Volume 4, Number 4 September 1986

IBM PC 3270 Emulation Program Presentation Space
Advanced Program-to-Program Communications
Revisable-Form Text Document Content Architecture
Document Interchange Architecture
IBM Enhanced Connectivity Facilities Introduction
IBM PC Interrupt Sharing Protocol

IBM Personal Computer Seminar Proceedings

The Publication for Independent Developers of Products for IBM Personal Computers

Published by International Business Machines Corporation
Entry Systems Division

Changes are made periodically to the information herein; any such changes will be reported in subsequent Proceedings.

It is possible that this material may contain reference to, or information about IBM products (machines and programs), programming or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such products, programming or services in your country.

IBM believes the statements contained herein are accurate as of the date of publication of this document. However, IBM makes no warranty of any kind with respect to the accuracy or adequacy of the contents hereof.

This publication could contain technical inaccuracies or typographical errors. Also, illustrations contained herein may show prototype equipment. Your system configuration may differ slightly. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever.

All specifications are subject to change without notice.

Contents

Introduction and Welcome

These are the Proceedings of the IBM Personal Computer Seminar, designed for independent developers of products for IBM Personal Computers. The purpose of these Proceedings is to aid you in your development efforts by providing relevant information about new product announcements and enhancements to existing products.

Throughout these Proceedings, the term IBM Personal Computer and the term family of IBM Personal Computers address the IBM Personal Computer, the IBM Personal Computer XT, the IBM PC*jr*, the IBM *Portable* Personal Computer, the IBM Personal Computer AT and the IBM PC Convertible (referred to as PC Convertible).

Purpose

What is our purpose in issuing a publication such as this? It is quite simple.

The IBM Personal Computer family is a resounding success. We've had a lot of help in achieving this success, and much of it came from the independent developers.

As you proceed with your development, do you at times wish for some bit of information or direction which would make the job easier? Information which IBM can provide? This is the type of information we want to make available to you.

Since we want to be assured of giving you the information you need, we ask you to complete the questionnaire which appears at the end of these Proceedings. Your response to this questionnaire will be taken into account in preparing the content of future issues, as well as the content of seminars we will present at microcomputer industry trade shows.

Topics

The following list gives a general indication of the topics we plan to cover in future seminars and include in the IBM Personal Computer Seminar Proceedings:

- Information exchange forum — letters to the editor format
- Development tools — languages, database offerings
- Compatibility issues
- New devices — capacities and speeds
- System capacities — disk and memory
- Enhancements in maintenance releases
- Tips and techniques
- New system software
- Hardware design parameters
- Tips on organizing and writing documents for clear and easy reading
- Changes to terms and conditions

IBM PC 3270 Emulation Program Presentation Space API

The IBM PC 3270 Emulation Program allows an IBM Personal Computer to emulate an IBM 3274 Control Unit with an attached IBM 3278/79 Display Station and an IBM 3287 Printer in varying configurations, facilitating standalone and network operation. Its 3270 display device emulator provides a presentation space, Application Programming Interface (API), to allow user-written application programs to interface with the terminal emulation software running in its primary partition. With one exception, described under the "Differences between the APIs" section, the PC 3270 Presentation Space API is a subset of the IBM 3270 PC Control Program, Version 2.00 and Version 3.00.

User-written application programs can run either in the PC 3270 Alternate (or DOS) Field or in a TopView window. TopView is a separately purchased program which provides an environment for running multiple application programs. When the application program is running in a TopView window, it can be set up to run even when in the background.

The PC 3270 Alternate Field is a DOS "window" provided by the PC 3270 program. When an application program is running in the PC 3270 Alternate Field, it is suspended whenever it is in the background.

Functions

The PC 3270 Presentation Space Application Programming Interface provides the following functions:

- Copies data from the presentation space (or display buffer) to an application buffer.
- Copies data from an application buffer to the presentation space (or display buffer).
- Writes keystrokes to the presentation space.
- Queries the state of the 3270 input inhibited indicator.
- Inhibits operator keystrokes to the 3270 display device.
- Queries the current 3270 cursor location.

Each function is achieved via an assembly language interface which defines the actions necessary to issue an API service request. All requests are processed in less than a second. No mechanism is provided to wait for an event to occur before returning from an API request. For this reason, API requests must be issued repetitively in order to determine when an event occurs.

Features

The PC 3270 Presentation Space API has the following features:

- Assembly language interface.
- Quick copy string algorithm.
- Removable API; API memory is freed when PC 3270 is ended.
- Application-initiated inhibit of operator keystrokes to the 3270 display device.
- Alternate (or DOS) Field for running user-written application programs.
- TopView compatibility for running user-written application programs.

Benefits

The PC 3270 Presentation Space Application Programming Interface offers the following benefits:

- Allows a PC programmer to improve or customize the user interface to existing host application programs. For example, the API can take the output (display screen) of a host program and re–display it in an improved format on the application program's display screen.
- Allows a PC programmer to automate often repeated operator procedures. For example, the API can be used to automate the logon procedure.
- Allows a PC programmer to communicate with a user-written host programs in order to transfer information and data. For example, output from a spreadsheet can be downloaded to the PC, processed, and then uploaded back to the host.

Additional API Notes

The PSCAPI.COM Program

PC 3270 provides a removable API. The application program writer should take care to handle situations such as the API not loaded or the API removed while in use. The implementation of the PC 3270 Presentation Space API allows both of these problems to be handled in a similar way. The PC 3270 PS API has a system extension called PSCAPI.COM, which is loaded separately from the rest of the API. It remains resident after it has terminated and provides the function of taking over the API interrupt vector and handling API requests after the API has been removed. When an API request is made after the API has been removed, the PSCAPI.COM program returns an "API not available" return code. User-written application programs should treat this return code as fatal and provide an escape mechanism to gracefully exit from their code.

Even though the PSCAPI.COM program provides this recovery mechanism, the operator may forget to run PSCAPI before running the user-written application program. In anticipation of this, the user written application program should check the interrupt vector for non–zero before assuming the API is available. Even if the interrupt vector is non-zero, it may not point to the PSCAPI.COM program. So, in addition to the above check, the user-written application program also should issue a valid API request and verify that it returns with a good return code before assuming the API is available.

The PSCAPI.COM program takes up very little memory and has virtually no affect on the memory left over for running other application programs.

Synchronization Considerations with a Presentation Space API

Several factors must be considered when waiting for host data to arrive in the presentation space (or display buffer). Is the 3270 display input inhibited? Has all of the host data arrived? Because fragmentation occurs when data is sent to a 3270 device, the application program should wait until input is no longer inhibited and all of the data has arrived. The user application must be able to handle input going in and out of the inhibited state several times before the host data arrives. For a user-written host application, a check sum or CRC could be used to determine when all of the host data has arrived. For existing host applications, searching for a unique character string on the display is usually sufficient.

Differences between the PC 3270 API and the 3270 PC Control Program

The PC 3270 API is a subset of the 3270 PC Control Program, Version 2.00. with the one exception of data key handling on the Write Keystroke service request. More information on this incompatibility is given below. As long as a user application program stays within the subset, it can be run on both products. The following text lists the subset supported by the PC 3270 API and indicates its compatibility with the 3270 PC Control Program:

- Only synchronous requests (BX = X'8020') are supported. The Get Request Complete service is not supported.

- The following Supervisor Service is supported:

  – Name Resolution

    Only the gate names SESSMGR, KEYBOARD, COPY, and OIAM are supported.

- The following Session Information Services are supported:

  – Query Session ID

    Only option code X'01', which obtains the session ID of specified short session name, is supported. Only the data code for the ASCII character 'E' (X'45'), which specifies the short session name is supported. The long session name is never returned nor is it ever used by the PC 3270 Emulation Program. Although the 3270 PC Control Program accepts more options and session names due to its multiple display session support, the PC 3270 API only accepts one due to its single display session support.

  – Query Session Parameters

    The address of the presentation space is not returned. The contents of the presentation space can be obtained and modified using the Copy String service request. Since PC 3270 only supports one setting of these parameters, the session type is always returned as DFT host session. The session characteristics are always returned indicating base attributes and no programmed symbols are supported; the number of rows and columns are always returned as 24 and 80.

  – Query Session Cursor

    Fully supported

- The following Keyboard Services are supported:
  - Connect to Keyboard

    Fully supported
  - Disable Input

    Fully supported
  - Write Keystroke

    Only the non–data (or non–character generating) keys are supported as compatible with the 3270 PC Control Program. Data (or character generating) keys are supported, but are NOT compatible. Although data key support is not compatible, it is similar to that of the 3270 PC Control Program. Instead of using the 3270 PC keystroke scan codes, the PC 3270 API uses ASCII or Device Buffer codes to represent data keys. These are the same codes as used by the Copy String service. Two techniques can be used to write user programs which are to run on both APIs.

    — A check can be made to determine which API is loaded, and an appropriate table can be used to translate keys for the API being used. The PC 3270 API is loaded if an "invalid scan code" return code is generated when a 3270 PC data key scan code (versus an ASCII or Device Buffer code) is passed to the Write Keystroke service.

    — The Copy String service can be used to perform a function similar to the Write Keystroke service.
  - Enable Input

    Fully supported
  - Disconnect from Keyboard

    Fully supported
- The following Copy Service is supported:
  - Copy String

    Only supports two source/target pairs: from application buffer to host session (presentation space) or from host session (presentation space) to application buffer. Supports only the session types for PC (ASCII translation) and host DFT (3270 Device Buffer translation) in the application buffer. ASCII and 3270 Device Buffer code is the same representation of character data as used by the 3270 PC Control Program.
- The following Operator Information Area Services are supported:
  - Read Operator Information Area Group

    Only group 8 indicators are supported. In group 8, only the following indicators are supported: machine check, communications check, program check, and application program has operator input inhibited. All other indicators are supported in that if any is non–zero, input is inhibited for some other reason.

Transmitting Binary Data to a User Written Host Application

If the PC 3270 Presentation Space API is to be used to transmit data between a user-written PC program and a user-written host program, the type of data being transmitted must be considered. Since the presentation space is a representation of the display screen, it is oriented for character data and not binary data. Binary data must be translated to character data before being transmitted across the link. This means the PC application writer must encode and decode binary data when copying to and from the presentation space (or display buffer). The following items must be considered when encoding binary data:

- Several values in the 3270 Interface Code are reserved for control character purposes. These values are used for field attributes and other control codes.
- Several values in the 3270 Interface Code are simply invalid.
- Several values in the 3270 Interface Code represent different characters depending upon the language selected at the host and on the PC 3270.

The 3270 I/O Interface Code for each language can be found in the 3270 Character Set Reference (GA27-2837-7). The 3270 I/O Interface Code is the code used by the host program. The code used by the PC program is either ASCII or 3270 Device Buffer code, depending on the option you select when the API request is coded.

Several schemes can be used to encode binary data to character data. The most common and easiest to implement is one which converts one byte of binary data to two bytes of character data. In this scheme, the half-byte binary values from X'0' to X'9' are

converted to the characters C'0' through C'9'. The half-byte binary values from X'A' to X'F' are converted to characters C'A' through C'F'. There are more complex and efficient schemes but, as long as the amount of data to be transmitted is small, the other schemes will not improve performance drastically. If large amounts of data are to be transmitted, interfacing to the SEND/RECEIVE programs should be considered.

Initiating SEND/RECEIVE File Transfer from an Application Program

A user-written application program can use the SEND/RECEIVE.COM programs to initiate the transfer of files to the host. This is done by issuing the DOS function call to load and execute the SEND.COM or RECEIVE.COM program. Since the PC 3270 SEND/RECEIVE programs set the DOS error level, the successful or unsuccessful status of the file transfer also can be determined. The following excerpt from a program shows how this could be done:

```
;--------------------------------------------------------------------
; EXEC_COMMAND
;
; This subroutine uses the DOS function call 4BH to execute the command
; pointed to by SI (first byte = length).
;--------------------------------------------------------------------
EXEC_COMMAND PROC NEAR
;--------------------------------------------------------------------
; First move the command to be executed to the COMMAND area
;--------------------------------------------------------------------
        PUSH DS                 ;Make ES=DS
        POP  ES                 ;
        MOV  AL,' '             ;Clear the command area
        MOV  CX,COMMAND_END - COMMAND
        MOV  DI,OFFSET COMMAND  ;
        CLD                     ;Clear the direction flag
        REP  STOSB              ;Store blanks
        XOR  CX,CX              ;
        LODSB                   ;Get length of command
        MOV  CL,AL              ;into CX
        MOV  DI,OFFSET COMMAND  ;
        REP  MOVSB              ;Move it to COMMAND below
        MOV  AL,CR              ;Add carriage return
        STOSB                   ;
        SUB  DI,OFFSET COMMAND_START ;Compute length of command
        MOV  AX,DI              ;including '/C' and
        MOV  COMMAND_LENGTH,AL  ;carriage return
;--------------------------------------------------------------------
; Find out where COMAND.COM is from COMSPEC setting in the environment
;--------------------------------------------------------------------
        PUSH CS                 ;Find the Program Segment Prefix
        POP  AX                 ;It starts 100H bytes before CS:0
        SUB  AX,10H             ;
        MOV  ES,AX              ;ES no points to it
        MOV  SI,2CH             ;ES:SI points at environment address
        MOV  AX,ES:WORD PTR [SI]
        MOV  ES,AX              ;ES = segment address of environment
        XOR  DI,DI              ;ES,DI is our environment pointer
FIND_C:
        MOV  CX,100H            ;Limit search
        MOV  AL,COMSPEC         ;AL = C in COMSPEC
        REPNE SCASB             ;Look for a C
        JE   FOUND_C            ;Jump if found C
        ERROR 'Cannot find COMSPEC string in environment'
FOUND_C:
        MOV  CX,7               ;Length of 'OMSPEC='
        MOV  SI,1               ;
TEST_COMSPEC:
        MOV  AL,ES:[DI]         ;Pick next character from enviro
        CMP  AL,COMSPEC[SI]     ;Compare with next char in 'COMSPE
        JNE  FIND_C             ;Go look for next C if unequal
        INC  DI                 ;Increment
        INC  SI                 ;the pointers
        LOOP TEST_COMSPEC       ;Go compare next characters
        MOV  DX,DI              ;All equal - ES:DX points to spec.
```

```
;---------------------------------------------------------------------
; Build parameter list and initialize registers for EXEC function call
;---------------------------------------------------------------------
        MOV  AX,OFFSET COMMAND_LENGTH
        MOV  CMD_PTR,AX          ;Set command pointer in parameter list
        MOV  CMD_PTR+2,DS        ;
        MOV  BX,OFFSET EXEC_LIST ;DS:BX points att parmeter block
        PUSH ES                  ;Exchange ES and DS
        PUSH DS                  ;so that
        POP  ES                  ;DS:DX points at COMSPEC string and
        POP  DS                  ;ES:BX points at parameter block
        MOV  AX,4B00H            ;EXEC function - load and execute
        PUSH DS
        PUSH ES
        MOV  SSREG,SS            ;Save stack registers
        MOV  SPREG,SP
        INT  21H                 ;Call DOS
        MOV  SP,SPREG            ;Restore stack registers
        MOV  SS,SSREG
        POP  ES
        POP  DS
        PUSH ES                  ;Restore DS
        POP  DS                  ;
        JC   EXEC_ERROR          ;Jump if error
        MOV  AH,4DH              ;WAIT function (checks ERRORLEVEL)
        INT  21H                 ;Call DOS
        CMP  AX,0                ;Command completed normally?
        JNE  EXEC_ERROR          ;Jump if not
        RET                      ;Return
EXEC_ERROR:
        ERROR 'The command could not be executed properly'
EXEC_COMMAND ENDP
DSEG    SEGMENT PUBLIC 'DATA'
;---------------------------------------------------------------------
; Data areas for EXEC_COMMAND
;---------------------------------------------------------------------
COMSPEC         DB 'COMSPEC='
COMMAND_LENGTH  DB 97
COMMAND_START   DB '/C '
COMMAND         DB 100 DUP(' ')
COMMAND_END     EQU $
SSREG           DW   0           ;Saved SS register
SPREG           DW   0           ;Saved SP register
;---------------------------------------------------------------------
; Parameter list for the EXEC call
;---------------------------------------------------------------------
EXEC_LIST       DW   0           ;
CMD_PTR         DW   0           ;Pointer to the parameter to be given
                DW   0           ;to COMMAND.COM, i.e. the /C string
FCB1__PTR       DW   0           ;FCB pointer - not used
                DW   0           ;
FCB2_PTR        DW   0           ;FCB pointer - not used
                DW   0           ;
DSEG    ENDS
```

Summary

The IBM PC 3270 Emulation Application Programming Interface (API) allows user written application programs to interface with IBM System/370 type hosts. The API provides many functions, features and benefits which allow the application programmer to improve his or her own productivity and the productivity of other terminal users.

Advanced Program-To-Program Communication for the IBM Personal Computer (APPC/PC)

The APPC / PC Program

APPC/PC is a data communications subsystem for the IBM Personal Computer. APPC/PC provides Systems Network Architecture (SNA) Advanced Program-to-Program Communication (APPC) for application programs that perform distributed transaction processing.

An application program using APPC/PC can communicate with application programs on other systems that support APPC. A transaction program is an application program that uses APPC/PC communication functions.

Transaction programs use APPC/PC verb sequences to communicate with other programs at other SNA nodes. You can regard this set of verbs as a programming language in which you can write conversations. APPC/PC verbs are coded as records, each having a precisely defined syntax. Your transaction programs gain access to APPC/PC facilities by providing verb records to the APPC/PC application program interface (API) that appears as an operating system extension of PC DOS.

Whether you write your own application or use an existing application, you must first set up your IBM Personal Computer to operate in a computer network. APPC/PC simplifies setup by providing configuration menus that help you configure your IBM Personal Computer for connection to an IBM Token-Ring Network, or to an SNA Data Communication Network, using Synchronous Data Link Control (SDLC) non–switched or switched facilities, or to both. A simple direct connection between IBM Personal Computers using SDLC also is possible.

What You Need

To use APPC/PC you need:

- The APPC/PC program product consisting of the APPC/PC Installation and Configuration Guide and two diskettes: the APPC/PC Program Diskette and the APPC/PC Structures and Sample Programs diskette.

  The Guide describes hardware and software requirements and provides information on planning, configuring, and loading APPC/PC.

  The APPC/PC Program Diskette contains:

  - APPC/PC load and unload commands
  - The APPC/PC configuration program
  - Data Link Control files
  - A sample type G conversation table.

  The APPC/PC Structures and Sample Programs diskette contains:

  - APPC/PC assembler verb structures, which are used to write application subsystems and transaction programs
  - A set of sample programs to illustrate the design of an application subsystem and the coding of APPC/PC verbs using the macro assembler verb structures.

- In addition to the APPC/PC program product, you need a language to compile your application subsystem and transaction programs. You can use any language that supports:

  - Issuing software interrupts
  - Setting general registers
  - Building parameter lists
  - Providing addressability to sections of code.

Features Supported

APPC/PC provides the SNA APPC node support as defined in the SNA Format and Protocol Reference Manual.

APPC/PC supports an open API including the following:

- Base APPC (LU Type 6.2/PU Type 2.1)
- Parallel sessions (when attached to peer PU Type 2.1 nodes)
- Synchronization level of None or Confirm
- Mapped conversation support (but no data mapping)
- Peer and boundary function attachment
- Support for conversation and session level security
- Network Management support for ALERT, PDSTATS and general NMVTs
- Normal Response Mode SDLC
  - Primary point–to–point
  - Secondary point–to–point or multipoint
  - Switched support for manual dial, manual answer and auto-answer
- IBM Token-Ring Network support, including connection to S/370 (CICS/VS) via 3725 controller. See April 16, 1986 announcement for more details.
- Compatibility with CICS/VS Version 1.7
- Operation under IBM PC DOS 3.2.

Related Publications

The following publications are related to APPC/PC.

- Advanced Program–to–Program Communication for the IBM Personal Computer Installation and Configuration Guide contains planning and IBM PC set–up information for APPC/PC.
- Advanced Program–to–Program Communications for the IBM Personal Computer Programming Guide, SX27–3757, contains APPC/PC verb descriptions and other information necessary to write application subsystems and transaction programs for APPC/PC.
- Introduction to Advanced Program–to–Program Communication, GG24–1584, contains general information about APPC.
- Systems Network Architecture Concepts and Products, GC30–3072, contains basic information on SNA for those readers wanting either an overview or a foundation for further study.
- Systems Network Architecture Technical Overview, GC30–3073, contains additional details on SNA, especially on functions and control sequences; it bridges the gap between the most elementary overview of SNA and the detailed descriptions of the formats and protocols.
- Systems Network Architecture Transaction Programmer's Reference Manual for LU Type 6.2, GC30–3084, contains reference information on LU Type 6.2 (APPC) verbs for programmers writing transaction programs to run on SNA.
- Systems Network Architecture Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2, SC30–3269, contains information for system programmers and others who need detailed information about SNA LU Type 6.2 (APPC) to adapt a program to function within an SNA network.
- Systems Network Architecture Sessions Between Logical Units, GC20–1868, contains references on SNA formats and protocols for LU types other than Type 6.2.
- Systems Network Architecture Reference Summary, GA27–3136, contains summary information on SNA formats and sequences.
- IBM SDLC General Information, GA27–3093, contains supplementary details of Synchronous Data Link Control.
- IBM Token–Ring Network Introduction and Planning Guide, GA27–3677, contains planning information for the IBM Token-Ring Network.
- IBM Token–Ring Network PC Adapter Guide to Operations, SA27–3710, is the IBM Token-Ring Network operations guide.
- IBM Token-Ring Network PC Adapter Technical Reference Manual, SC30–3383, contains additional reference information for the IBM Token-Ring Network.
- IBM Token-Ring Network Problem Determination Guide, SY27–1280, contains information on problem diagnosis for the IBM Token-Ring Network.
- IBM Option Instructions for the SDLC Communication Adapter, supplied with the IBM SDLC communication adapter, contains adapter installation and connector information.

Contents of Programming Guide

The following is the contents of Advanced Program-to-Program Communications Guide for the IBM Personal Computer Programming Guide, SX27–3757.

Chapter 1, "Introduction to APPC/PC," describes supported features of APPC/PC and how APPC/PC fits in the IBM PC relative to other IBM PC programs. It describes transaction programs and application subsystem programs and introduces other terms used throughout the manual.

Chapter 2, "Developing an Application Subsystem," describes functions that should be considered when designing an application subsystem.

Chapter 3, "Developing a Transaction Program," describes functions that should be considered when designing a transaction program.

Chapter 4, "Introduction to APPC/PC Verbs," briefly describes verb types and the common verb format used in chapters 5 through 9.

Chapter 5, "Using Control Verbs," describes the application program interface (API) to control verbs. First, the control verbs that the application subsystem sends to APPC/PC are described, followed by the control verbs APPC/PC sends to the application subsystem. Near the end of this chapter there are examples of activating and deactivating a node.

Chapter 6, "Using Transaction Mapped Conversation Verbs," describes the application program interface for mapped conversation verbs. Preceding the individual verb descriptions is a discussion of the conversation states that determine which verbs can be issued, and a discussion on understanding mapped conversation return codes.

Chapter 7, "Using Transaction Basic Conversation Verbs," describes the application program interface for basic conversation verbs. Preceding the individual verb descriptions is a discussion of the conversation states that determine which verbs can be issued, and a discussion on understanding basic conversation return codes.

Chapter 8, "Using the Network Management Verb," describes the verb used to provide management services information to a network management services function.

Chapter 9, "Other APPC/PC Services," describes other verbs provided by APPC/PC for the convenience of the programmer. One verb assists with communication between the application subsystem and transaction programs. Other verbs assist with data conversion (ASCII/EBCDIC), tracing facilities, and disabling and reenabling APPC/PC to avoid recursion problems in exit routines.

Chapter 10, "Resolving Error Conditions," describes three types of error conditions with possible solutions and ways to avoid them. The types of errors discussed are return codes, SYSLOG reported errors, and system deadlocks.

Appendix A, "Verb Operation Codes and Formats," lists the operation codes for APPC/PC verbs and the internal formats for the parameter lists passed between the application subsystem, or the transaction program, and APPC/PC.

Appendix B, "Conversation State Matrices," shows the conversation state transitions that can occur when a program issues a conversation verb.

Appendix C, "Verb Return Codes," shows the return codes that APPC/PC can report to a program through the RETURN CODE parameter of each verb.

Appendix D, "SYSLOG Type Codes," lists the SYSLOG type codes that represent error conditions; it includes data errors reported by the transaction program, link errors, configuration errors and system protocol errors.

Appendix E, "Sample Programs," describes the sample programs supplied on the APPC/PC Structures and Sample Programs diskette. There is also a listing for a sample CICS program.

Appendix F, "Sample CICS Host Configuration for APPC/PC," describes sample CICS and VTAM definitions necessary to use APPC/PC.

Appendix G, "APPC/PC Implementation of the LU Tpye 6.2 Architecture,"describes the optional functions of APPC architecture supported by APPC/PC. This appendix also includes a mapping of APPC/PC verbs and parameters with the verbs and parameters used in the APPC architecture documents.

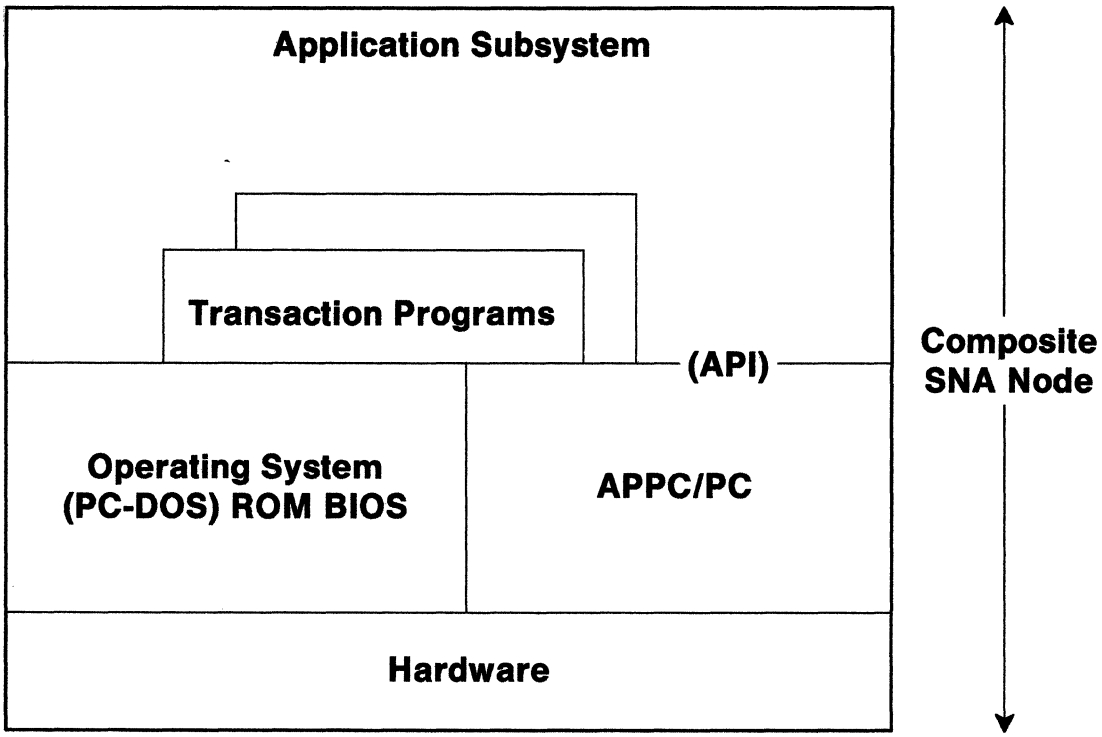
Appendix H, "ASCII/EBCDIC Translation Tables," describes translation tables used by the conversion verb provided by APPC/PC.

Appendix I, "Statement of Service," contains a discussion of IBM service as related to the APPC/PC program product.

A glossary and an index follow the appendixes.

SNA and APPC Terminology

The diagram below illustrates the relationship between the IBM PC hardware and the software components involved in the operation of APPC/PC.



The application subsystem refers to one or more programs whose primary function is to provide services for APPC/PC and transaction programs. The application subsystem logs errors, manages incoming conversations, loads transaction programs, and provides other services. The transaction programs use APPC/PC communication services to communicate with a partner transaction program to perform transactions. The API is the set of commands that the application subsystem and transaction programs use to communicate with APPC/PC. The application subsystem and transaction programs are provided by the user.

A remote transaction program can request a local application subsystem to start a local transaction program so that the programs can exchange data. The corresponding transaction programs are called partner transaction programs.

You can choose the type of conversation that your transaction program uses: a basic conversation or a mapped conversation. The type of conversation you use depends on whether you need full access to the SNA general data stream (GDS) as provided by basic conversations. A header field (LLID) precedes all data that a program sends in the GDS format. The logical length (LL) portion of the header field specifies the overall length of the data and the identification (ID) portion specifies the type of data.

In basic conversations, data passed to and received from the APPC/PC API must contain at least the LL fields of the GDS headers. The transaction program must build and interpret the LL fields but the ID fields are optional. The ID information is necessary only if the partner program expects to receive GDS variables.

In mapped conversations, the data that programs pass to and receive from the APPC/PC API is simply user data. A transaction program using mapped conversations does not require GDS headers to describe the data; therefore, the program

does not have to build or interpret these headers. When the transaction program uses mapped conversations, APPC/PC builds and interprets GDS variables.

APPC/PC verbs fall into three general categories:

- Conversation verbs used by a transaction program to communicate data.
- Control verbs that an application subsystem uses to request services from or provide services to APPC/PC.
- System services verbs for tasks such as network management and ASCII/EBCDIC conversion.

APPC / PC for the Transaction Programmer

APPC/PC provides a transaction program API and an interface to the control functions within the system programmer's application subsystem.

The communication services of APPC/PC extend the services that the operating system normally provides. These services include communication primitives that enable a transaction to use a conversation to communicate with a partner transaction. Each conversation is half-duplex, that is, the transaction program with the right to send data must give up that right before its partner transaction program can send data.

Understanding Locally Initiated and Remotely Initiated Transactions

A transaction can start in one of two ways: by an action initiated at your IBM PC or by an action initiated by a remote transaction program. Initially, the creating transaction program has the right to send data and the created transaction program does not have the right to send data. After initialization, the verb sequences that the programs issue determine the right to send data.

Supporting Multiple Conversations

A transaction program can have conversations with several partners simultaneously. However, a transaction initiated by a remote program is always a new transaction. Therefore, to have more than one conversation with remote programs, a local program must initiate all conversations except the first one.

Each conversation uses a logical resource called a session, and the conversation can use this session for as long as it requires. However, while a conversation is using a session, no other conversation can use it. When a transaction requests APPC/PC to allocate a conversation, APPC/PC responds by establishing a conversation and assigning it to a session.

APPC / PC for the System Programmer

The system programmer provides an application subsystem that uses the defined APPC/PC interface. The primary function of the application subsystem is to provide services for APPC/PC, but the system programmer can also use the application subsystem to provide services for the transaction programs.

The application subsystem manages the services of the communication node not managed by APPC/PC. These services include:

- Defining the logical characteristics of the node, including the Physical Unit (PU), the Logical Units (LUs), the partner LUs, and the desired number of sessions with each partner
- Activating the adapters
- Handling logged error messages
- Providing LU-LU passwords (if security features are required)
- Validating and loading a remotely initiated transaction program (if the capability for remote initiation is required)
- Managing cancellation of a transaction program (optional).

In addition to the application subsystem services, the system programmer can provide other services to the transaction program, as appropriate.

The interface between the application subsystem and APPC/PC consists of two parts:

- A verb interface to establish the PU and the LUs and to define partner LUs, session limits and other communication parameters.
- A set of exit routines to manage incoming transaction requests, log errors and provide LU-LU passwords.

Understanding Initial Application Subsystem to APPC/PC Interactions

The application subsystem must issue verbs to define the capabilities of the communication node. An ATTACH_PU verb establishes the PU and an ATTACH_LU verb establishes each LU. These verbs provide information such as the LU name, processing capabilities, and a method of handling incoming requests for conversations. The application subsystem issues verbs so that the transaction programmer does not need to be concerned with the system definition.

For example, the system programmer may want to provide, as part of the application subsystem, predefined utility routines to execute attachment sequences for particular system configurations.

Managing Incoming Requests for Conversations

The application subsystem must manage requests for conversations (incoming ALLOCATEs) from other transaction programs. The system programmer must decide, for each LU, the best way to manage these incoming ALLOCATEs. Three options are available:

- The application subsystem can reject incoming ALLOCATEs entirely
- The application subsystem can provide an exit procedure that APPC/PC can call when an incoming ALLOCATE arrives (asynchronous option).
- You can direct the LU to queue the incoming ALLOCATEs until the application subsystem requests them (synchronous option)

To process an incoming ALLOCATE, the application subsystem must validate the request, load and/or initiate the requested transaction program, and provide the transaction program with the parameters it must have to issue APPC/PC conversation verbs.

Recommended Tips/Guidelines

- Keep the application subsystem separate from the transaction program to minimize the changes when you add new transaction programs.
- If you are writing transaction programs in a high level language, use a separate routine to interface to APPC/PC. This will help the separation of assembler related instructions.
- If possible, use Mapped Conversation verbs for your transaction program.
- Use 1K or larger records for Token-Ring connections to improve performance. Use 256-byte records for SDLC connections.
- Avoid using Pacing count of zero.
- Be careful with the use of traces in an IBM PC DOS environment. Enough trace options are available, like trace output to storage, display, file or printer. At least, one will be suitable for your environment. Always run trace in a controlled environment.
- Use PASSTHROUGH verb to communicate between the application subsystem and the transaction program.

Sample Application Subsystem - Send Side

The sample program demonstrates how to code an application subsystem. The sample program does not have all the code. It is illustrating some portions of a sample application subsystem. A diskette with a more detailed sample application subsystem comes with the product.

```
;*****************************************************************************
;*                                                                           *
;*          SAMPLE APPLICATION SUBSYSTEM - SEND SIDE                         *
;*                                                                           *
;* Purpose: This program acts as the sample application subsystem for the    *
;*          SEND transaction program.  It demonstrates some of the functions *
;*          to be provided by a full function APPC/PC application subsystem. *
;*                                                                           *
;* Process:                                                                  *
;*          1. Verify that APPC/PC has been installed.                       *
;*          2. Translate ASCII names to EBCDIC                               *
;*          3. Set up Pass-through mechanism                                 *
;*          4. Initialize SNA session with the following sequence of APPC/PC *
;*             verbs:                                                        *
;*             - ATTACH_PU              (activate the Physical Unit)         *
;*             - ATTACH_LU              (activate the Logical  Unit)         *
;*             - ACTIVATE_DLC           (open the adapter)                   *
;*             - CNOS                   (initialize for single session)      *
;*          5. Define portion of memory to be used by sample program.        *
;*          6. Load & Execute sample SEND Transaction Program to issue:      *
;*             - TP_INITIATE(local)     (Application Subsystem unique verb)  *
;*             - ALLOCATE               (attach to remote partner TP)        *
;*               ....                   (conversation verbs)                 *
;*             - DEALLOCATE             (detach session with partner TP)     *
;*             - TP_ENDED               (End the Transaction Program)        *
;*          7. After termination of the sample TP, take down the session with*
;*             the following sequence of APPC/PC verbs:                      *
;*             - CNOS                   (set number of sessions to zero)     *
;*             - DETACH_LU              (take down the LU)                   *
;*             - DETACH_PU              (take down the PU)                   *
;*          8. Reset the Pass-through mechanism.                             *
;*          9. Exit to DOS                                                   *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;* Stack Segment - STACKSEG                                                  *
;*****************************************************************************
;
;* Note: The size of the stack is used in step 5.
stackseg   segment para stack 'stack'
           db      32 dup ("Stack...")
stackseg   ends
           page
;*****************************************************************************
;* Data Structure Definitions                                                *
;*****************************************************************************

;*
;* Include the necessary APPC/PC verb structure definitions.
;*
include    attachpu.str                ; ATTACH_PU    Data Structure
include    attachlu.str                ; ATTACH_LU    Data Structure
include    act_dlc.str                 ; ACTIVATE_DLC Data Structure
include    cnos.str                    ; CNOS         Data Structure
include    convert.str                 ; CONVERT      Data Structure
include    detachlu.str                ; DETACH_LU    Data Structure
;
;* Data area for saving stack pointer across Load & Execute (DOS) Request.
;* Note: The reason for placing it here (rather than in the data segment),
;*       is so that it is addressable off of the CS register.
;*

saved_ss   dw      0                   ; Saved SS;SP
saved_sp   dw      0                   ;

;
;*****************************************************************************
;* DOS entry point.                                                          *
;*****************************************************************************

send_as    proc    far

;*
;* Initialize the stack for a return to DOS.
;* On entry, DS & ES point to the Program Segment Prefix (PSP).
;*
           push    ds                  ; Use the Segment value of PSP
           xor     ax,ax               ; Offset of "Return to DOS"
           push    ax                  ;   request (INT 20) in the PSP
```

```
;*
;* Point DS and ES to the data segment.
;*
          mov     ax,dataseg          ; AX = Segment value of dataseg
          mov     ds,ax               ;   Put it in DS
          mov     bx,offset_env       ;     (save envirmonment address in
          mov     tp_parmblk.env_addr,bx ;   parameter block, see step 6)
          push    es                  ;
          mov     es,ax               ;   Put it in ES too
          pop     prefix              ; Save PSP address (see step 5)
          assume  es:dataseg          ; Inform assembler about ES

;*
;* 1. Verify that APPC/PC is installed.
;*
          push    ds                  ; Save DS (i.e., dataseg)
          xor     ax,ax               ; Clear AX
          mov     ds,ax               ; DS points to low memory
          mov     si,4 * appc_int     ; SI points to APPC/PC interrupt vector
          lds     si,¢si|             ; DS;SI now points to APPC/PC entry point
          sub     si,9                ; Point DS;SI to APPC/PC signature
          mov     di,offset signature ; DI points to expected value
          cld                         ; Clear the direction flag
          mov     cx,7                ; Length of signature
      rep cmpsb                       ; Is the APPC/PC signature present?
          pop     ds                  ; Restore DS
          je      translate           ; Is APPC/PC installed?
          display SNAmissing          ;   No - display message
          jmp     exit                ;      & exit to DOS
                                      ;
;*
;* 2. Translate ASCII names to EBCDIC (e.g., TP_NAMES)
;*
translate: display xlate_msg          ; Inform user of what we are doing
                                      ; First translate all the 8 character fields
          mov     conv_cb.convert_length,8
          mov     word ptr conv_cb.convert_source+2,ds
          mov     word ptr conv_cb.convert_source,offset att_pu_cb.attachpu_netid
          mov     word ptr conv_cb.convert_target+2,ds
          mov     word ptr conv_cb.convert_target,offset att_pu_cb.attachpu_netid
          appc_pc 251,conv_cb         ; Translate PU Net ID to EBCDIC
                                      ;
          mov     word ptr conv_cb.convert_source,offset att_pu_cb.attachpu_puname
          mov     word ptr conv_cb.convert_target,offset att_pu_cb.attachpu_puname
          appc_pc 251,conv_cb         ; Translate PU Name to EBCDIC
                                      ;
```

```
            mov     word ptr conv_cb.convert_source,offset att_lu_cb.attachlu_luname
            mov     word ptr conv_cb.convert_target,offset att_lu_cb.attachlu_luname
            appc_pc 251,conv_cb         ; Translate LU Name to EBCDIC
                                        ;
            mov     word ptr conv_cb.convert_source,offset prt_lu_cb.part_lu_pluname
            mov     word ptr conv_cb.convert_target,offset prt_lu_cb.part_lu_pluname
            appc_pc 251,conv_cb         ; Translate Partner LU Name to EBCDIC
                                        ;
            mov     word ptr conv_cb.convert_source,offset mode_cb.mode_modename
            mov     word ptr conv_cb.convert_target,offset mode_cb.mode_modename
            appc_pc 251,conv_cb         ; Translate Mode Name to EBCDIC
                                        ;
            mov     word ptr conv_cb.convert_source,offset cnos_cb.cnos_pluname
            mov     word ptr conv_cb.convert_target,offset cnos_cb.cnos_pluname
            appc_pc 251,conv_cb         ; Translate Partner LU Name to EBCDIC
                                        ;
            mov     word ptr conv_cb.convert_source,offset cnos_cb.cnos_modename
            mov     word ptr conv_cb.convert_target,offset cnos_cb.cnos_modename
            appc_pc 251,conv_cb         ; Translate Mode Name to EBCDIC
                                        ;
            mov     word ptr conv_cb.convert_source,offset tp_info.luname
            mov     word ptr conv_cb.convert_target,offset tp_info.luname
            appc_pc 251,conv_cb         ; Translate LU Name to EBCDIC
                                        ;
            mov     conv_cb.convert_length,64
            mov     word ptr conv_cb.convert_source,offset tp_info.tpname
            mov     word ptr conv_cb.convert_target,offset tp_info.tpname
            appc_pc 251,conv_cb         ; Translate TP Name to EBCDIC
;*
;* 3. Set up Pass-through mechanism.
;*
            push    ds                  ; Save DS
            push    cs                  ;
            pop     ds                  ;
            mov     dx,offset passthru  ; DS:DX -> Passthru routine
            appc_pc 255                 ; SET_PASSTHROUGH request
            pop     ds                  ; Restore DS

;*
;* 4. Initialize SNA session using APPC/PC verbs
;*
            display att_pu_msg          ;
            appc_pc 1,att_pu_cb         ; 4a. ATTACH_PU
            printrc att_pu_cb.attachpu_rc,RC_4
            or_rc   att_pu_cb.attachpu_rc,RC_4
            cmp     ah,0                ; Zero Return Code?
            je      do_attlu            ;   Yes - Do the ATTACH_LU
            jmp     reset_pt            ;   No  - Terminate the Subsystem
```

Sample Transaction Program - Send Side

The sample program demonstrates how to code a transaction program for a simple conversation. The sample program does not have all the code. It is illustrating some portions of the Send Side of a sample transaction program. A diskette with a more detailed sample transaction program comes with the product.

```
;*****************************************************************************
;*                                                                           *
;*          SAMPLE TRANSACTION PROGRAM - SEND SIDE                           *
;*                                                                           *
;* Purpose: This program is executed by the sample send side application     *
;*          subsystem to demonstrate a simple conversation.                  *
;*                                                                           *
;*    Note: It makes use of a verb implemented in the application subsystem  *
;*          (via the PASSTHROUGH mechanism) called TP_INITIATE.              *
;*                                                                           *
;* Process: (i.e., APPC/PC & Application Subsystem verbs issued)             *
;*          1. Translate ASCII names to EBCDIC                               *
;*          2. TP_INITIATE(local)      (Application Subsystem unique verb)   *
;*          3. ALLOCATE                (attach the remote TP)                *
;*          4. SEND_DATA               (send some data to the other TP)      *
;*          5. DEALLOCATE              (terminate the conversation)          *
;*          6. TP_ENDED                (bring down TP)                       *
;*          7. Exit to DOS             (and Application Subsystem)           *
;*                                                                           *
;*****************************************************************************

;*****************************************************************************
;* Stack Segment - STACKSEG                                                  *
;*****************************************************************************
stackseg   segment para stack 'stack'
           db      32 dup ("Stack...")
stackseg   ends
           page
```

```
;*****************************************************************************
;* Data Structure Definitions                                                *
;*****************************************************************************

;*
;* Include the necessary APPC/PC verb structure definitions.
;*
include    allocate.str                ; ALLOCATE     Data Structure
include    convert.str                 ; CONVERT      Data Structure
include    dealloca.str                ; DEALLOCATE   Data Structure
include    senddata.str                ; SEND_DATA    Data Structure
include    tp_ended.str                ; TP_ENDED     Data Structure
include    tp_init.str                 ; TP_INITIATE  Data Structure

;*
;* Macro used to invoke APPC/PC
;* Note: AX & DX are modified.
;*
appc_pc    macro   request,ctrl_blk
           ifnb    <ctrl_blk>
           mov     dx,offset ctrl_blk  ; DS:DX points to APPC/PC control block
           endif
           mov     ah,request          ; APPC/PC Function Request
           int     appc_int            ; APPC/PC Interrupt Request
           endm

;*
;* Macro to display a message on the screen.
           or      ah,al               ; OR this byte into AH
           loop    or_loop             ;
           ret                         ; AH = OR'd value of all the bytes
orrc       endp

;
;*****************************************************************************
;* DOS entry point.                                                          *
;*****************************************************************************

send_tp    proc    far

;*
;* Initialize the stack for a return to DOS.
;* On entry, DS & ES point to the Program Segment Prefix (PSP).
;*
           push    ds                  ; Use the Segment value of PSP
           xor     ax,ax               ; Offset of "Return to DOS"
           push    ax                  ;   request (INT 20) in the PSP
```

```
;*
;* Point DS and ES to the data segment.
;*
          mov     ax,dataseg          ; AX = Segment value of dataseg
          mov     ds,ax               ; Put it in DS & ES
          mov     es,ax               ;
          assume  es:dataseg          ; Inform assembler about ES

;*
;* 1. Translate ASCII names to EBCDIC (e.g., TP_NAMES)
;*
translate: display xlate_msg          ; Inform user of what we are doing
                                      ;
          mov     conv_buf.convert_length,8 ; Translate all the 8 character names
          mov     word ptr conv_buf.convert_source+2,ds
          mov     word ptr conv_buf.convert_source,offset alloc_buf.allocate_plunam
          mov     word ptr conv_buf.convert_target+2,ds
          mov     word ptr conv_buf.convert_target,offset alloc_buf.allocate_plunam
          appc_pc 251,conv_buf        ; Translate Partner LU name to EBCDIC
                                      ;
          mov     word ptr conv_buf.convert_source,offset alloc_buf.allocate_modnam
          mov     word ptr conv_buf.convert_target,offset alloc_buf.allocate_modnam
          appc_pc 251,conv_buf        ; Translate Mode Name to EBCDIC
                                      ;
          mov     word ptr conv_buf.convert_source,offset tpinit_buf.tp_init_luname
          mov     word ptr conv_buf.convert_target,offset tpinit_buf.tp_init_luname
          appc_pc 251,conv_buf        ; Translate LU Name to EBCDIC
                                      ;
          mov     conv_buf.convert_length,64 ; Now translate the 64 character name
          mov     word ptr conv_buf.convert_source,offset alloc_buf.allocate_tpname
          mov     word ptr conv_buf.convert_target,offset alloc_buf.allocate_tpname
          appc_pc 251,conv_buf        ; Translate TP Name to EBCDIC
                                      ;
          mov     word ptr conv_buf.convert_source,offset tpinit_buf.tp_init_tpname
          mov     word ptr conv_buf.convert_target,offset tpinit_buf.tp_init_tpname
          appc_pc 251,conv_buf        ; Translate TP Name to EBCDIC
                                      ;
;*
;* 2. TP_INITIATE(local)      (Application Subsystem unique verb)
;*
          display inittp_msg          ; Inform user of what we are doing
          appc_pc 7,tpinit_buf        ; TP_INITIATE(local) (Application Subsystem)v
          printrc tpinit_buf.tp_init_rc,RC_4
          or_rc   tpinit_buf.tp_init_rc,RC_4
          cmp     ah,0                ; Zero return code?
          je      do_alloc            ;   Yes - Continue with ALLOCATE
          jmp     exit                ;   No  - Continue no further
                                      ;
```

```
;*
;* 3. ALLOCATE                (attach the remote TP)
;*
do_alloc:  copystr tpinit_buf.tp_init_tpid,alloc_buf.allocate_tpid,8
allo_loop: display alloc_msg           ; Inform user of what we are doing
           appc_pc 2,alloc_buf         ; ALLOCATE the remote TP
           cmp     alloc_buf.allocate_pri_rc,0300h ; Primary RC = 0003?
           jne     alloc_rc            ; No - continue
           push    es                  ; Save ES
           les     dx,alloc_buf.allocate_sec_rc
           mov     ax,es               ; :AX = Secondary Return Code
           pop     es                  ; Restore ES
           or      dx,dx               ; Is secondary RC = 00000005 ?
           jne     alloc_rc            ;   No
           cmp     ax,0500h            ;   Maybe
           jne     alloc_rc            ;   Yes - Retry the ALLOCATE?
           display alloc_err           ;
retry_chk: mov     ax,0C07h            ;       - Ask the user
           int     dos_int             ;         (clear keyboard & wait for key)
           cmp     al,CR               ;
           je      allo_loop           ;         Retry requested
           cmp     al,ESC              ;
           jne     err_retry           ;
           jmp     end_tp              ;         Exit requested
err_retry: display retry_err           ;
           display retry_msg           ;
           jmp     short retry_chk     ;
alloc_rc:  printrc alloc_buf.allocate_pri_rc,RC_6
           or_rc   alloc_buf.allocate_pri_rc,RC_6
           cmp     ah,0                ; Zero return code?
           jne     end_tp              ;   No - End the TP
                                       ;
;*
;* 4. SEND_DATA               (send some data to the remote TP)
;*
           copystr tpinit_buf.tp_init_tpid,send_buf.senddata_tpid,8
           copystr alloc_buf.allocate_conv_id,send_buf.senddata_conv_id,4
           display send_msg            ; Inform user of what we are doing
           display data_msg+2          ; Display the message being transmitted
           appc_pc 2,send_buf          ; Send the message
           printrc send_buf.senddata_pri_rc,RC_6
                                       ;
```

```
;*
;* 5. DEALLOCATE              (terminate the conversation)
;*
          copystr tpinit_buf.tp_init_tpid,deall_buf.dealloca_tpid,8
          copystr alloc_buf.allocate_conv_id,deall_buf.dealloca_conv_id,4
          display deallocmsg          ; Inform user of what we are doing
          appc_pc 2,deall_buf         ; Deallocate the conversation
          printrc deall_buf.dealloca_pri_rc,RC_6
;*
;* 6. TP_ENDED                (bring down TP)
;*
end_tp:   copystr tpinit_buf.tp_init_tpid,tpend_buf.tp_ended_tpid,8
          display tp_end_msg          ; Inform user of what we are doing
          appc_pc 4,tpend_buf         ; TP_ENDED verb
          printrc tpend_buf.tp_ended_rc,RC_4
                                      ;
;*
;* 7. Exit to DOS             (and Application Subsystem)
;*
                                      ;
exit:     ret                         ; See notes at program entry point

send_tp   endp
codeseg   ends

          end     send_tp
```

Revisable-Form Text Document Content Architecture

Introduction

The Revisable-Form Text Document Content Architecture (RFT DCA) specifies how IBM office systems encode office documents for intersystem document interchange. More specifically, it describes how a document in a revisable state can be uniformly interpreted and understood by any office system supporting the architecture. Both functional richness and the interchange of this function are goals of the architecture but, of these two goals, functional interchange is more important.

A straight-forward method is used to achieve consistency of function and the interchange of this function. The method provides a single specification of the revisable document in transit in the network. This single specification gives detailed and explicit descriptions of all elements of the document in its revisable state. It separates the document layout from its content and allows for imbedding of format and text manipulation controls within the user's text. The specification must make as few implications as possible about the receiver's use of the document. The semantic detail of the data stream is equally as important as the precision of the syntax used in encoding the data stream.

Several IBM products, including those in the DisplayWrite family use the RFT DCA. It is also supported by a number of non-IBM products. The RFT DCA is described in two publications. The *Office Information Architectures*: Concepts (GC23-0765) contains an overview of the RFT DCA while the *Document Content Architecture*: Revisable-Form Text Reference (SC23-0758) is a guide to the details of the architecture.

Semantics and Syntax of the Architecture

The RFT DCA is a set of text elements, a set of text statements, and descriptions of the output of processing the text statements. An RFT DCA document is actually a "snapshot" of a text document at some stage in its overall creation or revision. It is in this context that the RFT Document is to be interpreted.

Text Elements and Operations

An understanding of the semantics of the Revisable-Form Text Document Content Architecture requires first a familiarity with all those parts of a text document that are utilized or identified when creating and modifying the document, such as characters, words, lines, pages, paragraphs, headers, margins, footnotes and indices, to name a few. These parts, or **text elements**, are precisely defined in order to avoid ambiguities when they are used. In a text document, the "character" is the basic element of text. Characters are then grouped into "words" which, in turn, are grouped into "lines." "Pages" are formed by grouping lines together, and so the definition process goes, until all text elements are defined.

The next step is to understand the actions, or **text operations**, that can be carried out on the text elements for document creation and change. A sampling of some of the text operations includes underscore, overstrike, keep, insert, move, adjust, align, release, establish, locate, begin, erase and print.

Text Statements

Text statements are directives, or commands, formed by associating a specific text operation with selected text elements. For instance, the text element "word" could be associated with the text operator "underscore" to get the statement "underscore this word". However, the association of "underscore" with "leading margin" to get "underscore the leading margin" is of little use, with current document editors. It is from the set of relevant combinations of text elements and operations - referred to here as *text statements* - that the semantics, or meaning, of the RFT DCA evolves.

Text Processes

Text processes are defined to carry out the directives of text statements. Simple text processes are those that support a single text statement. These can be identified, indirectly, from the descriptions of the controls defined in the architecture, such as BEGIN OVERSTRIKE, RETURN TO MASTER FONT, END KEEP or PRINT PAGE IMAGE NUMBER. Complex text processes use several processes (simple and complex) to carry out their task. The input to these processes is more

extensive and in some cases involves elements outside the document. The architecture also describes the output of some of these complex processes. Text editor functions, such as Line Adjust, Page Imaging and Pagination, are examples of complex text processes. The description of the output from processors of text statements is the basis for understanding the RFT DCA semantics. The clarity and precision of this description determines the effectiveness of the architecture for functional interchange.

Syntax of the Revisable-Form Text Document

The *syntactical* description of the RFT DCA. determines how the document is encoded so that the text elements and statements can be identified and related for input to the text processes.

Major Organizational Sections

The document, in its encoded form, is referred to as a data stream. The document must be presented for processing in a definite sequential order with its parts properly identified and the correct relationship of elements and statements maintained. The complete RFT DCA document consists of three major sections: the Format Units, the Text Units and the End Unit. Relative to the other two sections, the End Unit is trivial. Its sole function is to identify the end of the document in its encoded form. As a minimum, an RFT DCA document requires at least two Format Units, one Text Unit and the End Unit. Format Units always appear first, followed by Text Units, with the End Unit always the last unit in the document.

Format Units

The RFT DCA document must contain at least two Format Units (Format Unit 1 and Format Unit 2); third (Format Unit 3) is optional. Format Unit 1 will always be the first unit is the RFT DCA document; Format Unit 2 will always be the next unit in the document. If Format Unit 3 is present, it must follow Format Unit 2 before any Text Units are specified.

Format Units consist of several sets of statements used to define the initial values of all parameters to be used by the text processes. They are primarily used to establish formatting states, but they may identify theresources to be used when processing the text. Format Unit 1 contains the Document Parameters (a required set) and other optionally specified parameter sets, such as Character Replacement values and Punctuation Formats for data inserts.

Format Unit 2 and Format Unit 3 are used to establish all the formatting values needed for the document. Format Unit 2 (containing the Primary Master Format) is required. Format Unit 3 (containing the Alternate Master Format) may be defined optionally. Each of these contain sets of statements that establish the intial values needed when the text is presented in its intended format or layout. Each Master Format contains sets of statements to establish the Page Image, Lines, Tabs, Page Numbering, Line Numbering, Notation Formats, Table of Contents, Index, Print Medium or Operator Messages. Following the Master Format in the Format Unit is an optional Margin Text Declaration that is used to establish Margin Text layout and positioning, as well as to define the text content for the headers and footers.

Text Units

As stated earlier, every RFT DCA document must contain at least one Text Unit. The Text Unit, or units, must always follow all defined Format Units. The Text Unit represents the text document element called a "page," which may not necessarily match a page on a printer. It is on Text Unit boundaries that such page format elements as headers, footers and page numbers are applied, as is the resetting of several counters associated with such elements on the page as line numbers and footnote references, if requested by the user.

The Text Unit consists of two major divisions: an optional Format Change section at the beginning, and the required Body Text section. The Format Change section allows the user to change the active format optionally. It allows either the Primary or Alternate Master to be established as the active format, returning to a previously-defined Master or just changing the content of the current format. If format changes are specified, they become the active format values for this text unit and any subsequent text units until again changed by a later Format Change specification in a Text Unit.

The Body Text section consists of one or more Body Text Units. The Body Text Units carry the actual text of the document. Up to this point, all previous units contained definitions, range values, initial settings, identifiers and other statements, all of which described the format, or layout, to be used when the text is presented for display or print. Because several formatting, or presentation, states apply to document elements within the page, a method is needed to identify when to perform presentation changes to these elements without relying on page boundaries to make changes. This is the function of the RFT DCA controls. Controls may be included in the user's text to make changes

on character, word or line boundaries, such as changing typestyle for selected words or phrases or keeping specified lines together on the same page.

Constructual Elements

The RFT DCA uses several methods to encode the data stream. The primary method is **structured fields.** The entire encoded document is contained in consecutive structured fields. Several structured fields may be grouped together into **structures.** To accomodate the encoding of text statements within these structured fields, **controls** are used for those text statements that occur within the text of the document, such as margin text or body text. If the structured field or control must contain several pieces of information, this information is encoded using either **self-identifying parameters** or **positional parameters.**

Each format unit that is defined consists of a Format Unit Prefix (FUP) immediately followed by the format unit itself. FUPs are encoded as separate structured fields. Format Unit 1 consists of a structured field for the document parameters, a structured field for the character replacement table (if it is used), and 0-99 structured fields for any defined punctuation formats. Each of these structured fields consists of one or more self-identifying or positional parameters, except for the FUP which has no parameters with it. There are no structures or controls in any of this initial encoding of the RFT DCA document.

Constructual Elements of the Master Format

The encoding of Format Units 2 and 3 is more complicated than Format Unit 1. Structures are employed since the information they carry is more extensive. As in Format Unit 1, each unit is first preceded by a FUP structured field. This is followed by the Primary Master Format structure in Format Unit 2 or the Alternate Master Format Structure in Format Unit 3, which is then followed by the 1-6 structured fields that could be used to provide the Margin Text Declaration. Since the allowable content of Format Units 2 and 3 are both the same, a description of one also covers the other.

Format Unit 2 consists of a FUP, followed by the Primary Master Format (PMF) structure which, in turn, may be followed by the Margin Text Declaration structured fields. The PMF structure is composed of three required structured fields, which must be the first three in the structure in a specified order, and up to seven additional, optional structured fields. The three required structured fields are Page Image Parameters, Line Parameters and Tab Parameters, and they must appear in this order in the structure. Each of these structured fields contain several positional parameters. The optional structured fields may be in any order. The optional structured fields are Tab Parameters (Right-to-left), Line Numbering, Page Image Numbering, Page Format Parameters, Print Medium, Operator Message, Auto-Outline Parameters and Note Format Parameters. The parameters in each of these are self-identifying.

Margin Text Declarations consist of two to six structured fields each; the actual number depends on how the margin text is to be defined. If the same text is to appear only on the top or on bottom of each page, then one structured field for the Margin Text Parameters and one for the Margin Text itself is all that is needed. If different text appears on the top and bottom and on odd and even pages, then all six structured fields are required.

Constructual Elements of the Text Unit

Each Text Unit defined in the document is preceded by a Text Unit Prefix (TUP) structured field. This prefix introduces the Text Unit and contains a unique name for the unit. The name is formed from numeric characters, and each Text Uunit in the document must be in proper order, that is, each succeeding Text Unit has a higher 'name' than the Text unit immediately preceding it.

The TUP is followed by any optional Change Format Declaration structures or structured fields that may be defined. The structured fields that may be present are Establish Primary Master, Establish Alternate Master or Return To Master Format. These are simple structured fields with no parameters. The structure that may be defined is the Text Unit Format Change. Its content is the same as previously described for Format Unit 2 following the FUP. Its purpose is to keep the same previously established master as the active format, but to change selected parameters in the format, including the margin text declaration.

The rest of the Text Unit consists of one more Body Text Units. Each of these units is a structured field. The only requirement for the content of the Body Text Unit is that the last, or only, Body Text unit must contain a Page End control (a single byte control) as the last byte in the last Body Text Unit.

Encoding Structured Fields and Parameters

Structured fields and structures always begin with a 5-byte field represented by the letters LLCTF (a letter for each byte of this "introducer"). The LL is a 2-byte length value that gives the entire length of this structured field. This length value includes the 2 bytes occupied by the length field itself. The C byte is a CLASS value, the T byte is a TYPE value

and the F byte is a FORMAT value. These three bytes together are used to determine which structured field it is. All the structured fields defined by the RFT DCA have unique CTF values. Structures, as stated, consist of one or more structured fields. The introducer for a structure is encoded in the same format as that for a structured field and is also assigned a unique CTF value to differentiate it from other structures or structured fields. The LL value in the structure introducer would be for all the elements in the structure, including any structured fields it contains.

Positional parameters are the simplest method of encoding parametric values for structured fields or multibyte controls. They are identified by their position in relation to the start of the containing construct or the preceding positional parameter. Self-defining parameters (SIPs), may occur in any position within the containing construct. Each SIP contains a 2-byte introducer which consists of a 1-byte length field and a 1-byte TYPE identifier. The length value includes the entire length of the parameter, including the the 2-byte introducer. Since the length value is restricted to 1 byte, self-identifying parameters are restricted to a maximum length of 255 bytes.

Encoding Controls

Controls are text statements that occur in-line with the user's text. Controls may appear in body text, margin text, footnote text and the formatted text for outline entries, footnote references, index and table of contents entries, are anywhere the text of the document might appear while the document is still an RFT DCA document.

Single byte controls are code points reserved for control purposes rather than characters. In any of the code pages used by the architecture, the range of code values from X'00' through X'40' is considered the control code range. Some examples of single byte controls are BS (backspace), CRE (an adjustable carrier return), HT (horizontal tab) and INX (index). Each is assigned a code point that is the same for all code pages. The Page End (PE) control that is required as the last byte in the last Body Text Unit in a Text Unit is a single byte control. The single byte value of X'2B' is the encoded value in the data stream that identifies the start of a multibyte control.

Multibyte controls always start with a X'2B' value. This is always followed by three more bytes called the CLT bytes, where the C represents the CLASS byte, L represents a 1-byte length value of the control and T is the TYPE byte. For each multibyte control defined in the RFT DCA, there is a unique C and T combination assigned to it. The length value is the count of bytes following the C byte that are part of this multibyte control. This allows for encoding the positional or self-defining parameters, that are associated with this particular control. For example, if the multibyte control BEGIN OVERSTRIKE (BOS) was encoded in the data stream, it would be identified by a X'2B' followed immediately by a X'D40872'. The D4 and 72 together identify this as the BOS control. The 08 indicates there are 6 bytes of parameters that follow the T byte (the LT bytes are included in the the length value). The BOS Control has three required positional parameters - 4 bytes for the GCID, 1 byte for the overstrike character and 1 byte for bypass options. Assume that the remaining 6 bytes of the control are encoded as X'006E01006100'. Therefore, the Character Set and Code Page (GCID) to use are Character Set 110 and Code Page 256. The character to use for overstriking is the "/" (code point X'61' on code page 256). The final X'00' indicates nothing is to be bypassed while overstriking. Therefore, when the text that follows this control is presented for viewing (if it is WYSIWYG) or formatted for print, all characters and most character positions on the line will be overstruck by a slash until an END OVERSTRIKE control is processed. Other details that should be known before this control is actually implemented are discussed in the Reference Guide, (SC23-0758).

Supporting the RFT DCA Document

Several IBM products and a number of non-IBM products, support the RFT DCA document. Each has different total functional capabilities, different methods of interfacing to their users and different hardware requirements to support their functions. How, then, will all of these apparently dissimilar products operate in the same office network and interchange their documents without tailoring the transmission to fit the capabilities of receivers?

Interchanging RFT DCA Documents - The Product Transform

In a network, a revisable-form text document can carry text documents for revision purposes as well as documents in final-form. For presentation purposes. The RFT DCA has evolved to provide this single document type for text documents. All office products interchanging documents should provide a document transform as part of the product. The transform is the product interface to the network, supporting only the RFT DCA-defined document on the network side and the product-specific or internal form of the document on the product side. Now the product is loosely coupled to the architecture, and the transform acts as an effective buffer between the product and its implementation

and the RFT DCA and its definitions. Each product now 'talks' to a single entity, the RFT DCA, for functional capabilities in the interchange realm.

The next matter to resolve is how to optimize the actual function that each product can interchange with its partners in the office system network. The solution is simple to state, but it is probably the most difficult to achieve. First, all the products must use the RFT DCA document for interchange. Any function a product wishes to interchange has supporting text elements and text statements defined in the architecture so that each product has the necessary encoding values to support its function. Once the architecture is in place, the function can be sent and received in the RFT DCA documents it interchanges. It is clear, however, that for the function to be supported by other products in the system, they must add that function to their support. History tells us that products change in their own way and in their own time. Functionally enhancing an office system of several different products can be a demanding and frustrating task; however, it is not impossible. With the RFT DCA as the single focal point for setting functional content for interchange, uniform product support can be achieved.

Round-Tripping RFT DCA Documents

When a product of greater functional capability transforms and sends its document on the network and to be received and transformed by a product of lesser function, a problem can occur. What is the receiving product to do with those RFT DCA elements it does not support or even recognize?

The RFT DCA has a standing rule for all receivers of an RFT DCA document. Simply stated, the receiver is not to quit but is to make every effort to preserve the unrecognized or unsupported piece of the RFT DCA document. The product should make known to the user that an exception to normal processing has occurred, and to proceed under the assumption that some product in the network understands and supports this part of the RFT DCA document. Granted, each product will be limited as to how much it can preserve. It is also understood that while processing the document this information may be edited out or dropped for presentation. However, if the document is later sent back out on the network, the preserved pieces must flow back into the RFT DCA document in the original form (as preserved) and in the same relative position in the data stream.

Product-Provided Information

In addition to preserving elements, the architecture states that the product will make a "best can-do" substitution of values it does support and note that these values were added and are not the original values.

For example, all products are expected to fully support Code Page 256. However, some products use other code pages as well as CP256. If a product receives an RFT DCA document in which code pages other than 256 are used, that product is expected to preserve the original code page value and substitute as best it can for its form of the document. Since this actually is an exception action, it is the products way of continuing processing after the exception condition is detected (the unsupported code page). Similar situations exist with range values on line lengths, page dimensions, amount of margin text, dictionary IDs, page number numerals, print date formats and many other elements that offer ranges for parameter values or have several optional parameters values that may be used.

Product Implementation Responsibilities

The following is a list of the major areas a product needs to address when implementing support for the RFT DCA document:

1. Strive to maximize functional capability based on the function offered by other office products in the office network that would be your document interchange partners.

2. Provide exception flagging and exception action to allow processing of the document to continue.

3. Preserve, wherever possible, those RFT DCA elements you do not support or do not recognize. Return the preserved elements to the RFT DCA form of the document when it is interchanged.

4. Use substitution of values as an exception action in conjunction with preserving elements for processing the product form of the RFT DCA document.

5. Conform to the architecture for all encodings of the RFT DCA data stream.

Summary

The revisable-Form Text Document Content Architecture is both a semantic description of a document in a revisable form and a syntactic definition of this document in its data stream form. Office editors and word processors that utilize this structure for document interchange and conform to the support requirements when receiving and sending documents in the office network will achieve maximum functional interchange among otherwise independent and dissimilar office products.

Document Interchange Architecture (A Solution for Office Information Interchange)

Introduction

Office systems may differ in several ways–each offers different capabilities and answers the needs of different users. The thread that ties these systems together is information interchange. The goal is to allow dissimilar office systems to communicate with one another in an understandable manner.

To achieve this goal one requirement is a comprehensive and standardized method for conveying both the information and the intended use of this information in an office systems network. This is the purpose of IBM's Document Interchange Architecture (DIA).

DIA is a process–to–process communications architecture that defines how information and requests for processing functions are communicated in an office systems network. Essentially, DIA specifies the rules and data structure that establish the discipline for unambiguous interchange of information and processing requests between office systems.

Document Interchange Architecture can be viewed as consisting of:

- A set of logical components
- A set of processing services
- A set of protocols
- A data stream structure

Together these four areas form the foundation for information interchange.

DIA Logical Components

A network of office systems based on DIA is a set of interrelated logical components that lie within the framework of the physical components of a network. The logical components are defined by DIA and are implemented as processes executing in the physical components of the communications network. The nodes defined by DIA are:

- Source Node

A source node provides one or more users with DIA services, that initiate and control the interchange of information with recipients.

- Recipient Node

A recipient node provides one or more recipients with DIA services that control and receive information sent by source nodes.

- Office System Node (OSN)

An office system node provides DIA services that receive, store, route and deliver information from source nodes to recipient nodes. An office node also can interact with an appropriately configured network to distribute information to other office system nodes.

Source nodes, recipient nodes and office system nodes interchange information using the common transport services of the network. The logical components of a DIA office systems network are shown in Figure 1.

All nodes must be uniquely identified within the network. Specifically, a source node is identified by a source address and a recipient node by a recipient address. An office system node is identified by either an originating node address, when the OSN is supporting a source node, or a destination node address when, the OSN is supporting a recipient node. The nodes provide the services for the control and information exchange within the network.

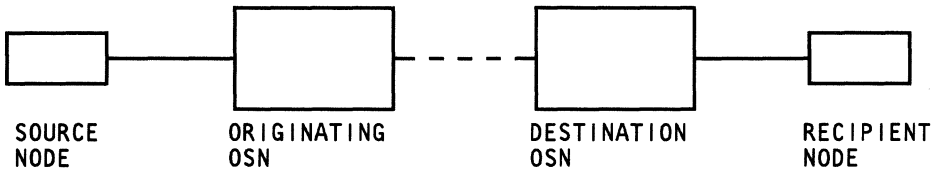


Figure 1. Logical Components of a DIA Network

DIA Services

DIA consists of a set of defined services performed by peer communication processes in the network nodes. Each DIA service carries out specific functions requested by the users. Users, in this context, can be application programs, devices or a person and, they represent the source or receiver of information flowing through the network.

The services defined by DIA are analogous to the functions performed in today's office. These services can be categorized into the following general areas:

- DIA Session Services
- Document Library Services
- Document Distribution Services
- Application Processing Services

DIA Session Services

DIA processes use DIA session services to establish a logical connection, called a DIA session, through which they may exchange information. A DIA session is established by a dialog of DIA commands between two nodes in the network. A command represents a functional unit of work to be performed by the receiving DIA process. For DIA sessions services, this set of commands is shown in Figure 2.

A DIA session exists after two DIA processes identify themselves and agree on the scope of work that is to be performed. While DIA defines a wide range of office system function, most office systems require only a subset of these functions for their operations. Therefore, the DIA session agreement mechanism provides the vehicle for defining the proper scope of function or work for each DIA session.

DIA commands are grouped into function sets that identify the scope of work for a DIA session. These function sets have been defined so that each set contains the commands required for a well defined, usable and complete set of functions for a given category of services.

COMMAND	COMMAND DESCRIPTION
SIGN–ON	Establishes a DIA session between two DIA processes.
SIGN–OFF	Terminates a DIA session.
ACKNOWLEDGE	A general replying command informing a DIA process of successful or unsuccessful completion of a request.

Figure 2. DIA Session Services Commands

Document Library Services

Document library services are used for storing and retrieving information electronically. These functions are analogous to manually filing and retrieving of paper documents in an office.

However, document library services can also perform activities that are cumbersome in a manual system. For example, when a document is electronically filed in a document library (electronic filing cabinet) a set of descriptors called a document profile is filed with it. The profile contains parameters that identify the contents of the document, such as the name under which it is filed, the authors, the subject, the date it was filed and other information pertinent to the management of the document in the library.

These document profiles can be used in searching for documents in a library. As an example, a user can ask the office system to search for all documents about a particular subject and by a certain author recieved by the library between any two dates. On completing the search, the office system produces a list of the documents that meet the user's search criteria. The user can then ask the office system to retrieve a specific document on the list from the library and deliver it to the user for printing or viewing. The set of commands provided by DIA library services are shown in Figure 3.

COMMAND	COMMAND DESCRIPTION
FILE	Files the identified document and associated document profile parameters in the document library.
RETRIEVE	Returns a copy of the identified library document to an authorized document requester. This command can also be used to request delivery of descriptors of documents found by a prior search command.
DELIVER	Sends a library copy of a document to the requestor as a result of a retrieve request.
SEARCH	Locates the documents in the library that match search criteria specified by the requestor of the search. Return of descriptors of the documents found can be requested by this command.
DELETE	Removes access to the identified document for the delete requestor. When all owners of the document have been deleted, the document is removed from the library.

Figure 3. DIA Library Services Commands

Document Distribution Services

Document distribution services send information such as messages or documents from one user to others in an office systems network. Documents and messages can be distributed between source and recipient nodes through office system nodes for later delivery to each recipient (deferred delivery) or sent directly from a source node to a recipient node (immediate delivery).

When documents or messages are delivered through an office system node, document distribution services in the source node do not establish a direct DIA session with document distribution services in the recipient node. Instead, a DIA session is established between the source node and the originating office system node. The originating office system node then queues the distribution request for later delivery to the specified recipients. If the recipient node is located on a different office system node, the information is sent through the network to the proper destination office system node, where the distribution is enqueued for later delivery. This deferred delivery alleviates the

problem of having all of the network's physical components available to the sender of a distribution request.

When the recipient node establishes a DIA session with its office system node, it can obtain a summary list of documents and messages. It also can receive any or all the documents or messages, or it can cancel delivery of any or all the information queued for the recipient at the destination office system node.

This concept is analogous to a postal system where the destination of information can be viewed as the flow of mail in the system and the OSN recipient queues as mail boxes. The set of commands shown in Figure 4 provides the functions for this distribution system view.

Additionally, the sender of a document or message can specify a distribution priority for it relative to other distribution requests. That is, senders can request information to be delivered to recipients faster. The sender also can request notification of delivery of a document or message to its recipients. This notification is called a confirmation-of-delivery message.

Document distribution services allow users to send a document or message to a distribution list defined in an office system node. The office system node will queue a copy of the document or message to each recipient defined on the distribution list. Each recipient can then take delivery of an individual copy of the document or message.

COMMAND	COMMAND DESCRIPTION
REQUEST-DISTRIBUTION	Sends document and/or messages from a source node to an office system node for delivery to specified recipient nodes.
OBTAIN	Requests delivery of one or more documents and/or messages scheduled for delivery to a recipient node.
DELIVER	Sends a document and/or message from an office system node to a source or recipient node.
LIST	Provides a list of documents and messages queued for delivery on an office system node for the requesting recipient node.
STATUS-LIST	Notifies the recipient node that one or more documents and/or messages are available.
CANCEL-DISTRIBUTION	Cancels distribution status information or cancels the delivery of distributed documents and/or messages.

Figure 4. DIA Document Distribution Service Commands

Application Processing Services

Application processing services define commands that request an office system node to perform several additional functions. These additional functions allow users to manipulate document profiles associated with a document (for example, to add or delete keyword descriptors used for searching the library), to invoke a program to process documents, and to invoke user application programs, procedures or processes. A summary of the functions in DIA application processing services is shown in Figure 5.

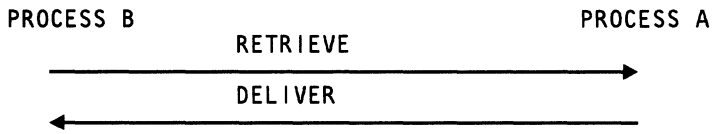
COMMAND	COMMAND DESCRIPTION
EXECUTE	Requests an office system node to schedule the named process for execution.
FORMAT	Requests an office system node to invoke a specified formatting process using the identified document as the source.
DELIVER	Sends the document produced as a result of the format operation to the requesting recipient node.
MODIFY	Requests an office system node to revise document control information fields.

Figure 5. DIA Application Processing Services Commands

DIA Protocols

Information exchanged between DIA processes consists of commands and user information. To achieve this information exchange, DIA defines a request/reply command protocol. To illustrate the DIA request/reply command protocol, consider the following scenario: Process B sends a request to Process A to retrieve a document from Process A's document library; Process A interprets the request, retrieves the document from the library, and delivers the document to the requestor (Process B).

DIA defines Process B as the requestor of a service and Process A as the server of the request. These are the logical roles performed by DIA processes.



The above scenario illustrates the basic DIA request/reply protocol. The RETRIEVE command is the function request and the DELIVER command is the reply to the function request.

This scenario also illustrates that the server (Process A) responds on demand to the requestor (Process B). The on demand request/reply protocol is one of the command classes defined by DIA to perform a unit of work desired by the requestor at the server. The on demand command class is called the Synchronous Reply Required (SRR) command class, that is, the command execution and the command reply is processed synchronously between the requestor and the server. Other command classes in DIA are the No Reply Required (NRR) and the Asynchronous Reply Required (ARR) command classes. The NRR command class is used by the requestor when the function does not require a replying command by the server. The ARR command class is used by the requestor when the function requested does not need to be performed synchronously but can be deferred for later processing by the server.

DIA Data Stream Structure

DIA defines a data stream structure called a Document Interchange Unit (DIU) to exchange requests and replies between a requestor and a server. Figure 6 illustrates the major DIU components.

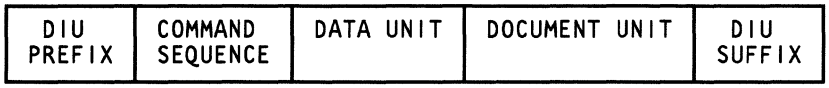


Figure 6. DIA Document Interchange Unit

The DIU consists of the following components:

- The *DIU prefix* introduces and identifies the information that follows in the data stream as a Document Interchange Unit.
- The *command sequence* contains the DIA command that specifies the function to be performed.
- The *data unit* contains command operand information. This component is optional and is present when defined by the command.
- The *document unit* contains the document profile and the document content. This component is optional and is present only when a document is sent from one DIA process to another.
- The *DIU suffix* specifies the end of the DIU and indicates the normal or abnormal completion of the DIU request.

These data stream components are further composed of substructures called subcomponents. Examples of subcomponents are command operands and document profiles. All DIU components and their associated subcomponents begin with a structured field called an introducer. The introducer uniquely identifies the components and subcomponents and defines its length. The introducer provides for an extendable, self describing, variable length data stream.

Summary

Document Interchange Architecture supports a logical view of an office systems network that enables users to interchange information in the network. Specifically, this architecture provides:

- Document library services

These services let users file, retrieve and delete documents and other information from a document library and to search the library for documents that meet user-specified criteria. Document library services provide the ability to organize, manage, and control information assets.

- Document distribution services

These services let users electronically distribute correspondence, reports, contracts, proposals and other information in an office systems network. Document distribution services provide timely, cost effective and efficient dissemination of information among users.

- Application processing services

These services let users change documents, change the search descriptors of stored documents and invoke user-written programs to accomplish specific functions. Applicatin processing services provide for the flexible placement of office system applications in the network.

References

- *Office Information Architectures: Concepts*, GC23-0765, IBM Corporation, March 1983.
- *Document Interchange Architecture: Technical Reference*, SC23-0781, IBM Corporation, May 1985.
- *Document Interchange Architecture: Interchange Document Profile Reference*, SC23-0764, IBM Corporation, May 1985.
- *Document Interchange Architecture: Transaction Programmers Guide*, SC23-0763, IBM Corporation, May 1985.

Introduction to IBM Enhanced Connectivity Facilities

IBM Enhanced Connectivity Facilities are a set of programs for interconnecting IBM Personal Computers and IBM System/370 host computers operating with the MVS/XA or VM/SP environment. The goal of IBM Enhanced Connectivity is to provide a consolidated approach to PC-to-host communication. IBM Enhanced Connectivity Facilities provide a single interface that allows applications programmers to write personal computer and host applications that run on a variety of communications connections. The interface is designed to shield an application from the underlying systems' communication environment.

IBM Enhanced Connectivity Facilities Characteristics

Figure 1 on page 38 identifies the characteristics of the IBM Enhanced Connectivity Facilities described below:

- A consistent interface for application programs in a personal computer to request services, data or both from a host. The requesting program is referred to as the **requester.**

- A consistent interface for programs in a host to reply to requests for services, data, or both from personal computers. The program that services the request is referred to as the **server.**

- A consistent interface for handling communications between requesters and servers. The program, provided in both personal computers and hosts is referred to as the **router.** The routers provide a new **Server-Requester Programming Interface (SRPI)** : a request interface for requesters, and a reply interface for servers. This interface isolates requesters and servers from the underlying communications environment.

  Writing to the SRPI enables applications to be used in a number of communication environments. The SRPI provides a consistent interface for programs in personal computers to obtain services, data or both from different hosts.

Requester applications on the PC use the SRPI by issuing the Send_Request function. Invocation of the Send_Request function is analogous to a main routine "calling" a subroutine. The host server assumes the role of a subroutine and "returns" data to the requester. The Send_Reply function is used by the host server to "return" the data to the PC requester. The details associated with reliably transmitting the requester data to the host server are handled by the PC Router and the Host Router. The same is true for host server data returned to the PC requester.

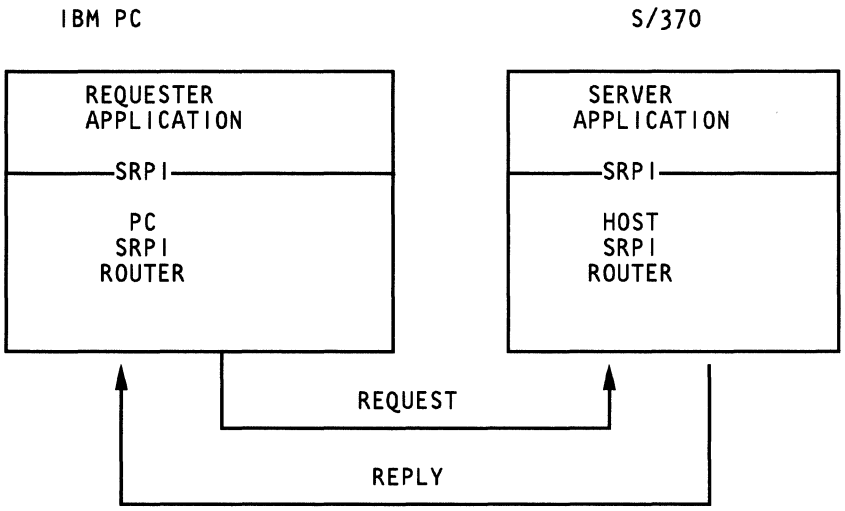


Figure 1. **SRPI Flow Overview**

Advantages of IBM Enhanced Connectivity Facilities

IBM Enhanced Connectivity Facilities:

- Provides a consolidated host attachment solution for a variety of physical connections supported by:
  - IBM PC 3270 Emulation Program, Version 3.00
  - IBM 3270 Control Program, Version 3.00
- Shields applications from communications layers
- Simplifies distribution of personal computer and host data to personal computer applications
- Simplifies access to customer-written personal computer software
- Improves programmer productivity, when the SRPI is used to develop distributed applications
- Reduces communications expertise required in developing distributed applications
- Simplifies access to host facilities by personal computer applications.

What You Need

Program Products

IBM offers a group of Enhanced Connectivity program products for selected IBM systems. The following IBM PC program products provide the SRPI function required by PC requesters:

- IBM PC 3270 Emulation Program, Version 3.00 (59X9969) (DOS 3.1 or 3.2 prerequisite)

  Hardware Supported
  - PC
  - PC XT
  - PC AT
- IBM 3270 PC Control Program, Version 3.00 (58X9968) (DOS 3.1 or 3.2 prerequisite)

  Hardware Supported
  - 3270 PC (except 5271 models 24 and 26)
  - 3270 PC AT (all models)

On the host, the SRPI function is provided for servers by the following IBM program products:

- TSO Extensions (TSO/E), Release 3, with the MVS/XA feature includes the the IBM Enhanced Connectivity support (5665-285)

- VM/System Product, Release 4 includes the IBM Enhanced Connectivity support (5664-167)

Related Publication

The following book describes the functions of the Enhanced Connectivity Facilities.

- Introduction to IBM System/370 to IBM Personal Computer Enhanced Connectivity Facilities (GC23-0957)

The following book explains how to write PC requesters using the SRPI.

- IBM Programmer's Guide for the Server-Requester Programming Interface for the IBM Personal Computer and the IBM 3270-PC (SC23-0959)

The following book explains how to write and install servers using MVSSERV, the host router for MVS. It is intended for application designers and programmers who design and write servers and server initialization/termination programs; and system programmers who install MVS/XA systems.

- IBM TSO Extensions Programmer's Guide to the Server-Requester Programming Interface for MVS Extended Architecture (SC28-1309)

The following book explains how to write and install servers in a VM/SP system. It is intended for application designers and programmers who design and write servers and server initialization/termination programs; and system programmers who install VM/SP systems.

- IBM Programmer's Guide to the Server-Requester Programming Interface for VM/System Products (SC24-5291)

Send/Request Function

The Send_Request function is the mechanism used by a PC requester to ship data and parameters to a host server. As described earlier, issuing a Send_Request is analogous to "calling" a subroutine, and thus it provides a synchronous interface for the requester. When the server returns the results of its processing, control is returned to the PC requester.

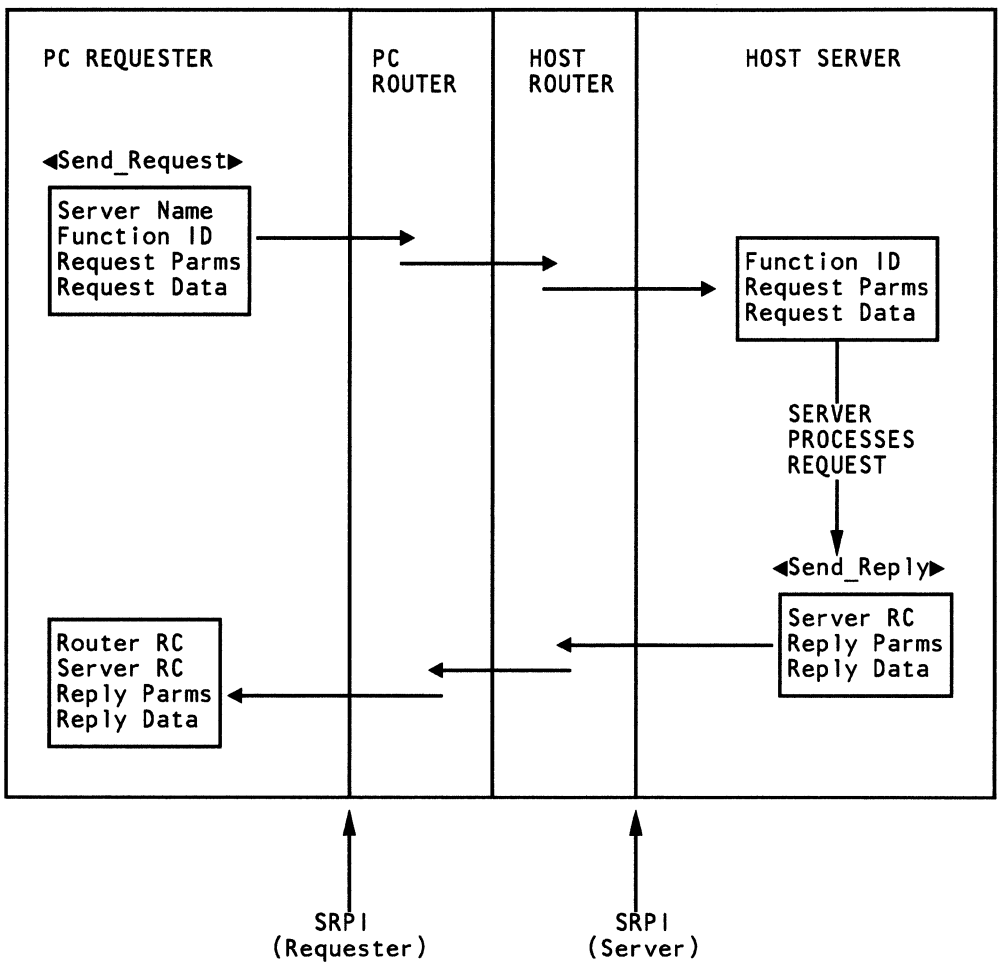


Figure 2. Requester/Server Flow

The requester selects a server by providing a Server Name. The requester optionally may provide a Function_ID, which may be used to specify a particular server function.

In addition to the Server Name and the Function_ID, the requester may send up to 32,763 bytes of parameters and up to 65,535 bytes of data to the host server program with a single Send_Request. To send up to 32,763 bytes of parameter information to the server, the requester supplies the parameter length in bytes and a pointer to a buffer containing the parameters. Similarly, to send up to 65,535 bytes of data to the server, the requester supplies the data length in bytes and a pointer to a buffer containing the data.

The requester also specifies the maximum number of parameters and amount of data it is willing to receive from the server. The requester may receive up to 32,763 bytes of reply parameters and 65,535 bytes of reply data from the server. The requester must supply the buffer(s) required to receive the parameters and/or data from the server.

The parameters associated with the Send_Request function are described in greater detail on the following pages.

Parameters Supplied by the PC Requester

Figure 3 describes the various parameters supplied by a PC requester using the Send_Request function. The parameters which have default values can be initialized to their default values by using the appropriate initialization procedure provided with each of the language interfaces supplied by IBM.

Name of Parameter	Required/ Optional	Default Value	Description
Server	Required	None	The name of the S/370 server must be 8 bytes long (PC/ASCII), left justified, and padded with blanks. Leading blanks, embedded blanks and a name consisting of all blanks are invalid. Valid characters (for an English system) consist of A to Z (upper and lower case) and three special characters (#, $ and @).
Function ID	Optional	0	A 2-byte binary number that specifies the server function being requested. Values of 0 to 65,535 are valid for specification by a requester.
Request Parameters Buffer Length	Optional	0	A 2-byte unsigned binary length that specifies the byte length of the request parameters to be passed to the server. Values of 0 to 32,763 are valid. A value of 0 indicates that there are no request parameters to be sent to the server.
Request Parameters Buffer	Optional	None	The 4-byte address of the parameters, if any, to be passed to the server. A non-zero value in the request parameters buffer length indicates that there are parameters to be passed.
Request Data Buffer Length	Optional	0	A 2-byte unsigned binary length that specifies the byte length of the request data to be passed to the server. Values of 0 to 65,535 are valid. A value of 0 indicates that there is no request data to be sent to the server.
Request Data Buffer	Optional	None	The 4-byte address of the data, if any, to be passed to the server. A non-zero value in the request data buffer length indicates that there is data to be passed.
Reply Parameters Buffer Length	Optional	0	A 2-byte unsigned binary length that specifies the byte length of the reply parameter buffer supplied by the requester. Values of 0 to 32,763 are valid. A value of 0 indicates that no reply parameters are expected from the server.
Reply Parameters Buffer	Optional	None	The 4-byte address of the reply parameter buffer. Its presence is indicated by a non-zero Reply Parameters Buffer Length.
Reply Data Buffer Length	Optional	0	A 2-byte unsigned binary length that specifies the number of bytes in the reply data buffer supplied by the requester. Values of 0 to 65,535 are valid. A value of 0 indicates that no reply data is expected from the server.
Reply Data Buffer	Optional	None	The 4-byte address of the reply data buffer. Its presence is indicated by a non-zero reply data buffer length.

Figure 3. Requester Supplied Parameters

Parameters Returned to the PC Requester

The following parameters are returned to the PC requester when processing of the Send__Request function is complete.

Name of Parameter	Description
PC Router Return Code	A 4-byte return code indicating the results of the Send__Request execution. An OK return code indicates the host server was successfully invoked. See Appendix A of "IBM Enhanced Connectivity Facilities Programmer's Guide for the Personal Computer and the 3270 Personal Computer, SC23-0959-0"for a complete description of return codes.
Server Return Code	The return code returned from the host server. The content and the format of the server return code are defined by the individual host server. The server return code is always 32 bits.
Replied Parameter Length	A 2-byte unsigned binary length that specifies the number in bytes of the parameters returned from the server. The value will be less than or equal to the value specified by the requester in the reply parameters buffer length.
Replied Data Length	A 2-byte unsigned binary length that specifies the number in bytes of the data returned from the server. The value will be less than or equal to the value specified in the reply data buffer length.

Figure 4. Requester Returned to PC Requester

Usage Notes

Notes: The PC Router Return Code should be examined first by the requester application. If the PC Router Return Code is not successful, the following fields are undefined:

- Server Return Code
- Replied Parameter Length
- Replied Data Length

This means that these fields may or may not have been altered by the PC Router. Therefore, the calling application should not expect these fields to be either maintained or altered across any unsuccessful call to the SRPI.

The Server Return Code is application data generated by the host server.

The server name is translated from ASCII into an EBCDIC representation prior to transmission to the host router.

Language Interfaces

IBM provides interface routines to support requesters written in three languages. The interface routines supplied by IBM support the following languages.

1. IBM Pascal Compiler, Version 2.0
2. IBM C Compiler, Version 1.0
3. IBM Macro Assembler, Version 1.00 and Version 2.00

By using the IBM Macro Assembler support, users can write their own interface routines to other high-level languages. The interface routines are included on the product diskette(s) for:

- IBM PC 3270 Emulation Program, Version 3.0
- IBM 3270 PC Control Program, Version 3.0

Sample PC Requester Program Overview

Three sample requester program segments are provided in the following sections, one for each of the language interfaces provided by IBM. Each sample program builds a version of the Connectivity Programming Request Block (CPRB). The CPRB is the control block used to issue the Send__Request function.

A programmer using Pascal or C will use a logical representation of the CPRB, while the Macro Assembler programmer will use a CPRB mapping which matches the physical control block mapping. A file for each of the three CPRB mappings is provided on the product diskette(s):

```
Pascal          -  UUPCPRB.INC
C COMPILER      -  UUCCPRB.H
Macro Assembler -  UUMCPRB.INC
```

Each sample requester performs the same function. A Send__Request is sent to the host with:

- Function ID set to "READ"
- Server name set to "IBMabase"
- The request parameter buffer containing a flag set to "COMMIT"

The hypothetical server reads a record from the database, and returns the record in the reply data buffer. The database record returned from the server contains:

- Customer name
- Customer address
- Customer balance

The sample programs continue with normal processing if the customer balance returned in the reply data buffer is positive. The following high-level algorithm describes the processing:

- Initialize the request parameters buffer
- Initialize default values in CPRB, using send_req_init procedure
- Set server name in CPRB
- Set function ID in CPRB
- Set request parameter buffer address and length in CPRB
- Set reply data buffer address and length in CPRB
- Issue Send_Request function
- If SRPI router return code OK
  - If server return code OK
    - If customer balance positive
    - Continue processing

Pascal Language Interface

Overview

This section is for programmers interested in writing a requester in Pascal. To assist the programmer, the following files are included on the program product diskette(s):

```
UUPINIT.OBJ  - Subroutine to initialize the Pascal CPRB.
UUPSENDR.OBJ - Sendrequest subroutine.
UUPCPRB.INC  - Pascal CPRB record definition and return code mappings.
UUPPROCS.INC - External routine declarations.
```

By linking UUPINIT.OBJ and UUPSENDR.OBJ with the compiled Pascal requester program, the programmer can easily access the SRPI function.

The parameters on the Send Request function are grouped in a single Pascal record structure of type UERCPRB. The init__send__req__parms procedure can be used to initialize all the default sendrequest parameters in the structure of type UERCPRB.

The Pascal procedure which implements the Send__Request function is called "sendrequest". The sendrequest procedure has a single parameter, which is the 32-bit address of a UERCPRB record. The UERCPRB record contains the server name, the Function__ID, and pointers to the parameters and data to be sent to the server. The UERCPRB record also contains pointers to buffers to receive parameters and data from the host server.

Sample Pascal Requester

```
program psample;
        (*$INCLUDE:  'UUPCPRB.INC'*)
        (*$INCLUDE:  'UUPPROCS.INC'*)
const
        func1       = 1;                        (* Read record          *)
        rcok        = #00000000;                (* Server RC OK         *)
        server      = 'IBMabase';               (* Server name          *)

type    custrec = record                        (* Customer record      *)
           cusname       [00]: string(25);      (* Customer name        *)
           cusaddr       [25]: string(25);      (* Customer address     *)
           cusbal        [50]: real;            (* Balance              *)
        end;

type    parms = record                          (* Request Parms        *)
           flags         [00]: byte;            (* Flags                *)
        end;

const   qpacom      = #02;                      (* Commit               *)

var     cprbads       :   UERCPRBPTR;           (* CPRB address         *)
        retcode       :   integer4;             (* Return Code          *)
        pcprb         :   uercprb;              (* CPRB record          *)
        pqparms       :   parms;                (* Request parms        *)
        prcustrec     :   custrec;              (* Customer record      *)

begin
pqparms.flags := qpacom;                        (* Set Request Parms    *)
cprbads  :=  ADS pcprb;                         (* Init CPRB record     *)
init_send_req_parms(cprbads);

pcprb.uerserver := server;                      (* Set server name      *)
pcprb.uerfunct  := func1;                       (* Set function ID      *)

pcprb.uerqparml  := sizeof(pqparms);      (* Set request parms length *)
pcprb.uerqparmad := ADS pqparms;          (* Set request parms addr   *)

pcprb.uerrdatal  := sizeof(prcustrec);    (* Set reply data length    *)
pcprb.uerrdataad := ADS prcustrec;        (* Set reply data address   *)

retcode  :=  sendrequest(cprbads);              (* Issue send request   *)
if retcode = UERERROK then                      (* If RC ok             *)
begin
   if pcprb.uerservrc = rcok then               (* If server RC ok      *)
   begin
      if prcustrec.cusbal > 0 then              (* Check balance        *)
      begin
         (* CONTINUE PROCESSING                                        *)
      end;
   end;
end;
end;                                            (* endproc              *)
```

Pascal CPRB Mapping

The following mapping is defined in UUPCPRB.INC.

```
(*****************************************************************
*                          CPRB Record                           *
*****************************************************************)
TYPE
    UERCPRBPTR = ADS of uercprb;
    uercprb = RECORD
      uerserver     : string(8);   { ASCII name of server       }
      uerfunct      : word;        { Function ID                }

      uerqparml     : word;        { Request Parameters Length  }
      uerqparmad    : adsmem;      { Request Parameters Address }
      uerqdatal     : word;        { Request Data Length        }
      uerqdataad    : adsmem;      { Request Data Address       }

      uerrparml     : word;        { Reply Parameters Length    }
      uerrparmad    : adsmem;      { Reply Parameters Address   }
      uerrdatal     : word;        { Reply Data Length          }
      uerrdataad    : adsmem;      { Reply Data Address         }

      uerretcode    : integer4;    { Return Code                }
      uerservrc     : integer4;    { Server Return Code         }
      uerrepldplen  : word;        { Replied Parameters Length  }
      uerrepldlen   : word;        { Replied Data Length        }
    end;
```

C Language Interface

Overview

This section is for programmers interested in writing a requester in C. To assist the programmer, the following files are included on the program product diskette(s):

```
UUCINIT.OBJ  - Subroutine to initialize the C CPRB.
UUCSENDR.OBJ - Send request subroutine.
UUCCPRB.H    - C CPRB record definition and return code mappings.
```

By linking UUCINIT.OBJ and UUCSENDR.OBJ with the compiled C requester program, the programmer can easily access the SRPI function.

The parameters on the send__request function are grouped in a single C structure of type UERCPRB. The init__send__req__parms procedure can be used to initialize all the default send__request parameters in the structure. The C send__request function has a single parameter which is the 32-bit address of a structure of type UERCPRB. The UERCPRB structure contains the server name, Function ID and pointers to the parameters and data to be sent to the server. The UERCPRB record also contains pointers to buffers to receive parameters and data from the host server.

Sample C Requester

```
#include <uuccprb.h>

char    cserver[9] = "IBMabase";        /* Server Name                  */

main()
{
UERCPRB ccprb;                          /* CPRB structure               */
struct  {                               /* Customer Record Structure    */
        char    cusname[25];            /* Customer Name                */
        char    cusaddr[25];            /* Street Address               */
        float   cusbal;                 /* Balance                      */
        }       ccustrec;

struct  {                               /* Request Parameters Struct    */
        char    qpaflags;               /* Processing Flags             */
#define QPACOM 0x02                     /* Commit transaction           */
        }       cqparms;

#define FUNC1  1                        /* Func Code: Get Record        */
#define RCOK 0x00000000                 /* Server Return Code OK        */

        long int retcod;                /* SRPI router return code      */

                                        /* PROC (MAIN)                  */
cqparms.qpaflags = QPACOM;              /* SET PROCESSING OPTION =      */
                                        /* COMMIT TRANSACTION           */
init_send_req_parms(&ccprb);            /* INITIALIZE CPRB STRUCTURE    */

ccprb.uerserver = cserver;              /* MOVE SERVER NAME & FUNCTION*/
ccprb.uerfunct = FUNC1;                 /* INTO CPRB STRUCTURE          */

ccprb.uerqparml = sizeof cqparms;       /* SET CPRB REQUEST PARAMETER */
ccprb.uerqparmad = &cqparms;            /* BUFFER INFORMATION           */

ccprb.uerrdatal = sizeof ccustrec;      /* SET CPRB REPLY DATA BUFFER */
ccprb.uerrdataad = &ccustrec;           /* INFORMATION                  */

retcod = send_request(&ccprb);          /* SEND REQUEST TO SERVER       */

if (retcod == UERERROK)                 /* SRPI router RC is good       */
{
  if (ccprb.uerservrc == RCOK)          /* SERVER RETURN CODE IS GOOD */
  {
      if (ccustrec.cusbal > 0)          /* ACCNT BALANCE IS POSITIVE  */
      {
        /* CONTINUE PROCESSING                                        */
      }                                 /* ENDIF                        */
  }                                     /* ENDIF                        */
}                                       /* ENDIF                        */
}                                       /* ENDPROC (MAIN)               */
```

C CPRB Mapping

The following mapping is defined in UUCCPRB.H.

```
/*************************************************************/
/*                      CPRB Structure                       */
/*************************************************************/

typedef struct {

     /* supplied parameters -- not changed by Send_Request */

      char far      *uerserver;  /* ASCII name of server       */
      unsigned int uerfunct;     /* Function ID                */

     /* request parameters and data */

      int       uerqparml;       /* Request Parameters Length  */
      char far  *uerqparmad;     /* Request Parameters Address */
      unsigned int   uerqdatal;  /* Request Data Length        */
      char far  *uerqdataad;     /* Request Data Address       */

     /* reply parameters and data */

      int       uerrparml;       /* Reply Parameters Length    */
      char far  *uerrparmad;     /* Reply Parameters Address   */
      unsigned int   uerrdatal;  /* Reply Data Length          */
      char far  *uerrdataad;     /* Reply Data Address         */

     /* returned parameters */

      long int  uerretcode;      /* Return Code                */
      long int  uerservrc;       /* Server Return Code         */
      int       uerrepldplen;    /* Replied Parameters Length  */
      unsigned int  uerrepldlen; /* Replied Data Length        */
      } UERCPRB;

/*************************************************************/
/*                  Interface Routines                       */
/*************************************************************/

extern void init_send_req_parms(UERCPRB far *);
extern long int send_request(UERCPRB far *);
```

Macro Assembler Language Interface

Overview

This section is for programmers interested in writing a requester in Macro Assembler. To assist the programmer, the following files are included on the program product diskette(s):

```
UUMINFAC.MAC  - Various macros to build CPRB and Send_Request (Source).
UUMCPRB.INC   - Macro Assembler CPRB mapping.
```

The programmer issues a Send__Request via a set of IBM supplied macros. The programmer must insure that the ES:DI registers point to the CPRB whenever invoking any of the macros.

All macro parameters are positional. The macros may be invoked with null parameters. When a parameter is null, the corresponding field in the CPRB is not accessed. All parameters are optional in terms of invoking macros. The requester application should not issue the send__request verb until all fields in the CPRB have been set to their intended values.

The following macros are provided to assist the programmer in building the CPRB, issuing the Send__Request and obtaining the returned parameters.

```
SEND_REQ_INIT - Sets default values in the CPRB

- EXAMPLE:  SEND_REQ_INIT

SET_REQ_PARMS - Sets server name and function ID.

- EXAMPLE:  SET_REQ_PARMS SERV_NAM,FUNCT

SET_REQ_BUFFERS - Sets the values
of the request data and request parameters
buffers and the corresponding lengths.

- EXAMPLE:  SET_REQ_BUFFERS QPARM_BUF,QPARM_LEN,QDATA_BUF,QDATA_LEN
- EXAMPLE:  SET_REQ_BUFFERS ,,QDATA_BUF,QDATA_LEN

SET_REPLY_BUFFERS - Sets the values of the reply data and reply
parameters buffers and the corresponding lengths.

- EXAMPLE: SET_REPLY_BUFFERS PARM_BUF,PARM_LEN,DATA_BUF,DATA_LEN

SEND_REQUEST - Executes the send_request by issuing an interrupt.

- EXAMPLE:  SEND_REQUEST

GET_REPLY - Retrieves the parameters returned when a send_request
has been processed.

- EXAMPLE:  GET_REPLY RET_CODE,SERV_RC,REP_PARM_LEN,REP_DATA_LEN
```

Sample Macro Assembler Requester

```
        INCLUDE uuminfac.mac
        INCLUDE uumcprb.inc
;---------------------------------------------------------------
        SUBTTL  'Customer Record Mapping'
mcustrec      STRUC
mcusname      db      25 dup (?)      ;name
mcusaddr      db      25 dup (?)      ;street address
mcusbal       dd      ?               ;balance
mcustrec      ENDS
;---------------------------------------------------------------
        SUBTTL  'Request Parameters Mapping'
mqparms       STRUC
mqpaflags     db      ?               ;Processing flags
mqparms       ENDS
;
;Equates for processing flags defined in STRUC mqparms
mqpacom       equ     02H             ;Commit the transaction
;---------------------------------------------------------------
        SUBTTL  'mWORK - Work Area Segment'
mwork         SEGMENT 'data'
mdabuf        db      SIZE mcustrec dup (?)  ;Allocate buffer
;                                             for customer records
mdabuf@       dd      mdabuf                 ;Vector to customer
;                                             record buffer
mdabufl       equ     SIZE mcustrec          ;Length of a customer
;                                             record
mqprmbuf      db      SIZE mqparms dup (?) ;Allocate a buffer
;                                             for request parms
mqprmbuf@     dd      mqprmbuf               ;Vector to request
;                                             parameters buffer
mqprmbufl     equ     SIZE mqparms           ;Length of a request
;                                             parameters

mserver_l$    equ     $                      ;First character of
;                                             server name
mserver       db      'IBMabase'             ;Server name
mserver_len$  equ     $-mserver_l$           ;Length of server name

mfunc1        equ     1                      ;Func code: Get Record
mrcok         equ     0000H                  ;Return Code: OK

mretcode      dd      ?                      ;SRPI router RC
                org     mretcode-mwork
mrclow        dw      ?                      ;Low word of return code
mrchigh       dw      ?                      ;High word of return code

mservrc       dd      ?                      ;Server Return Code
                org     mservrc-mwork
msrvrclow     dw      ?                      ;Low word of return code
msrvrchigh    dw      ?                      ;High word of return code
mwork         ENDS
;---------------------------------------------------------------
mcprbseg      SEGMENT 'data'
mcprb         db      SIZE uercprb dup (0FFH) ;Allocate space
;                                              for CPRB
mcprbseg      ENDS
;---------------------------------------------------------------
```

```
mstack        SEGMENT stack 'stack'
              dw      255 dup (OFFFFH) ;Allocate a stack
mstaktop      dw      OFFFFH           ;First stack entry
mstack        ENDS
;******************-END DEFINITIONS-*****************************
        SUBTTL  'Main procedure'
msampl        segment 'code'
        assume  cs:msampl
;************************-PROCEDURE*******************************
;/                       PROC (MAIN)                               *
mentry:
;/                      1. ESTABLISH A STACK                       *
        assume ss:mstack
        mov     ax,seg mstack
        mov     ss,ax
        mov     sp,offset mstaktop
;/                      1. SET DS TO POINT TO WORK AREA            *
        assume  ds:mwork
        mov     ax,seg mwork
        mov     ds,ax
;/                      1. GET ADDRESS OF REQUEST PARAMETERS       *
        assume  es:mwork
        les     di,mqprmbuf@            ;ES:DI -> request
;                                        parameters buffer
;/                      1. SET PROCESSING OPTION = COMMIT
        mov     BYTE PTR es:[di+mqpaflags],mqpacom
;/                      1. GET ADDRESS OF CPRB INTO ES:DI          *
        assume  es:mcprbseg
        mov     ax,SEG mcprbseg
        mov     es,ax
        mov     di,OFFSET mcprb         ;ES:DI -> CPRB
;/                      2. . INITIALIZE THE CPRB <SEND_REQ_INIT>   *
        SEND_REQ_INIT
;/                      2. . MOVE SERVER NAME AND FUNCTION (GET
;&                             RECORD) INTO CPRB <SET_REQ_PARMS>   *
        SET_REQ_PARMS   mserver,mfunc1
;/                      2. . SET CPRB REQUEST PARAMETERS BUFFER
;&                             INFORMATION <SET_REQ_BUFFERS>       *
        SET_REQ_BUFFERS mqprmbuf@,mqprmbuf1
;/                      2. . SET CPRB REPLY DATA BUFFER INFORMATION
;&                             <SET_REPLY_PARMS>                   *
        SET_REPLY_BUFFERS ,,mdabuf@,mdabuf1
;/                      2. . SEND THE REQUEST TO THE SERVER
;&                             <SEND_REQUEST>                      *
        SEND_REQUEST
;/                      2. . GET SRPI ROUTER RC AND SERVER RC
        GET_REPLY  mretcode,mservrc
;/                      2. . IF SRPI ROUTER RC IS GOOD             *
        cmp     mrchigh,uererrokeq
        je      goodrc1                 ;exit label is >127
        jmp     end                     ; bytes away
```

```
goodrc1:
;/                      3. . .IF THE SERVER RETURN CODE IS GOOD         *
        cmp     msrvrchigh,mrcok        ;Compare high word of server rc
        je      goodrc2                 ;exit label is >127
        jmp     end                     ; bytes away
goodrc2:
        cmp     msrvrclow,mrcok         ;Compare low word of server rc
        jne     end
;/                      4. . . . IF THE ACCOUNT BALANCE IS POSITIVE     *
        mov     si,WORD PTR mdabuf@     ;get offset of data buf,
;                                         DS:SI -> data buffer
        mov     ax,WORD PTR [si+mcusbal] ;Get low word of balance
        mov     dx,WORD PTR [si+mcusbal+2]  ;Get high word of balance
        sub     dx,0                    ;Subtract zero from the high word
        jl      end                     ;Negative balance, quit
        jg      continue                ;Positive balance, continue
        cmp     ax,0                    ;Is low word zero?
        je      end                     ;Yes-zero balance, quit
continue:
;       CONTINUE PROCESSING
end:
;/                      1. RETURN TO DOS                                *
          mov     ax,4C00H              ;Return to DOS with
          int     21H                   ;return code zero
msampl        ENDS
        END     mentry
```

Macro Assembler CPRB Mapping

```
;;*****************************************************************************
;;*                    CPRB MAPPING                                           *
;;*****************************************************************************
;;
uercprb         STRUC
uerrbsiz         dw    ?                ;Size of CPRB in bytes
uerversion       dw    ?                ;Version Number
;uerversnum      EQU   0100H            ;Current version value

uerretcode       dd    ?                ;Return Code

uerverbtyp       db    ?                ;Verb Type
;uersendreq      EQU   1                ;Send_Request

                 db    ?                ;Reserved

uerfunct         dw    ?                ;Function ID

                 dw    ?                ;Reserved

uerqparml        dw    ?                ;Request Parameters Length
uerqparmad       dd    ?                ;Request Parameters Address

uerqdatal        dw    ?                ;Request Data Length
uerqdataad       dd    ?                ;Request Data Address

uerrparml        dw    ?                ;Reply Parameters Length
uerrparmad       dd    ?                ;Reply Parameters Address

uerrdatal        dw    ?                ;Reply Data Length
uerrdataad       dd    ?                ;Reply Data Address

                 dw    ?                ;Reserved

uerservrc        dd    ?                ;Server Return Code

uerrepldplen     dw    ?                ;Replied Parameters Length
uerrepldlen      dw    ?                ;Replied Data Length

uerwkarea        db    46 dup(?)        ;Work Area

uersrvnml        dw    ?                ;Server Name field length
uerserver        db    8 dup(?)         ;Server Name
uercprb          ENDS
;  -----------  Field Definitions  -----------
uerversnum       EQU   0100H            ;Current version value
uersendreq       EQU   1                ;Send_Request
uersubfunct      EQU   0103H            ;Sub function (register AX)
uerinterrupt     EQU   07FH             ;Call/Return interrupt value
```

IBM PC Interrupt Sharing Protocol

Introduction

This article defines and establishes an Interrupt Sharing protocol that enables multiple hardware adapters on the PC Bus to share a single interrupt request line. This protocol allows an interrupt request line to be used either by multiple sharing adapters or exclusively by one nonsharing adapter. Sharing and nonsharing adapters cannot be physically intermixed on the same interrupt request line.

In Interrupt Sharing, the shared interrupt hardware allows the interrupt line to float high, and each interrupting device requiring service is allowed to momentarily pulse this line low. The protocol permits only one interrupt to be serviced at any given time, with other adapters required to retain the status of pending interrupts.

Each adapter must provide an Interrupt Status bit and an Interrupt Enable bit that can be set, reset or interrogated by system software. The adapter sets the Interrupt Status bit when it has an interrupt pending and the bit is reset by software only after the interrupt has been serviced. Since only one adapter is serviced at any given time, this bit when set ensures that the device requiring interrupt service eventually will be serviced. The Enable bit is set/reset by software to enable/disable the adapter's interrupting capability. The interrupt handler software resets this bit, before unlinking from the chain of interrupt handlers, to ensure that adapters do not cause system lockups by interrupting when the handler is unlinked. The Enable bit defaults to a disabled state following a system initialization or reset.

A multitasking operating system environment needs a software scheme that allows tasks to link their interrupt handler onto a chain of interrupt handlers, share the interrupt line while the task is active, and then unchain their interrupt handler from the chain once the task is completed. Interrupt handlers for all adapters sharing an interrupt request line must be linked to the chain of interrupt handlers while the task is active.

This Interrupt Sharing protocol is also documented in the last release of the PC AT Technical Reference Manual (P/N 6280070) dated September 1985. Corrections or additions to the information in the PC AT Technical Reference Manual are indicated by the "vertical change bar" in the left margin.

Interrupt Sharing Hardware

Figure 5 illustrates a typical hardware implementation for controlling an adapter's interrupt request output in order to share an interrupt request line. An interrupt originating from the adapter (Interrupt Status bit) is AND'ed with the adapter's Interrupt Enable bit, and the result is used to activate the interrupt request pulse generating circuits unless an interrupt pulse is already active on the IRQ line.

The adapter's interrupt pulse, propagated through the pulse generating circuits, must be between 125 and 1000 nanoseconds. The high-to-low transition of the pulse is used to arm the Interrupt Controller, while the low-to-high transition causes the Interrupt Controller to recognize the interrupt. All adapters, including the requesting adapter, must latch the interrupt pulse on the IRQ line to disable their pulse generating circuits. Disabling the pulse generating circuits prevents adapters from generating interrupts while an existing interrupt is being serviced. Therefore, all adapters sharing an interrupt level are required to monitor the IRQ line.

It is the responsibility of the interrupt handler to rearm the interrupt sharing logic on the adapters after the interrupt has been serviced. Only after a Global Rearm (reset of latched condition) is issued can an adapter once again place its request on the IRQ line. This global rearm is accomplished by an I/O Write to address 02FXH or 06FXH, depending upon the interrupt level. (The X in the address equates to the interrupt level being serviced; i.e., 02F2H for level 2 or 9, 02F7H for level 7, 06F2H for level 10, 06F7H for level 15.) An adapter is required to decode the least significant 11 address bits for this rearm.

An adapter must reissue its interrupt if its Interrupt Status bit is set when the interrupt sharing logic is rearmed. Since only one adapter can be serviced at any given time, this prevents lost interrupts when multiple adapters have interrupts outstanding.

Interrupts must never be generated when an interrupt handler is not available to service the adapter card's interrupt. This prevents the possibility of locking up an interrupt level. All designs for adapter cards must ensure that their interrupts can be disabled and not allowed to remain active after their application has terminated.

The Interrupt Status bit and Interrupt Enable bit must be reset during a system reset condition.

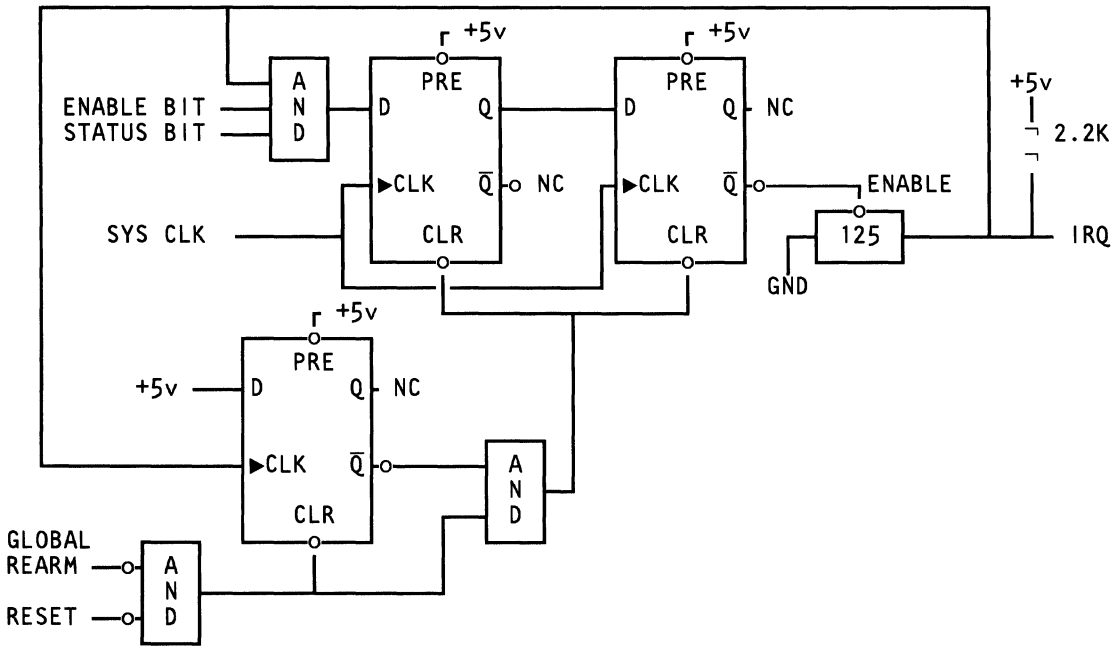


Figure 5. SHARED INTERRUPT HARDWARE LOGIC

Interrupt Sharing Software

Interrupt Sharing software operating in a multitasking environment must support the linking of a tasks interrupt handler to a chain of interrupt handlers, the sharing of the interrupt level while the task is active, and the unlinking of the interrupt handler from the chain once the task is complete.

To link an interrupt handler, the newly activated task's interrupt handler replaces the interrupt vector in low memory with a pointer to its own interrupt handler. (Refer to the section "ROS Considerations" for interrupt handlers stored in ROS). The interrupt handler must preserve the interrupt vector it is replacing and use it as a forward pointer to the next interrupt handler in the chain. This old interrupt vector must be stored at a fixed offset from the entry point of the new task's interrupt handler.

When the system acknowledges an interrupt request, each interrupt handler must determine whether it is the appropriate interrupt handler for the adapter presenting the interrupt request. This is accomplished by the handler reading the contents of its adapter's Interrupt Status register.

If the handler's device caused the interrupt, the handler must service the interrupt, reset the Interrupt Status bit, clear the interrupts (CLI), issue a nonspecific End of Interrupt (EOI), issue a Global Rearm (I/O Write to address "02FX" or or "06FX") and then execute a Return From Interrupt (IRET).

If the handler's device did not cause the interrupt, the handler passes control to the next interrupt handler in the chain via the use of the previously stored forward pointer.

To unlink an interrupt handler from a chain, the task first locates its handler's position within the chain. By starting at the interrupt vector in low memory and using the offset of each handler's forward pointer to locate the entry point of each handler, the chain can be searched until the task finds its own handler. Each interrupt handler's signature (424BH) (Refer to the Section "Interrupt Chaining Structure") must be checked to ensure that a valid forward pointer exists. The task's forward pointer replaces the forward pointer of the previous handler in the chain, thus removing the handler from the chain. NOTE: If the interrupt handler can not locate its position in the chain, the interrupt handler can not unlink.

An application-dependent unlinking error-recovery procedure must be incorporated into the unlinking routine for those situations where the unlinking routine discovers that the interrupt chain has been corrupted (an interrupt handler is linked but does not have a valid signature). All Interrupt Sharing handlers, except those in ROS (refer to the section "ROS Considerations"), must use 424BH as the signature to avoid corrupting the chain.

During a system reset condition, a short routine for each interrupt handler must be executed with the AUTOEXEC.BAT to disable interrupts from their responsible devices.

Interrupt Chaining Structure

The Interrupt Sharing software chaining structure is in a 16-byte format containing a 4-byte forward pointer (FPTR), a 2-byte signature and 8 reserved bytes (RES__BYTES) as depicted in the following coding example. It begins at the third byte from the interrupt handler's entry point. The first instruction of every handler is a short jump around the structure, placing the structure at a known offset from the beginning of the handler routine. Since the position of each interrupt handler's chaining structure is known (except for the handlers on adapter ROS), the FPTRs can be updated when linking and unlinking.

The FIRST flag is used to determine the handler's position in the chain when linking and unlinking for shared interrupt levels 7 and 15 only. The RESET routine, an entry point for the operating system, must disable the adapter's interrupt and return (FAR) to the operating system.

```
ENTRY:  JMP          SHORT   PAST       ; Jump around structure
        FPTR         DD      0          ; Forward Pointer
        SIGNATURE    DW      424BH      ; Used when unlinking to identify
                                        ;  compatible interrupt handlers
        FLAGS        DB      0          ; Flags
        FIRST        EQU     80H        ; Flag for being first in chain
        JMP          SHORT   RESET
        RES_BYTES    DB      DUP 7(0)   ;Future Expansion
PAST:   ...                             ;Actual start of code
```

ROS Considerations

Adapters with interrupt handlers in ROS *must* implement chaining by storing the 4 byte forward pointer (FPTR) in on-adapter latches or ports. If the adapter is sharing interrupt levels 7 or 15, it must also store the FIRST flag that indicates whether it is positioned first in the chain of interrupt handlers. Storage of this information is required because it can not be guaranteed that handlers in ROS will always link first and never unlink. The ROS handler must contain the signature 0000H beginning at the seventh byte from the handler entry point since the forward pointer in ROS handlers is not stored at the third byte from the handler entry point.

Interrupt Sharing Precautions

The following precautions should be taken before implementing Interrupt Sharing:

- The defined Interrupt Sharing protocol is intended to run only in the Real Address Mode (i.e. 80286 Real Address Mode). It is not intended to run in the Virtual Address Mode.
- Interrupts must be disabled before control is passed to the next handler on the chain. Disabling of the interrupts allows the next handler to receive control as if a hardware interrupt had caused it to receive control.
- The interrupts must be disabled before the nonspecific EOI is issued and not reenabled in the interrupt handler to ensure that the IRET is executed (at which point the flags are restored and the interrupts reenabled before another interrupt is serviced, protecting the stack from excessive build up).
- All interrupt handlers must have a short routine that can be executed with the AUTOEXEC.BAT at power-on reset to disable their adapters' interrupts. Execution of this routine, along with a reset of the Interrupt Sharing hardware, ensures that adapters are deactivated if the user reboots the system.
- Interrupt handler implementations must store data in memory in Intel format (i.e. word 424BH is stored as 4B42H in memory).

General Implementation Information

Information about the 8259A Programmable Interrupt Controller can be found in any recent edition of the Intel Microprocessor and Peripherals Handbook. In the IBM PC family, the Interrupt Mask Register lies at I/O Port 21H. Specific End of Interrupt (EOI) values for the various interrupt levels are listed (67H for level 7). The Specific EOI is accomplished by issuing an OUT to the 8259A's operational control register using Operational Control Word 2 (OCW2) (an OUT to I/O Port 20H on the PC family). A NonSpecific EOI is accomplished by issuing an OUT of 20H to the operational control register.

Interrupt Sharing Code Examples

The following are coding examples of a Linking Structure, an Interrupt Handler and an Unlinking Structure implementing the Interrupt Sharing concept.

Linking Code Example

```
            PUSH      ES
            CLI                          ;Clear interrupts
;Set forward pointer to value of interrupt vector in low memory
            ASSUME    CS:CODESEG,DS:CODESEG
            PUSH      ES
            MOV       AX,350FH           ;DOS get interrupt vector
            INT       21H
            MOV       SI,OFFSET CS:FPTR ;Get offset of your forward pointer
                                         ; in an indexable register
            MOV       CS:[SI],BX         ;Store the old interrupt vector
            MOV       CS:[SI+2],ES       ; in your forward pointer for
                                         ; chaining
            CMP       ES:BYTE PTR[BX],CFH ; Test for IRET
            JNZ       SETVECTR
            MOV       CS:FLAGS,FIRST     ;Set up first in chain flag
SETVECTR:   POP       ES
            PUSH      DS
;Make interrupt vector in low memory point to your handler
            MOV       DX,OFFSET ENTRY    ;Make interrupt vector point to
                                         ; your handler
            MOV       AX,SEG ENTRY       ;If DS not = CS, get it and
            MOV       DS,AX              ; put it in DS
            MOV       AX,250FH           ;DOS set interrupt vector
            INT       21H
            POP       DS
;Unmask (enable) interrupts for your level
            IN        AL,IMR             ;Read interrupt mask register
            JMP       $+2                ;IO delay
            AND       AL,07FH            ;Unmask interrupt level 7
            OUT       IMR,AL             ;Write new interrupt mask
            MOV       AL,SPC_EOI         ;Issue specific EOI for level 7
            JMP       $+2                ; to allow pending level 7 interrupts
            OUT       OCR,AL             ; (if any) to be serviced
            STI                          ;Enable interrupts
            POP       ES
```

Notes:

1. The operating system must ensure that the SEG:OFF points to a valid interrupt handler or to an IRET (CFH) for Levels 7 and 15.

2. If your adapter card ROS installs your interrupt handler on Levels 7 or 15 during ROMSCAN (before the operating system is loaded), you must test the SEG:OFF for 0000H or F000H as well as for an IRET to determine if it is first. The IRET test applies to linking only after the operating system is loaded.

Interrupt Handler Example

```
YOUR_CARD   EQU     xxxx            ;Location of your card's interrupt
                                    ; control/status register
ISB         EQU     xx              ;Interrupt bit in your card's
                                    ; interrupt control/status register
REARM       EQU     2F7H            ;Global Rearm location for
                                    ; interrupt level 7
SPC_EOI     EQU     67H             ;Specific EOI for 8259's interrupt
                                    ; level 7
EOI         EQU     20H             ;Non-specific EOI
OCR         EQU     20H             ;Location of 8259 operational
                                    ; control register
IMR         EQU     21H             ;Location of 8259 interrupt mask
                                    ; register
MYCSEG      SEGMENT PARA
            ASSUME  CS:MYCSEG,DS:DSEG
ENTRY       PROC    FAR
            JMP     SHORT PAST      ;Entry point of handler
FPTR        DD      0               ;Forward Pointer
SIGNATURE   DW      424BH           ;Used when unlinking to identify
                                    ; compatible interrupt handlers
FLAGS       DB      0               ;Flags
FIRST       EQU     80H
JMP         SHORT   RESET
RES_BYTES   DB      DUP 7(0)        ;Future expansion
PAST:       STI                     ;Actual start of handler code
            PUSH    ...             ;Save needed registers
            MOV     DX,YOUR_CARD    ;Select your status register
            IN      AL,DX           ;Read the status register
            TEST    AL,ISB          ;Your card caused interrupt?
            JNZ     SERVICE         ;Yes, branch to service logic
            TEST    CS:FLAGS,FIRST  ;Are we the first ones in?
            JNZ     EXIT            ;If yes, branch for EOI and Rearm
            POP     ...             ;Restore registers
            CLI                     ;Clear interrupts
            JMP     DWORD PTR CS:FPTR ;Pass control to next guy on chain
SERVICE:    ...                     ;Service the interrupt
EXIT:
            CLI                     ;Clear the interrupts
            MOV     AL,EOI
            OUT     OCR,AL          ;Issue non-specific EOI to 8259
            MOV     DX,REARM        ;Rearm the cards
            OUT     DX,AL
            POP     ...             ;Restore registers
            IRET
RESET:      ...                     ;Disable your card
            RET                     ;Return FAR to operating system
ENTRY       ENDP
            MYCSEG  ENDS
            END     ENTRY
```

Unlinking Code Example

```
            PUSH    DS
            PUSH    ES
            CLI                         ;Clear interrupts
            MOV     AX,350FH            ;DOS get interrupt vector
            INT     21H                 ;ES : BX points to first of chain
            MOV     CX,ES               ;Pickup segment part of interrupt vector
;Are we the first handler in the chain?
            MOV     AX,CS               ;Get code seg into comparable register
            CMP     BX,OFFSET ENTRY     ;Interrupt vector in low memory
                                        ; pointing to your handler's offset?
            JNE     UNCHAIN_A           ;No, branch
            CMP     AX,CX               ;Vector pointing to your handler's
                                        ; segment?
            JNE     UNCHAIN_A           ;No, branch
;Set interrupt vector in low memory to point to the handler pointed to
; by your pointer
            PUSH    DS
            MOV     DX,WORD PTR CS:FPTR
            MOV     DS,WORD PTR CS:FPTR[2]
            MOV     AX,250FH            ;DOS set interrupt vector
            INT     21H
            POP     DS
            JMP     UNCHAIN_X
UNCHAIN_A:    ; BX = FPTR offset, ES = FPTR segment, CX = CS
            CMP     ES:[BX+6],4B42H     ;Is handler using the appropriate
                                        ; conventions (is SIGNATURE present in
                                        ; the interrupt chaining structure)?
            JNE     exception           ;No, invoke error exception handler
            LDS     SI,ES:[BX+2]        ;Get FPTR's segment and offset
            CMP     SI,OFFSET ENTRY     ;Is this forward pointer pointing to
                                        ; your handler's offset?
            JNE     UNCHAIN_B           ;No, branch
            MOV     CX,DS               ;Move to compare
            CMP     AX,CX               ;Is this forward pointer pointing to
                                        ; your handler's segment?
            JNE     UNCHAIN_B           ;No, branch
;Locate your handler in the chain
            MOV     AX,WORD PTR CS:FPTR ; Get your FPTR's offset
            MOV     ES:[BX+2],AX        ;Replace offset of FPTR of handler
                                        ; that points to you
            MOV     AX,WORD PTR CS:FTPR[2] ; Get your FPTR's segment
            MOV     ES:[BX+4],AX        ;Replace segment of FPTR of handler
                                        ; that points to you
            MOV     AL,CS:FLAGS         ;Get your flags
            AND     AL,FIRST            ;Isolate FIRST flag
            OR      ES:[BX+6],AL        ;Set your first flag into prior routine
            JMP     UNCHAIN_X
UNCHAIN_B:  MOV     BX,SI               ;Move new offset to BX
            PUSH    DS
            PUSH    ES
            JMP     UNCHAIN_A           ;Examine next handler in chain
UNCHAIN_X:  STI                         ;Enable interrupts
            POP     ES
            POP     DS
```

Summary

The IBM PC Interrupt Sharing protocol is a useful enhancement to the PC Bus. This protocol defines both the hardware and software requirements necessary to ensure an orderly sharing of the same interrupt level. Interrupt sharing will ease the task of configuring a system by allowing multiple adapters (that have implemented interrupt sharing) to share the same interrupt level.

IBM Personal Computer Seminar Proceedings

Publication Number	Volume	Topic
	V1.1	Contains identical information as V1.2
(G320-9307)	V1.2	IBM PC DOS 2.0 and 1.1 Comparison Compatibility Guidelines - Application Development 8087 Math Co-Processor IBM Macro Assembler
(G320-9308)	V1.3	IBM PC DOS 2.1 & Comparison to DOS 2.0 and 1.1 IBM PC*jr* Architecture & Compatibility Cartridge BASIC IBM Personal Communications Manager-Modem Drivers
(G320-9309)	V2.1	Contains identical information as V2.2
(G320-9310)	V2.2	IBM Software Support Center International Compatibility Requirements IBM Personal Computer Cluster Program
(G320-9311)	V2.3	IBM Personal Computer Cluster Program Sort, Version 1.00 FORTRAN and Pascal Compiler, Version 2.00 PC*jr* Cartridge Tips and Techniques
(G320-9312)	V2.4	IBM Personal Computer AT Architecture ROM BIOS Compatibility & Software Compatibility IBM PC DOS 3.0
(G320-9313)	V2.5	IBM PC Network Overview, Hardware & Program IBM PC Network BIOS (NETBIOS) Architecture
(G320-9314)	V2.6-1	TopView
(G320-9315)	V2.7	IBM Personal Computer Resident Debug Tool
(G320-9319)	V2.8-1	IBM PC Network SMB Protocol
(G320-9316)	V2.9	IBM Personal Computer XENIX, Version 1.00
(G320-9317)	V2.10	IBM PC Professional Graphics Software IBM PC Graphical Kernel & File Systems IBM Plotting System Library IBM Professional FORTRAN IBM PC Data Acquisition & Control Adapter & SW IBM General Purpose Interface Bus Adapter & SW
(G320-9318)	V2.11-1	IBM Enhanced Graphics Adapter
(G320-9320)	V3.1	IBM PC Information Panel (3295 Plasma Display)
(G320-9321)	V3.2	IBM BASIC Compiler 2.00
(G320-9322)	V3.3	IBM Personal Computer C Compiler
(G320-9323)	V3.4	IBM Asynchronous Communications Server Protocol
(G320-9324)	V3.5	IBM Personal Computer Voice Communications Option
(G320-9325)	V4.1	IBM Personal Computer XENIX, Version 2.00
(G320-9326)	V4.2	IBM Personal Computer System Extensions: - IBM Topview, V1.10 - IBM Graphics Development Toolkit Program, V1.10 - IBM PC Local Area Network Program, V1.10 - IBM PC 3270 Emulation Program, V2.00 IBM Personal Computer Enhanced Keyboard

(G320-9327)	V4.3	IBM PC Convertible
(G360-2697)	V4.4	IBM PC 3270 Emulation Prog. Presentation Space API Advanced Program-to-Program Communications (APPC/PC) Revisable-Form Text Document Content Architecture Document Interchange Architecture Introduction to IBM Enhanced Connectivity Facilities IBM PC Interrupt Sharing Protocol

Notes

G360-2697

IBM Corporation
Editor, IBM Personal Computer Seminar Proceedings
Internal Zip 3636
Post Office Box 1328
Boca Raton, Fl 33429-1328