequent API calls that retrieve the DAB will show the change in color settings. Also, changing the color support mode causes the entire DAB to be returned in the next Get_Update call that requests DAB update records.

Format

```
Set_Cursor (&req_blk, asyncFlag);
```

Parameters

```
BYTE    color_supp;              /* passed */
```

Definitions

asyncFlag

This flag may be one of the following (see "Checking for a Completed Request" in Chapter 1 for more information):

ASYNC Specify this constant to return control immediately to the caller.

SYNC Specify this constant to prevent control from returning to the caller until the request completes.

color_supp

This BYTE specifies the type of color support the application needs, which affects how the driver sets the color bits in the DAB.

❖ *Note:* If the presentation space is unformatted, the color returned by the driver is always green unless the SCS_NO_COLOR constant is specified.

You can specify the following color support modes:

SCS_NO_COLOR Specify this constant to always set the DAB color bits to not support color (x'000').

SCS_2_COLOR Specify this constant for two base colors and no extended colors. The driver examines only the field attribute to determine the color setting for the DAB. The driver returns CK_WHITE if the intensified bit is set in the field attribute; if the bit is not set, the driver returns CK_GREEN.

SCS_4_COLOR Specify this constant for four base colors and no extended colors. The driver examines only the field attribute to determine the color setting for the DAB, and returns one of the following colors:

CK_GREEN	Unprotected, normal intensity
CK_RED	Unprotected, intensified
CK_BLUE	Protected, normal intensity
CK_WHITE	Protected, intensified

SCS_2_COLOR_EXT

Specify this constant to support extended colors with two base colors.

The color setting in the DAB is a copy of the EAB color setting with this exception: When extended color is in effect (base color override bit set to 1), and the color bits in the EAB are set to the default values, the driver examines the field attribute and sets the DAB to white for intensified fields and green for non-intensified fields. When base color is in effect (base color override bit reset to 0), the EAB is ignored and the only two colors set are white and green, in the same fashion as for SCS_2_COLOR.

SCS_4_COLOR_EXT

Specify this constant to support extended colors with four base colors. This causes much of the same behavior as SCS_2_COLOR_EXT except that, when base color is in effect (that is, the base color override bit is reset), colors are set in the same fashion as for SCS_4_COLOR.

Errors

INP_INHIBITED_ERR	This error indicates that an input-inhibited condition existed; the write operation to the PS was disallowed.
PARM_ERR	This error indicates that the specified offset was out of range for the current screen size.

Term_3270_API

Purpose This 3270 API call shuts down the interface.

Format

```
Term_3270_API(api_vars);
```

Parameters None

Example The following example, taken from the sample application presented in Chapter 2 of this guide,

```
if (err = Open_Host_Connection(&(dft[0]->req_blk))) {
    ErrorMessage("Open_Host_Connection Error",err);
    Term_3270_API(api_vars);
    return 0;
    }
```

Errors None

Chapter 4

Apple 3270 API Device Drivers

Most 3270 device drivers are distributed as system files containing an 'INIT' 31 resource. During startup, a Macintosh looks at all the files in the System Folder to determine if any file has an 'INIT' 31 resource. If an 'INIT' 31 resource is found, the resource is loaded, executed, and closed, and the search for 'INIT' 31 resources continues. Thus, a 3270 device driver must be contained in a single file that also contains the 'INIT' resource.

❖ *Note:* The 'INIT' 31 system resource type is described in detail in *Macintosh Technical Note #14.*

The user installs an 'INIT' 31 resource by simply dragging the file containing the driver and 'INIT' resource into the System Folder. To un-install the 3270 device driver, the user drags the file out of the System Folder. Thus, if your 3270 API device driver is 'INIT' 31, you don't need to write and distribute an installation utility.

The interface supports drivers that reside permanently in the system heap and those that live temporarily in the application heap. Drivers running on the system heap should be placed in an INIT file that gets loaded at system boot-up. Drivers that live only for the duration of the application will run in the application heap. A driver of this type should be stored in an ordinary resource file. In both cases, the files are placed in the startup volume's System Folder.

An Apple 3270 Device Driver must conform to the rules for Macintosh device drivers, as described in *Inside Macintosh, Volume II,* in the Device Manager chapter, and in Chapter 9 in *Designing Cards and Drivers for the Macintosh*, and *Macintosh Technical Note #14*

Besides following the rules in those manuals, and adhering to the 3270 API interface as detailed in this manual, Apple 3270 Device Drivers have some special characteristics. These characteristics are described in detail in this chapter.

Input inhibited conditions

Your driver must check for an input inhibited condition prior to sending the initial keystroke, and after each keystroke. If an input inhibited condition exists prior to sending the initial keystroke, the initial keystroke should be a RESET.Others keys, with two exceptions, will cause the call to be terminated immediately with an INP_INHIBITED_ERR. The exceptions are the SYSREQ and ATTN keys which are valid even when an input inhibited condition is in effect.

During the course of sending keystrokes, if an input inhibited condition arises, the driver must terminate the call.

Your driver must not send keystrokes that follow an AID, SYSREQ, or ATTN in the keystroke buffer.

Supporting API calls

A driver which supports an API call must be capable of implementing the call's default mode of operation. It may optionally support none, some, or all of the modifiers. If a request has an unsupported modifier set, the driver must return MOD_UNSUPP_ERR.

To mark calls as supported or, set the high-order bit (bit 15) of the word corresponding to the call in an array of 16-bit words, as follows:

Word 0	Open_Host_Connection
Word 1	Close_Host_Connection
Word 2	Get_Host_Connection_Info
Word 3	Connect_To_PS
Word 4	Disconnect_From_PS
Word 5	Send_Keys
Word 6	Copy_To_PS
Word 7	Copy_From_Buffer
Word 8	Copy_To_Field
Word 9	Copy_From_Field
Word 10	Copy_OIA
Word 11	Search_String
Word 12	Find_Field
Word 13	Get_Update
Word 14	Get_Cursor
Word 15	Set_Cursor
Word 16	Set_Color_Support
Word 17	Send_Passthru_Data
Word 18	Get_Passthru_Data
Word 19	Post_Passthru_Reply
Word 20	Do_Special_Func
Word 21	Activate_Prt_Sess
Word 22	Deactivate_Prt_Sess
Word 23	Get_DSC_Prt_Data
Word 24	Get_LU1_Prt_Data
Word 25	Post_Prt_Reply
Word 26	Send_Prt_Control
Word 27	Check_Session_Bind

Array elements can be accessed by the symbolic names defined for the request codes. (See the list of request codes defined in Appendix C of this manual.)

If your driver supports a call, the driver must support all of the errors documented for the call must be supported by a driver if it supports the call. In addition, other more general errors can also occur, and the driver must be capable of passing those errors on to the application.

A special driver function

The API interface does provide a call just for the driver's use. The Do_Special_Func call allows you to add a special function that doesn't normally exist in the interface. However, don't use the call unless you have to, since it defeatsthe purpose of the API if an application has to concern itself with a lot of details pertinent to a specific driver.

Writing a DFT-CU driver

This section describes anything special that a DFT or CU driver must know about individual API calls. See Chapter 3 for the full specifications of the API calls.

Supporting passthrough data

In order to support the Get_Passthru_Data and Send_Passthru_Data calls, your driver must process structured fields in a serial fashion. Each time an application passes a structured field that requires a reply, your driver should suspend processing for the session until the reply is received.

Close_Host_Connection and DFT-CU drivers

Your driver must always respond to this request and immediately shut down the connection method, regardless of any requests that may be held at the time.

When the application makes this call, the API interface code will issue a `PB_Control` with `csCode = CLOSE_HOST_CONNECTION` and a pointer to the Close_Host_Connection request block in `csParam[0]` and `csParam[1]`.

Connect_To_PS and DFT-CU drivers

For the usual case where the `KEEP_SESSION` modifier is not specified, a Connect_To_PS call should result in the following session processing

- For a DFT emulation, the driver presents an AEDV (Offline) status to the CU. This prompts the CU to issue an Unbind and/or Notify if the host protocol is SNA. If the host protocol is local non-SNA, the CU returns a Unit Check when the session is selected. The driver then sends an AEDV (Online) status.
- For a CU emulation, the driver issues an Unbind and/or Notify indicating device unavailability to the host if the host protocol is SNA. If the host protocol is local non-SNA, the CU returns a Unit Check when the session is selected. The driver then issues a Notify indicating device availability.

If the DO structured field ID (0x0F02) is specified in the array pointed to by the `type_pass_datap` parameter, the driver should inspect each DO structured field and take appropriate action based upon the ID field before passing the field on to the application; for example, a destination ID of 0 should cause the driver to stop forwarding subsequent structured field to the application.

Deactivate_Prt_Sess and DFT-CU drivers

Your driver must always respond to this request and immediately shut down the connection method, regardless of any requests that may be held at the time.

Disconnect_From_PS and DFT-CU drivers

Your driver must always respond to this request and immediately terminate the connection to the PS, regardless of any requests that may be held at the time. Terminatethe connection by taking action in one of the following ways, depending upon the type of driver:

- For a DFT driver, present an AEDV (offline) status to the CU, which prompts the CU to do one of the following:

  If the host protocol is SNA, the CU issues an Unbind and/or a NOTIFY.

  If the host protocol is local non-SNA, the CU returns a Unit Check when the session is selected.
- For a CU driver, send an UNBIND and/or a NOTIFY to the host if the host protocal is SNA. If the host protocol is local non-SNA, the CU returns a Unit Check when the session is selected.

Get_Host_Connection_Info and DFT-CU drivers

The `misc` parameter for this call allows you to supply any information you wish, up to a limit of 8 bytes.

For this call, your driver will need to process `drvr_type`, `io_compl_supp`, and `port_map`. The other parameters should be left for the PS. Only the `port_info` array element that is associated with the card (obtain the port ID via a GetCard call) should be filled in.

`drvr_type` indicates if the driver supporting the connection is temporary (`GI_TEMP_DRVR`), residing in the application's heap, or is permanent (`GI_PERM_DRVR`)), residing in the system heap.

When the application makes this call, the API interface code will issue a `PB_Control` with `csCode = GET_HOST_CONNECTION_INFO` and a pointer to the Get_Host_Connection_Info request block in `csParam[0]` and `csParam[1]`.

Get_LU1_Prt_Data and DFT-CU drivers

When the underlying host protocol for this call is SNA, the buffer size in this call has implications at the protocol level: the smaller the buffer, the fewer and/or smaller RU's can be stored and the longer the driver must wait to send a pacing response. A 4K buffer is the minimum recommended size. A 32K buffer is overkill. Refer to the *3174 Functional Description - SNA Protocol* - Pacing (LU type 1) and RU Lengths sections for further information.

When an FMH, structured field,.or last portion of a regular chain of data is transferred to the application, the driver should set `data_end` to `GLP_END_REPLY`. The driver can withhold a pacing response to prevent its buffer from overflowing while a response is forthcoming from the application If the application responds with a non-zero sense code, the driver should dispose of the rest of the chain (buffered and/or forthcoming from the host).

You should set the end_job parameter to TRU when End Bracket (EB) has been detected and the last segment in the last RU of the chain is passed to the application.

If your driver can support FMHs and structured fields, perform the following tasks:

- If the FI bit is set on an MIC or LIC RU, reject the RU and abort the current chain. Respond to the next Get_LU1_Prt_Data request by setting the `data_end` parameter to the `GLP_END` constant to indicate that the data unit was aborted.
- Ensure that a Read Partition Query structured field in the chain, that the CDI is present on the last-in-chain RU, and that EB is not set for the chain. Otherwise, recject the chain with a 0x0829 sense code and set the `data_end` parameter to the `GLP_END` constant to indicate that the data unit was aborted.

Open_Host_Connection and DFT-CU drivers

When the application makes this call, the API interface code will, after issuing a `PB_Open`, issue a `PB_Control` with `csCode` OPEN_HOST_CONNECTION and a pointer to the Open_Host_Connection request block in `csParam[0]` and `csParam[1]`.

Post_Prt_Reply and DFT-CU drivers

A non-SNA driver must map the sense code to an Op-Check and a Sense byte if `sense_code` is non-zero.

Send_Keys and DFT-CU drivers

The driver must check for an input inhibited condition prior to sending the initial keystroke and after each keystroke. The call terminates when an input inhibited condition arises during the course of sending keystrokes.

Send_Passthru_Data and DFT-CU drivers

Send_Passthru_Data can be supported only by a DFT/SNA driver. Data sent must be bypassed by the Presentation Services function in the driver; and sent directly to the SNA LU 2 function. The data passed becomes a RU to which the SNA LU 2 function attaches the appropriate TH/RH.

Among other possibilities, this call and the Get_Passthru_Data call are intended to support the use of Destination/Origin (D0) Structured Fields. An example of an application that employs the D0 Structured Field Protocol is the IND$FILE 3270 PC file transfer method.

Writing a CUT driver

This section describes anything special that a CUT driver must know about individual API calls. See Chapter 3 for the full specifications of the API calls.

Close_Host_Connection and CUT drivers

Your driver must always respond to this request and immediately shut down the connection method, regardless of any requests that may be held at the time.

When the application makes this call, the API interface code will issue a `PB_Control` with `csCode = CLOSE_HOST_CONNECTION` and a pointer to the Close_Host_Connection request block in `csParam[0]` and `csParam[1]`.

Connect_To_PS and CUT drivers

For the usual case where the `KEEP_SESSION` modifier is not specified, a Connect_To_PS call should result in the driver presenting POR to a Poll from the CU.

Disconnect_From_PS and CUT drivers

Your driver must always respond to this request and immediately terminate the connection to the PS, regardless of any requests that may be held at the time. Terminate a conection by no longer responding to poll commands from the CU.

Get_Host_Connection_Info and CUT drivers

The `misc` parameter for this call allows you to supply any information you wish, up to a limit of 8 bytes.

For this call, your driver will need to process `drvr_type`, `io_compl_supp`, and `port_map`. The other parameters should be left for the PS. Only the `port_info` array element that is associated with the card (obtain the port ID via a GetCard call) should be filled in.

`drvr_type` indicates if the driver supporting the connection is temporary (`GI_TEMP_DRVR`), residing in the application's heap, or is permanent (`GI_PERM_DRVR`)), residing in the system heap.

When the application makes this call, the API interface code will issue a `PB_Control` with `csCode = GET_HOST_CONNECTION_INFO` and a pointer to the Get_Host_Connection_Info request block in `csParam[0]` and `csParam[1]`.

Open_Host_Connection and CUT drivers

When the application makes this call, the API interface code will, after issuing a `PB_Open`, issue a `PB_Control` with `csCode` OPEN_HOST_CONNECTION and a pointer to the Open_Host_Connection request block in `csParam[0]` and `csParam[1]`.

Send_Keys and CUT drivers

The driver must check for an input inhibited condition prior to sending the initial keystroke and after each keystroke. The call terminates when an input inhibited condition arises during the course of sending keystrokes.

Appendixes

Appendix A

Error Codes

This appendix describes the API error codes returned in the `result` field of the API request block. A driver (or related software) is responsible for filling in `result`. The majority of error codes returned by a driver fall in the generic category; a driver should return relatively few driver-specific codes.

The high-order byte of `result` determines the error category, as follows:

Generic	0x80
API Glue	0x81
Apple CUT/DFT driver	0x90
Apple CUT card	0x91
APPLE DFT card	0x92
AppleLine driver	to be assigned
Simware driver	to be assigned
Avatar CUT driver	to be assigned
Avatar DFT driver	to be assigned
DCA CUT driver	to be assigned
DCA DFT driver	to be assigned
CXI CUT driver	to be assigned
CXI DFT driver	to be assigned

The low-order byte of `result` contains the error code. The error categories and codes are shown in the tables in this appendix.

Table A-1
Generic error codes

Name	Value	Description
NO_ERR	0x0000	Request completed successfully
RSP_PENDING	0x0001	Reponse pending; changed by driver when processing of request is complete
REQ_CODE_ERR	0x8002	Invalid API request code
CONN_ID_ERR	0x8003	Invalid connection ID
PORT_ID_ER	0x8004	Invalid port ID
PS_UNSUPP_ERR	0x8005	Driver does not support specified PS
SRCH_STR_NOT_FND_ERR	0x8006	No matching string found
PS_NOT_CONNECTED_ERR	0x8007	Specified PS not connected
DATA_XFER_TRUNC_ERR	0x8008	Data passed to or from application was truncated
PS_UNAVAIL_ERR	0x8009	PS in use or no more PSs available
END_OF_PS_ERR	0x800A	Beginning or end of PS encountered during copy or search
INP_INHIBITED_ERR	0x800B	Input-inhibited condition exists; write operation to PS disallowed
HOST_RSP_PENDING_ERR	0x800C	AID key sent to host; X Clock/System present
PARM_ERR	0x800D	Invalid request parameter
DATA_ERR	0x800E	Invalid data passed
GLUE_ERR	0x800F	Internal API error
HARDWARE_ERR	0x8010	Hardware failure detected by driver
REQ_OUTSTANDING_ERR	0x8011	Request rejected because another is outstanding
LOST_DATA_ERR	0x8012	Driver lost data because of buffer overflow
FIELD_NOT_FND_ERR	0x8013	No field matched search criteria
DRVR_ERR	0x8014	Internal driver error
CONFIG_ERR	0x8015	Invalid configuration information
EAB_UNSUPP_ERR	0x8016	EAB not supported by driver
DAB_UNSUPP_ERR	0x8017	DAB not supported by driver

DABE_UNSUPP_ERR	0x8018	DABE not supported by driver
MOD_UNSUPP_ERR	0x8019	modifer(s) specified not supported by driver
WRITE_PROT_FLD_ERR	0x801A	Write operation attempted into a protected field
CONN_ALREADY_OPEN_ERR	0x801B	Connection to driver or PS already present
CONN_NOT_OPEN_ERR	0x801C	Open_Host_Connection call not previously issued
PSS_UNSUPP_ERR	0x801D	Programmed Symbol Sets not supported by driver
TIMEOUT_ERR	0x801E	Request timed out
CU_NO_RSP_ERR	0x801F	CU not responding or device-to-CU connection break
WRITE_ATTR_ERR	0x8020	Write operation attempted to overwrite an attribute
PS_UNFMT_ERR	0x8021	PS currently unformatted
SPEC_FUNC_FAILED_ERR	0x8022	Do_Special_Func request failed
STATE_ERR	0x8023	Invalid request for current state
CHG_TO_DEFAULT_SCR_ERR	0x8024	Screen changed to default size
CHG_TO_ALT_SCR_ERR	0x8025	Screen changed to alternate size
NO_HOST_SESS_ERR	0x8026	Host session no longer exists
SESS_TYPE_ERR	0x8027	Invalid request for session type
REQ_UNSUPP_ERR	0x8028	Request unsupported by driver

Table A-2
API interface error codes

Name	Value	Description
GLU_RES_FILE_ERR	0x8101	Resource file error
GLU_DRVR_OPEN_ERR	0x8102	Driver could not be opened
GLU_VARS_ERR	0x8103	Invalid handle to glue variables

Table A-3
Apple CUT/DFT driver error codes

Name	Value	Description
ADVR_SLOT_NOT_CNFG_ERR	0x9001	Slot not configured; application did not request that that slot be downloaded
ADVR_68K_DNLD_ERR	0x9002	Download of 68000 failed; check whether or not the APPLE DFT 2 file is present in the System folder. If that file is present, report the error to Apple.

ADVR_8344_DNLD_ERR	0x9003	Download of DP8344 download ; check whether or not the APPLE DFT 3 file is present in the System folder. If that file is present, report the error to Apple.
ADVR_INIT_68K_ERR	0x9004	68000 initialization failed; report error to Apple
ADVR_INIT_8344_ERR	0x9005	DP8344 initialization failed; report error to Apple
ADVR_SEND_ERR	0x9006	MRDOS Send failed; report error to Apple
ADVR_RCV_ERR	0x9007	MRDOS Receive failed; report error to Apple
ADVR_PS_TASK_ERR	0x9008	Presentation services task does not exist; report error to Apple
ADVR_GET_MSG_ERR	0x9009	MRDOS GetMsg failed; report error to Apple
ADVR_RES_FILE_ERR	0x900A	resource file error; report error to Apple
ADVR_NO_ICCM_ERR	0x900B	local ICCM does not exist; report error to Apple
ADVR_FILE_ERR	0x900C	file error encountered; report error to Apple

Table A-4
Apple CUT card error codes

Name	Value	Description
ACUT_STATE_ERR	0x1101	internal state machine error

more to be defined

Table A-5
Apple DFT card error codes

Name	Value	Description

to be defined

Table A-6
APPLELINE error codes

Name	Value	Description

to be defined

Table A-7
SIMWARE error codes

Name	Value	Description

to be defined

Table A-8
AVATAR CUT error codes

Name	Value	Description

to be defined

Table A-9
AVATAR DFT error codes

Name	Value	Description

to be defined

Table A-10
DCA CUT error codes

Name	Value	Description

to be defined

Table A-11
DCA DFT error codes

Name	Value	Description

to be defined

Table A-12
CXI CUT error codes

Name	Value	Description

to be defined

Table A-13
CXI DFT error codes.

Name	Value	Description

to be defined

Appendix B

Control Key Codes

Table B-1 lists the codes for one of the most common keyboards (***which one???***).

Table B-1
3270 DFT-CU control key codes

Control key	Definition	Value
APL on or off	CK_APL_ON_OFF	0x41
Attn	CK_ATTN	0x28
Backtab	CK_BACK_TAB	0x37
Clear	CK_CLEAR	0x11
Cursor Left	CK_CURS_LEFT	0x33
Cursor Right	CK_CURS_RIGHT	0x34
Cursor Up	CK_CURS_UP	0x31
Cursor Down	CK_CURS_DOWN	0x32
Cursor Select	CK_CURS_SELECT	0x2b
Delete	CK_DELETE	0x1e
Device Cancel	CK_DEV_CNCL	0x27
Dup	CK_DUP	0x20
Enter	CK_ENTER	0x01
Erase EOF	CK_ERASE_EOF	0x2d

Table B-1 (continued)
3270 DFT-CU control key codes

Control key	Definition	Value
Erase Input	CK_ERASE_INP	0x0f
Extended Selection	CK_EXT_SELECT	0x40
Field Mark	CK_FIELD_MARK	0x1f
Home	CK_HOME	0x39
Ident	CK_IDENT	0x2a
Insert	CK_INSERT	0x0e
New Line	CK_NEW_LINE	0x3a
PA1	CK_PA1	0x21
PA2	CK_PA2	0x22
PA3	CK_PA3	0x23
PF1	CK_PF1	0x02
PF2	CK_PF2	0x03
PF3	CK_PF3	0x04
PF4	CK_PF4	0x05
PF5	CK_PF5	0x06
PF6	CK_PF6	0x07
PF7	CK_PF7	0x08
PF8	CK_PF8	0x09
PF9	CK_PF9	0x0a
PF10	CK_PF10	0x0b
PF11	CK_PF11	0x0c
PF12	CK_PF12	0x0d
PF13	CK_PF13	0x12
PF14	CK_PF14	0x13
PF15	CK_PF15	0x14
PF16	CK_PF16	0x15
PF17	CK_PF17	0x16

Table B-1 (continued)
3270 DFT-CU control key codes

Control key	Definition	Value
PF18	CK_PF18	0x17
PF19	CK_PF19	0x18
PF20	CK_PF20	0x19
PF21	CK_PF21	0x1a
PF22	CK_PF22	0x1b
PF23	CK_PF23	0x1c
PF24	CK_PF24	0x1d
Print	CK_PRINT	0x26
Reset	CK_RESET	0x29
SysReq	CK_SYSREQ	0x30
Tab	CK_TAB	0x38
Test	CK_TEST	0x24
Text on or off	CK_TEXT_ON_OFF	0x36
Field Inherit		
Color	CK_FI_COLOR	0x50
Extended highlighting	CK_FI_EXTHI	0x51
Symbol set	CK_FI_SYMSET	0x52
Color		
Blue	CK_BLUE	0x53
Red	CK_RED	0x54
Pink	CK_PINK	0x55
Green	CK_GREEN	0x56
Turquoise	CK_TURQ	0x57
Yellow	CK_YELLOW	0x58
White	CK_WHITE	0x59
Black	CK_BLACK	0x5A

Table B-1 (continued)
3270 DFT-CU control key codes

Control key	Definition	Value
Extended Highlighting		
Reverse video	CK_REVERSE	0x5B
Blink	CK_BLINK	0x5C
Underscore	CK_UNDERSC	0x5D
Symbol Set - A	CK_SYM_A	0x5E
Symbol Set - B	CK_SYM_B	0x5F
Symbol Set - C	CK_SYM_C	0x60
Symbol Set - D	CK_SYM_D	0x61
Symbol Set - E	CK_SYM_E	0x62
Symbol Set - F	CK_SYM_F	0x63

Appendix C

Request Codes

Table C-1 lists the actual request codes and their values, which you can use for debugging.

Table C-1
3270 API request codes

Request code	Value
RC_OPEN_HOST_CONNECTION	0x01
RC_CLOSE_HOST_CONNECTION	0x02
RC_GET_HOST_CONNECTION_INFO	0x03
RC_CONNECT_TO_PS	0x04
RC_DISCONNECT_FROM_PS	0x05
RC_SEND_KEYS	0x06
RC_COPY_TO_PS	0x07
RC_COPY_FROM_BUFFER	0x08
RC_COPY_TO_FIELD	0x09
RC_COPY_FROM_FIELD	0x0A
RC_COPY_OIA	0x0B
RC_SEARCH_STRING	0x0C
RC_FIND_FIELD	0x0D
RC_GET_UPDATE	0x0E

```
RC_GET_CURSOR                0x0F
RC_SET_CURSOR                0x10
RC_SET_COLOR_SUPPORT         0x11
RC_SEND_PASSTHRU_DATA        0x12
RC_RECEIVE_PASSTHRU_DATA     0x13
RC_POST_PASSTHRU_REPLY       0x14
RC_DO_SPECIAL_FUNC           0x15
RC_ACTIVATE_PRT_SESS         0x16
RC_DEACTIVATE_PRT_SESS       0x17
RC_GET_DSC_PRT_DATA          0x18
RC_GET_LU1_PRT_DATA          0x19
RC_POST_PRT_REPLY            0x1A
RC_SEND_PRT_CONTROL          0x1B
RC_CHECK_SESSION_BIND        0x1C
RC_SET_COLOR_SUPPORT         0x1C
```

Table C-2 provides a list of definitions for programmers who prefer to write more terse code. Such definitions allow an application to use fewer characters to access a field within a particular request. For example, you could use the following statement:

```
blk.openhc.open_type = WARM;
```

as a short form of the following statement:

```
blk.req.open_host_connection.open_type = WARM;
```

If you wish, you can also add your own defines to shorten other names.

Table C-2
3270 API alternate defines

Short form	Long form
openhc	req.open_host_connection
closehc	req.close_host_connection
getinfo	req.get_host_connection_info
connps	req.connect_to_ps
discps	req.disconnect_from_ps

sendkey	req.send_keys
cpytops	req.copy_to_ps
cpyfbuf	req.copy_from_buffer
cpytfld	req.copy_to_field
cpyffld	req.copy_from_field
cpyoia	req.copy_oia
srchstr	req.search_string
findfld	req.find_field
getupd	req.get_update
getcurs	req.get_cursor
setcurs	req.set_cursor
sndpdata	req.send_passthru_data
rcvpdata	req.receive_passthru_data
spec	req.do_special_func
actprt	req.activate_prt_sess
dactprt	req.deactivate_prt_sess
getdsc	req.get_dsc_prt_data
getlul	req.get_lul_prt_data
postprt	req.post_prt_reply
sndpctl	req.send_prt_control
chkbind	req.check_session_bind

Glossary

(***Writer's note: the definitions for the glossary items will be included in the next draft.***)

API interface routines: Definition

API request block: Definition

DAB: Definition

DBC: Definition

Device driver: Definition

Display attribute buffer: Definition

EAB: Definition

Extended attribute buffer: Definition

Formatted presentation space: Definition

Glue: Definition

Host: Definition

Input inhibited: Definition

Modifiers: Definition

Offset: Definition

OIA: See Operator Information Area

Operator Information Area: Status line shown at the bottom of the screen on a 3270-type terminal.

Presentation space: Definition

Unformatted presentation space: Definition

Keyword: Definition

Bibliography

(***Writer's Note: Standard bibliographic information to be supplied when we decide how many of these we want to reference here.***)

IBM 3174/3274 Control Unit to Device Product Attachment Information (Oct 16, 1986)

IBM 3270 Information Display System Character Set Reference (GA27-2837)

IBM 3270 High Level Language Application Program Interface Programming Guide (59X9474)

Inside Macintosh, Volumes I, II, and III.

Inside Macintosh, Volume IV.

Inside Macintosh, Volume V.

Inside Macintosh X-Ref.

Human Interface Guidelines: The Apple Desktop Interface.

Macintosh Programmer's Workshop Reference

MPW C Reference.

Programmer's Introduction to the Macintosh Family.

Technical Introduction to the Macintosh Family.

Index

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using the Apple Macintosh™ Plus and Microsoft® Word. Proof and final pages were created on the Apple LaserWriter® Plus. POSTSCRIPT™, the LaserWriter's page-description language, was developed by Adobe Systems Incorporated.

Text type is ITC Garamond® (a downloadable font distributed by Adobe Systems). Display type is ITC Avant Garde Gothic®. Bullets are ITC Zapf Dingbats®. Program listings are set in Apple Courier, a monospaced font.