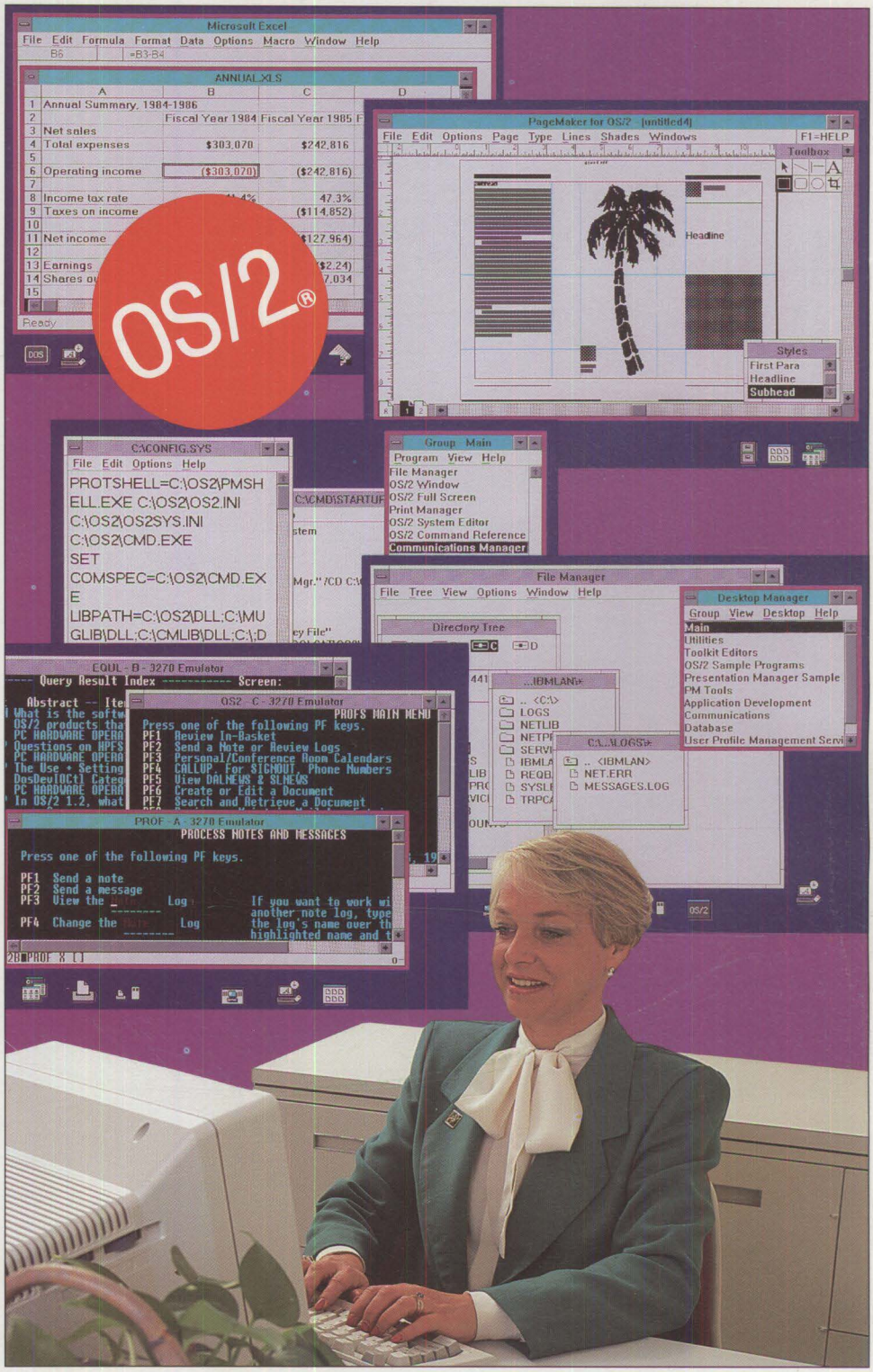
Personal Systems

Issue 2, 1990



IBM Personal Systems Technical Solutions

IBM®

IBM Personal Systems Technical Solutions is published by the U.S. Marketing and Services Group, International Business Machines Corporation, Roanoke, Texas, U.S.A.

Editor	Libby Boyd
Consulting Editor	Ed Bamberger
Technical Consultants	Andrew Frankford
	Chris Jagger
Communications	Elisa Davis
Layout, Cover Design	Corporate Graphics
Illustrator	Bill Carr
Photographer	Layne Murdoch
Automation Consultant	Andrew Frankford
Manager	Bill Hawkins

To correspond with the *IBM Personal Systems Technical Solutions*, please write to the editor at:

IBM Corporation
Internal Zip 40-A2-04
One East Kirkwood Blvd.
Roanoke, TX 76299-0015

To subscribe to this publication, use an IBM System Library Subscription Service (SLSS) form, available at IBM branches, and specify form number GBOF-1229.

Permission to republish information from this publication is granted to publications that are produced for non-commercial use and do not charge a fee. When republishing, please include the names and companies of authors, and please add the words "Reprinted by permission of *IBM Personal Systems Technical Solutions*."

Titles and abstracts, but no other portions, of information in this publication may be copied and distributed by computer-based and other information-service systems. Permission to republish information from this publication in any other publication or computer-based information system must be obtained from the editor.

IBM believes the statements contained herein are accurate as of the date of publication of this document. However, IBM hereby disclaims all warranties as to materials and workmanship, either expressed or implied, including without limitation any implied warranty of merchantability or fitness for a particular purpose. In no event will IBM be liable to you for any damages, including any lost profits, lost savings or other incidental or consequential damage arising out of the use or inability to use any information provided through this service even if IBM has been advised of the possibility of such damages, or for any claim by any other party.

Some states do not allow the limitation or exclusion of liability for incidental or consequential damages so the above limitation or exclusion may not apply to you.

This publication could contain technical inaccuracies or typographical errors. Also, illustrations contained herein may show prototype equipment. Your system configuration may differ slightly.

IBM has tested the programs contained in this publication. However, IBM does not guarantee that the programs contain no errors.

This information is not intended to be a statement of direction or an assertion of future action. IBM expressly reserves the right to change or withdraw current products that may or may not have the same characteristics or codes listed in this publication. Should IBM modify its products in a way that may affect the information contained in this publication, IBM assumes no obligation whatever to inform any user of the modifications.

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 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.

© Copyright 1990 by International Business Machines Corporation

Contents

Software

Operating System/2™

Random Data

Trademarks

IBM, OS/2, Personal Computer AT, AT, Personal System/2, PROFS, PS/2, Quietwriter, RT, and Wheelwriter are registered trademarks of International Business Machines Corporation. AIX, Assistant Series, Audio Visual Connection, AVC, C/2, COBOL/2, Commnication Manager, Database Manager, DisplayWrite 4, FORTRAN/2, Independence Series, KnowledgeTool, Micro Channel, MVS/ESA, MVS/XA,MVS/SP, NetView, NetView/PC, Operating System/2, OS/400, Pascal/2, Personal Computer XT, PC XT, PhoneC-ommnunicator, Plant Floor Series, Presentation Manager, Screen Reader, SpeechViewer, Storyboard, Systems Application Architecture, SAA, System/360, VM/XA, and 3090 are trademarks of International Business Machines Corporation.

Accent is a trademark of AIcom Corporation.
C++ is a registered trademark of AT&T.
PageMaker is a registered trademark and Aldus is a trademark of Aldus Corporation.
Apollo is a trademark of Dolphin Systems, Incorporated.
Audapter is trademark of Personal Data Systems.
Calltext 5050 5010 and Prose 2020 are trademarks of SpeechPlus, Incorporated.
Crosstalk is a trademark of DCA Communication.
dBase III Plus is a trademark of Ashton-Tate, Corporation.
DECtalk is a trademark of Digital Equipment Corporation.
Digital Research and CP/M are registered trademarks of Digital Research, Incorporated.
EASEL is a registered trademark of Interactive Images, Incorporated.
Echo PC is a trademark of Street Electronics Corporation.
Epson is a trademark of Epson Corporation.
Ethernet is a trademark of Xerox Corporation.
Intel is a registered trademark and i860, 386, i486, 80286, 80386, 80386SX, and 80486 are trademarks of Intel Corporation.
LP-DOS is trademark of VisionWare, Incorporated.
Logitech is a trademark of Logitech.
Lotus is a registered trademark of Lotus Development Corporation.
Lotus 1-2-3 is a trademark of Lotus Development Corporation.
Macintosh is a registered trademark of Apple Computer, Incorporated.
Microsoft, CodeView and MS-DOS are registered trademarks and Windows is a trademark of Microsoft Corporation.
MultiScope is a trademark of MultiScope, Inc.
Personal Speech System and TYPE 'N TALK are trademarks of Votrax, Incorporated.
PostScript is a registered trademark of Adobe Systems, Incorporated.
Software Carousel is a trademark of SoftLogic Solutions, Incorporated.
Touch Tone is a trademark of Bell Canada, Canada.
UNIX is a registered trademark of AT&T Bell Laboratories.
VoxBox is trademark of Infovox, Corporation.
Windows is a trademark of Microsoft Corporation.
WorkPerfect is a trademark of WordPerfect Corporation.
Vista is trademark of Telesensory, Incorporated.

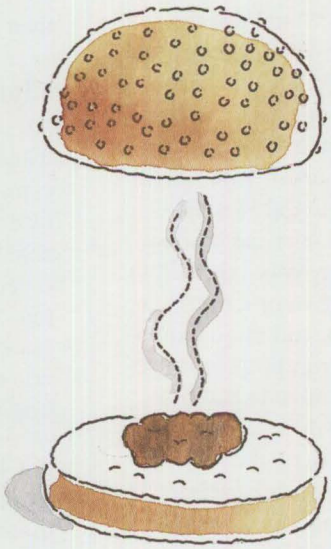
OS/2 End User Advantages

Dave Dill
IBM Corporation
Dallas, Texas

Operating System/2™ (OS/2) is IBM's newest multitasking operating system for use with 80286™-, 80386™-, and 80486™-based personal computers. This article describes the user requirements that led to the development of OS/2 and why a multitasking operating system is so important in today's complex business environment. It also looks at the productivity benefits available to users of OS/2, both from the operating system itself and from new applications developed for the Presentation Manager™ user interface of OS/2.

Several years ago a nationwide hamburger chain commissioned a series of advertisements that featured a little old lady who drove her automobile around to the competitor's hamburger restaurants. After elbowing her way to the counter, she demanded in a loud voice, "Where's the beef?" As sometimes happens with clever advertising campaigns, selected phrases remain and become part of the language long after the commercials have disappeared. "Where's the beef?" became one of those expressions, a phrase indicating a desire for quality in products and services.

It's surprising that in today's world of complex hardware and software, most of us have forgotten the phrase, "Where's the beef?" We seem to accept a torrent of technical details as evidence that this or that product is superior. We seldom ask what impact the product will have on employees, how the product will improve employee productivity, or what kind of training and support the product will require. Simply put, when it comes to personal computer hardware and software, we no longer seem to care where the beef is.



The fact is that the OS/2 operating system can make users more productive. This article highlights, in everyday terms, the gains available to end users from OS/2 Extended Edition. The approach is non-technical, explaining the value of OS/2 without requiring an understanding of bus architecture, DMA addressing or other technical details. I often wonder if the little old lady in the hamburger commercials would have been satisfied with an explanation of the atomic structure of hamburger buns when she asked her famous question.

The Original PC Assumptions

Many people feel that OS/2 is unnecessary, that DOS with its modifications and extensions will suffice as an operating system for many years to come. After all, they reason, DOS has never been more popular than it is today, it continues to grow and be enhanced, and its applications continue to be updated.

To understand why IBM and Microsoft felt it necessary to invest millions of dollars and thousands of hours in the development of a new operating system, it is necessary to understand the original DOS requirements.

Normally, when IBM considers developing a new product, the existing market is surveyed to understand how the proposed product might be used. However, in the case of planning the original IBM Personal Computer, there was no existing market to survey, because there were no existing PC users in the business world. In place of a detailed survey of users, and because an understanding of the intended market is vital to building any product, IBM developed a series of assumptions about how the PC would be used, who would use it, what they would use it for, and where it would be used. There were hundreds of assumptions covering all aspects of PC use, but for our

purposes there were four fundamental assumptions made that resulted in limiting the use of DOS in today's sophisticated business environment.

A Single Machine Running One Program at a Time: First, it was assumed that the original PC would be a single-user machine and that DOS would run only one program at a time. The key phrase here is "one program at a time" – not a program and an emulator, or a program and a utility, or a program and a Terminate-and-Stay-Resident routine, but one program at a time. This assumption, more than any other, allowed DOS to be relatively simple in structure.

With only one program running at a time, there could be, by definition, only one outstanding interrupt at any time and, thus, no requirement for re-entrant code in the operating system or the hardware BIOS. When a program requests service via an interrupt, that program is suspended until the service is completed; in a system with only one program running, servicing of that suspended program's request is the only work to be done. DOS was kept simple and small by enforcing this single outstanding interrupt concept.

Limited Memory and I/O Requirements: Second, these machines were designed for limited memory and I/O requirements. Most PCs had 64 KB of memory; some went as high as 96 KB or even 128 KB. Most applications, however, were built to run in a machine with only 64 KB of memory, including space for DOS, so there was certainly no need for a sophisticated memory management routine. When a program was started, all memory was allocated to that program, and if that program didn't release any memory, there was no room for any other program to run. The I/O requirements ranged from none, which required no diskette storage, to a maximum of two 160 KB, single-sided, diskette drives. With these minimal data storage requirements, DOS disk routines did not need to be sophisticated or optimized. After all, when the maximum user data available was 160 KB, and DOS occupied the other drive, how sophisticated did the disk routines need to be?

A Limited Marketplace: Third, and very important, was the assumption that PCs would be used only by computer-literate people. It was never expected that PCs would come into such general use that they would be given to people who did not fully understand how the hardware or the operating system works. PCs were intended for a narrowly focused market, with only a few hundred thousand PCs projected to be sold over the product's life. No company was expected to have more than five or ten PCs at any one location, and these would be in key areas, run by people specially trained to operate and program them. For these knowledgeable users, the "C" prompt is not only user-friendly, but is the best user interface possible because it is the quickest and easiest way to work with a PC. The "C" prompt is not "unfriendly," it just is not "friendly" to the people who wound up using PCs. If businesses had not insisted on putting PCs on secretaries' desks, on shipping docks, on manufacturing lines and on executives' desks, there would have been no problem with the user interface. But businesses did, and now there is.

No Communication Requirements: Finally, PCs were never intended to be communicating devices. They were designed as stand-alone machines. IBM did offer an asynchronous communication card with the original machines, but communications, especially high-speed communications to a host device with the accompanying data-transfer requirements, were never considered an option. If users needed data from a host device, it was assumed they would use a 3270-type terminal to query the host, then key the returned data into the PC. As a result, DOS never had a requirement to support communications. Therefore, since the PC's first days, users have an emulator, which is an operating system extension, to handle their communication requirements.

The Requirement For OS/2

All these assumptions were invalidated on the first day the PC became available, because the business community began to change the way it did business based upon the productivity potential of the PC. Of all the changes made, three stand out when considering the viability of DOS and the original marketing assumptions.

A Change in the Way Data Is Processed: Corporations quickly discovered that the PC gave the people who need to process data the ability to do their own processing. This made them significantly more productive and responsive to the needs of the business. If the person who needs the data has control over how and when to process the data, the results will be more accurate and timely. The PC freed users from the scheduling problems of host machines. If they needed results on a particular day or data processed in a special way, they had

the power to do it themselves. But this placed a tremendous burden on communications between the PC and the host. In order to do all this customized processing, huge amounts of raw, or unprocessed, data had to flow between the host and thousands of remote PCs.

Businesses also began to change the type of processing that they did with their PCs. The first PCs ran small record-keeping applications, such as accounts payable, accounts receivable and payroll. Programs were easy to use, and the amount of data processed was small. Today, businesses are turning toward forecasting what will happen rather than recording what did happen. For instance, businesses today want to know what will happen to their market share if they raise their advertising budget 10 percent and their competition doesn't follow suit. To do this, businesses began buying or building complex economic models to help forecast the future of their products and businesses. But this has caused a gigantic growth in the requirements for data. These models require vast amounts of information to accurately forecast the future, and the data required is no longer simple and straightforward.

Most host machines are consumed with the normal daily work and cannot always be as responsive as users might like. Rather than try to force their host machines to be more responsive, businesses have discovered that it is easier and cheaper to give personal computers to the people needing the data processed and allow them to set the requirements and do the processing. The problem with this is that it became no longer possible to forecast accurately when users will need access to data or what data they will need. Data has now become an "on-demand" item. When the user needs data, the host must supply it.

Users have also changed the type of data they process. The original PCs were designed for text data, and DOS was built to display and process text data. In fact, IBM did not offer a graphics monitor when the PCs were first introduced. Today, however, graphics data is the norm, rather than the exception, with business charting, desktop publishing, computer aided design (CAD) and the like. These exciting graphics applications place tremendous demand on PC memory. To hold a single page of data scanned at 400 dots per inch and in 16 colors or shades of gray can consume from 8 to 12 million bytes of memory, depending on the viewing options. DOS was not designed to handle this much data – it lacks the internal routines to move massive amounts of screen data and also lacks the large memory support needed to handle this much data.

There is a real need to connect every user to every source of data.

Finally, corporations have begun to transform their hosts into giant "data utilities." Like the utilities that supply electricity when the user throws the switch, hosts are now expected to supply data when the user throws the data-transfer switch. This has placed a tremendous strain on communication networks. Because hosts never know when or how much communication capability they will need, they often must plan for the maximum.

A Growing Emphasis On Communications: Businesses have come to realize that, after the actual products they build or sell, data is their most important asset and access to the data, wherever it is, is absolutely critical for those with authorization to use it.

Data is now a "company-wide" resource that must be accessible by everyone with a "need to know," regardless of where he or she is located. The idea that developed several years ago with the introduction of local area networks, that data is unique and could be kept entirely within a department, has been proved wrong. A company cannot predict who will need what data or where it will be needed. As a result, there is a real need to connect every user to every source of data. This amounts to connecting every user to every other user and to every host machine. Needless to say, this has caused tremendous demands on communications.

Furthermore, because computers in use today were made by several manufacturers, the standard IBM SNA communication network may not be sufficient. Users often need to run a variety of communication protocols concurrently. Ethernet™, TCP/IP, X.25, and seven-bit asynchronous are just a few of the ways companies connect their own computers and connect to computers in other companies. For DOS users, this is a nightmare. Most DOS emulators do not support multiple concurrent communication disciplines. As a result, DOS users often require multiple emulators – none of which run concurrently, each with different keyboards, different interfaces and

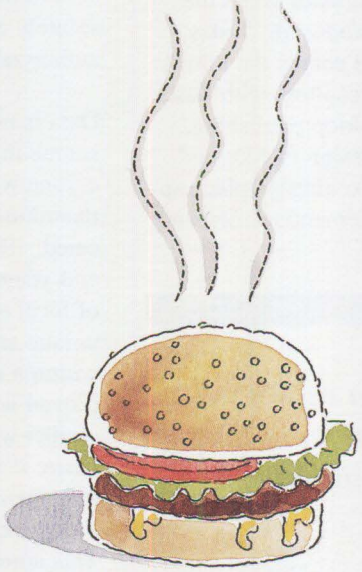
different data-transfer protocols – to perform their daily work.

The Proliferation Of Workstations: One of the assumptions about the original PC and DOS design was that the PC would only be used by computer-literate people who knew how the hardware and software worked. These users were considered sophisticated enough to support themselves; training and after-installation support were not required for them. After all, what support is needed by users who write their own applications, may have modified the DOS operating system, and can strip, repair and reassemble a broken PC in about an hour? These were the intended users of PCs.

This, however, is not the way the market has developed. PCs and their follow-on Personal System/2® (PS/2) machines have been moved into every corner of most businesses. Employees of every skill level within most companies now have computers available and are expected to mold at least part of their jobs around their computers. This presents companies with real problems, because this new breed of users requires extensive training and after-installation support. They do not have the expertise to support themselves. Many less sophisticated users have been given so much automation that they require support to perform anything but the most standard of functions. This has led to huge increases in support staffs and a large increase in related costs.

Productivity Gains From OS/2

How can business users running under DOS solve all these problems? Some relief is offered by OS/2 itself, by what can be called the OS/2 environment, while additional relief is offered by new applications written to use the OS/2 and Presentation Manager (PM) interfaces. Each of these areas offers its own productivity enhancements for users.



The productivity gains available from the OS/2 environment (separate from gains available from the OS/2 and PM interfaces) are due to two things: the ability to address up to 16 MB of memory, and the ability to have more than one task running at a time. This does not mean there are only two productivity gains for users. To the contrary, there are many different ways to increase productivity under OS/2, but all of the benefits available from the OS/2 environment relate in some way to these two features.

Rapid Access To Required Applications: One way a user can benefit from large memory addressing is to load more than one application at a time. With more than one application running, the applications appear interruptible, and users can choose which applications get their attention and when to move from one application to another. This is an important feature of the OS/2 operating system, because it lets the user select what to run and when to run it.

DOS certainly allows the user to select what to run, but once an application is started, the computer is unavailable for the duration of the application. Anyone who has ever formatted a diskette knows that while the format program is running, the PC cannot do additional work. This can be extremely frustrating to users. Copying large amounts of data to diskettes presents the same problem. Once the copy is started, the user must wait for the copy to complete before anything else can be run, regardless of the user's requirements. This unproductive time is the main reason most users don't back up their hard drives – they cannot afford the 45 to 90 minutes the backup requires, time their PCs are unavailable for other things.

It is not enough to talk about running more than one application at a time, without also mentioning the background processing capability of OS/2. Background processing is the ability to run applications out of the user's sight. This gives the user the benefit of the application without having to deal with the complexities of the application or the problems of running multiple concurrent applications. The user can select applications from a menu and ignore the required support programs running in the background. Imagine transferring data without having to deal with the transfer program; browsing a bulletin board without having to deal with the asynchronous communication task; or creating a report from a database without having to deal with the data

extraction program. In contrast, although DOS can jump from one application to another, the DOS user is aware of all of the running applications and must deal with each of them at some point in their execution.

Individual Applications Can Use Memory More Efficiently: A second way productivity can benefit from large memory addressing is the ability of individual applications to acquire large amounts of memory for their own use. Significant gains in performance can be achieved by applications when a large amount of the data they use can be held in memory at one time. Sorting, extracting, analyzing, drawing and manipulating data are significantly faster when the application does not have to swap large amounts of the data continuously to disk because of limited amounts of available memory. Applications heavily dependent upon data can run up to 50 percent faster under OS/2, which has the ability to address more memory and thus hold more data.

If you have ever run into the spreadsheet message "no more memory," or tried to spell-check a large document when only a few pages could fit into memory at one time, or tried to manipulate a scanned image under DOS when only a few hundred thousand bytes of memory were available to hold the image, you know the frustration users feel with the DOS limited memory environment. There is just not enough room under DOS for today's sophisticated data applications.

Today, with the DOS operating system, users are trying to cope with limited memory by using one of a number of memory-management routines. The problem is these routines often require specially written applications, special hardware, or unique drivers that may not coexist with other hardware or drivers. In short, under DOS, managing the amounts of memory required by today's applications is difficult and frustrating at best.

Reduced Execution Time Through Better Storage Management: The third benefit from OS/2's ability to address up to 16 MB of memory is the reduced complexity of applications. This reduced complexity means faster and easier development and quicker execution. Applications running under OS/2 can have a single large main module that eliminates the need to manage a complex overlay structure. With OS/2, the operating system handles the allocation of free memory and the transfer of unused program code to disk when memory is overcommitted. Applications can be optimized for faster execution when the need to manage a complex overlay structure is removed.

Everyone has at some time struggled under DOS with slow-running applications that had to manage complex structures and large data areas, and with programs that had to move overlays into and out of memory trying to keep up with the user. Everyone has experienced the delay between the time a PF key was pressed and the time the function was ready for use. These delays come from the need to compress millions of bytes of logic into relatively small overlays. Not only is the overall program complex with its need to manage overlays and pass data, but each overlay itself becomes more complex, because it must manage the interaction with all other overlays.

Overlap Long-Running Tasks Or Subtasks: In addition to large memory addressing, productivity gains within the OS/2 environment come from multitasking, which is the ability to run more than one application at a time, and multithreading, which is the ability of a single application to have more than one logic path executing at a time. However, the productivity gains from multitasking and multithreading require more than just running two applications at a time. The user must take care to find the right mix of jobs so that those running in the background can execute with little or no user involvement. It would do little good to run multiple spreadsheets, for example, because the spreadsheets running in the background would receive little user attention and produce few, if any, results.

Communications and host data transfer are prime examples of good background applications. They are usually long-running functions, often they can be automated, they usually do not require user input to produce the desired results, and they often appear complex to users. Consider any application that uses data from a host computer. Data transfer becomes part of that application, not because the application requires it, but because of the need to position the data at the PC. As a result, the user becomes involved in this process which is not easy to understand and often time-consuming. For some applications, the time to transfer the data exceeds the time to run the application. For those applications, wouldn't it be convenient to transfer the data automatically as a by-product of turning on the PC? With OS/2, this can be done easily by starting a background task that transfers the data from the host prior to the time the application is to run, making the data available

without user involvement, and saving the transfer time.

But many businesses need more than the traditional host SNA link. They need local area network links, asynchronous communication links, links to public networks, and links to non-IBM computers, all in addition to their traditional IBM host links. If a business is involved in a "just-in-time" inventory system with its suppliers, it may need links to its suppliers' computers because tomorrow's raw materials are on the suppliers' shop floor today. To understand the work in progress, the business needs to know what its suppliers will deliver, and that takes automated links to be productive. DOS emulators do not support these concurrent multiprotocol links, but OS/2 does, and they all can run in the background if required. Three or four communication links running in the background that supply data automatically to a production application running in the foreground can be a very powerful productivity tool.

Often users will dispute the productivity of multitasking because their machines are dedicated to running one application. They may, in fact, believe that multitasking will improve productivity, but feel it doesn't apply to their particular circumstances. After all, they reason, how productive can multitasking be when they run the same text processing program all day long? But overlapping the same application within the same machine can be very productive. DOS cannot do this overlapping, but OS/2 can.

Think about most PCs with dedicated applications. Regardless of the application, there are natural wait-periods in every application, periods when the machine is busy and the operator is left with nothing to do. Text processing is an excellent example of this. Sooner or later, the need to print high-quality documents will bring the PC to a stop while the documents print. Even with background printing, there are still occasions when managing data and printing will cause the PC to wait, making the operator unproductive. Wouldn't it be more productive if, after starting one of these long-printing applications, the operator moved to another OS/2 session, re-executed the same text application, and began to work with another document? The operator might well have several sessions running, each executing the same application but working on different documents. By allowing the operator to keep working during those wait-periods, OS/2 has increased the operator's productivity significantly.

Run Multiple Parts Of A Job Simultaneously: Ask most users what they do with their personal computers and they will tell you they run something "big" like general ledger, accounts receivable, sales analysis or desktop publishing. In other words, users tend to think of what they do as one large application running for a long time; productivity gains are hard to extract from long-running tasks. The fact is, however, they really are running many small tasks that, when put together, make up the overall application. Within any application, data input, file updating, sorting, printing, and other functions are usually found. All of these steps make up the big application. But users do not see the individual parts, because under DOS it makes no difference in overall time how they arrange the individual tasks. The sum of the individual tasks is equal to the time of the application.

With OS/2, however, rearranging the individual tasks often leads to greater productivity. Overlapping the longer-running tasks with one or more shorter tasks can effectively eliminate the time of the shorter tasks. For example, when an application contains a relatively long-running data search routine, start it first so it can be overlapped with shorter report preparation activities. If data needs to be downloaded from a host computer, do that while preparing your final documentation. If you can overlap a five-minute data transfer during a 30-minute application, you have increased your productivity by 17 percent.

Applications can automatically start overlapped subtasks via multithreading to achieve significant productivity gains. When an individual step will take a relatively long time to complete, the application can automatically start a second thread that will run at its own pace while control is returned to the user. The user can then shift to another part of the application or another application to keep working while the secondary thread completes. These productivity gains can occur automatically without the user having to rearrange the schedule of execution.

Probably the best example of task overlap is a background communication task automatically collecting data and feeding it to a foreground task. The user may never know that the background task is running, yet the data automatically appears within the foreground application. Imagine sitting in front of a spreadsheet and seeing the data you need appear within the spreadsheet automatically, continually refreshing itself to show the most current data at all times. The user needs to know nothing about communications, data transfer, cut-and-paste or

clipboards to achieve gains in productivity from such an application.

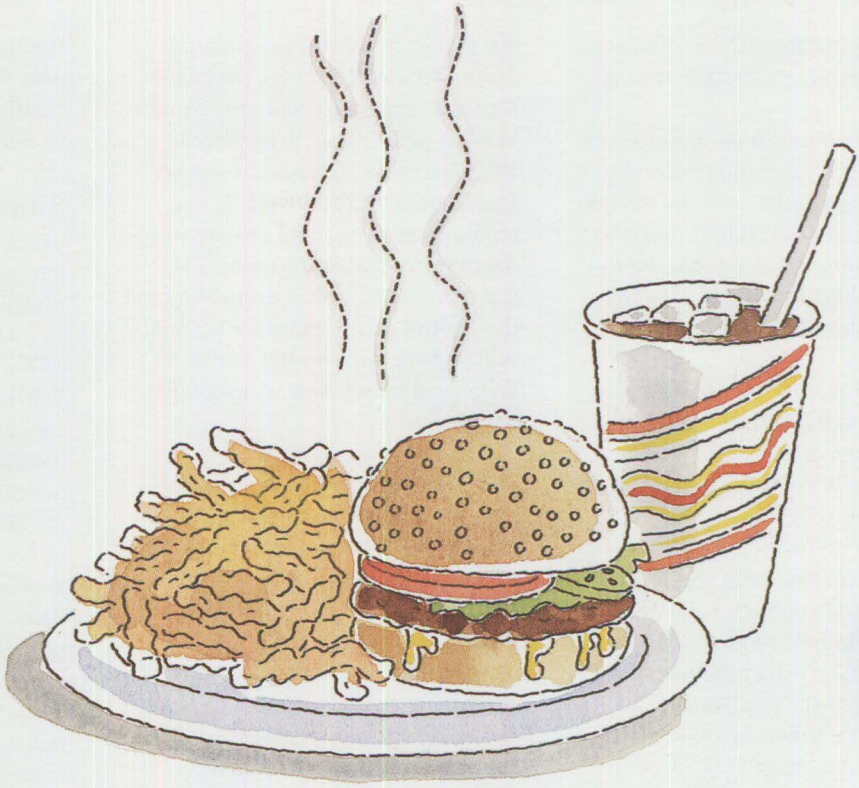
Overlap Key Applications With Less Important Applications: The key to real productivity gains is keeping the users focused on what is important to them. Most users get paid to do select things; throwing lots of "windows" at users will not make them more productive if the important functions get lost in the crowd.

If the user's job is to manage a company asset with a spreadsheet, then the spreadsheet should be kept centered on the screen at all times. For this user, everything else is secondary. OS/2 can make the user more productive by making the spreadsheet more productive, but the focus must remain on the spreadsheet. All users – from secretaries to loading-dock personnel to order-entry clerks – need to keep focused on the applications that are key to them.

This presents an opportunity: the background execution and multithreading capabilities of OS/2 keep users focused on what is important to them. Users are relieved of the responsibility of running and controlling the lesser tasks and can concentrate on what they get paid to do.

Productivity Gains From OS/2 and PM Applications

Productivity gains come not only from the OS/2 environment, but also from applications written especially for the Presentation Manager. This is where the gains become more unique to each user and the applications produce real magic. The magic comes from hiding the mundane and complex functions in the background so users see only the



data they need in the applications they are running. The productivity gains available when writing Presentation Manager applications come from the increased sophistication and integration of the applications.

More Sophisticated And Integrated Applications: There are hundreds of new and exciting capabilities built into OS/2, the Communications Manager, Database Manager and Presentation Manager. We have already touched on a few. But the availability of multithreading, data piping, dynamic data exchange, and named pipes means new and better ways to write applications. Users should look for new ways to do business, rather than just think of running the same old DOS applications under OS/2.

Sophisticated applications executing multiple logical paths simultaneously within the same application hold exciting potential for programs that border on artificial intelligence. Applications can now completely modify their execution patterns based upon received data. The user no longer needs to be knowledgeable in all aspects of an application, because the application can automatically compensate for a changing environment by executing alternate logical paths. The application can now make many of the decisions that formerly were made by the user. Applications can be in communication with multiple host locations and multiple vendors to make intelligent decisions about inventory and product movements, while the user simply enters the order into the personal computer.

Easier To Use

With all the sophistication possible with OS/2 applications, it is important to note that for most users, the Presentation Manager environment

is easier to use than DOS. The standardized graphics interface means:

- All applications look and operate the same. Navigating around one application is the same as navigating around all of them. Support and training are reduced. New applications look just like existing applications.

- All possible user actions are displayed on the screen in easy-to-use menus and dialog boxes. The user is not left to guess what to do next.

- A sophisticated help feature can be made available for every action the user is required to take, and all help works the same, regardless of the application, because it is now a function of the operating system.

- Background processing can remove the complex and difficult functions from user control requiring less application-specific knowledge by the user.

Easier And Faster Application Development

Don't overlook the cost and time involved in developing applications under OS/2 and the Presentation Manager. The cost of application development should include design, development, and writing as well as the maintenance. While it may be cheaper today to develop an application under DOS, considering that 80 percent of an application's cost involves ongoing maintenance, the OS/2 application will almost always be cheaper over the life of an application.

No matter how well programmers write DOS applications, they will face maintenance problems. A new version of DOS, a change in the DOS interrupt structure, new and larger emulators, new shells and menuing systems, and growth in the data or movement of the data – these are some of the events beyond the control of the programmer that will surface early in the life of a DOS application and will require maintenance.

OS/2 avoids many of these problems by using architected interfaces that are serviced through loose calls rather than a tight register bind. Because OS/2 provides significantly more services than DOS, programmers need not write application code to perform what should be an operating system function; and the less code, the less maintenance. The use of Dynamic Link libraries means one executing routine can be shared by all programs that need it. There is no more need for copy books and no more need to recompile and link the hundreds of applications that use the same book. Instead, write it once; compile it once; and maintain it once. That's true productivity for programmers.

The missing pieces for meaningful OS/2 and PM application development are now beginning to arrive. OS/2 Programming Tools and Information Version 1.2 contains new high-level language support that allows easier control of the PM environment so that programmers not familiar with C or Macro Assembler can code in COBOL or FORTRAN. A sophisticated procedure language will be added to OS/2 with Extended Edition version 1.2. And new PM coding tools and application development systems are available that allow programmers to build applications without the need to master PM coding techniques.

Summary

These are not the only productivity gains available to users running under OS/2 and the Presentation Manager. These are not even most of the potential gains available. There are almost as many productivity gains available from OS/2 and the Presentation Manager as there are different applications. This article simply suggests some of the more common productivity gains available to users. Some of these gains may apply in a particular situation and some may not; none are absolute. But a look at existing applications should show the potential for enormous productivity gains from the OS/2 operating system, the Presentation Manager, the Communication Manager™ and the Database Manager.

ABOUT THE AUTHOR

Dave Dill is a senior market support representative in the Personal Systems Technical Support Center in Dallas, Texas, providing technical support to IBMers and customers for OS/2 Standard Edition and the Presentation Manager. Dave joined IBM in 1968 and worked in the field as a finance industry systems engineer. He then moved to Finance Industry Development where he helped develop several finance industry and PC-based applications. For the last five years, he has provided market support for PCs and for the DOS and OS/2 operating systems.

What's New in OS/2 Standard Edition Version 1.2?

Craig Chambers
IBM Corporation
Dallas, Texas

This is a summary of the new features in Operating System/2 (OS/2) Standard Edition (SE) Version 1.2.

Version 1.2 of OS/2 SE continues the improvements begun with the inclusion of the Presentation Manager in OS/2 SE Version 1.1. These improvements give version 1.2 a completely new look and make it considerably easier to use than previous versions. Improvements include visual items such as 3-D push buttons, a new Desktop Manager and online reference system, a buffer for commands, and a greatly improved File Manager.

The new online command reference is an installation option. If users have questions about how to use a command, its format, or the meaning of its parameters, entering **HELP** will prompt the system to display a complete description of that command.

With all of these changes and enhancements, OS/2 Version 1.2 now requires 12 MB of disk space if all features are installed.

Desktop Manager

The Task Manager and Start Programs windows have been replaced with a new window called the Desktop Manager.

Programs are arranged into groups which are listed in the Desktop Manager window. These groups can either be opened automatically when the system is started, or can be started by the user. Within a group, programs can be started just as they were in version 1.1 by clicking on them with the mouse pointer or highlighting them by moving the action bar and hitting the enter key.

One interesting new feature is the ability to start DOS programs in the DOS compatibility session from the program groups under the Desktop Manager or the File Manager.

The group windows can be arranged anywhere on the screen. A new option of the Desktop Manager allows the user to save the screen arrangement. When the system is restarted, the screen is arranged just as it was when it was saved.

Programs can be moved between the various groups by simply clicking on them with the mouse pointer and dragging them to a different group. The order of the programs with the list in a group can be changed the same way; click on the program and drag it to the location in the list where you want it to appear.

Like in the previous release of OS/2, pressing the Ctrl+Esc keys brings up a window with a list of the currently active tasks. In version 1.2, the programs in this list are ordered, with the programs used most recently shown at the top of the list. This makes it easier to switch between programs.

DOS Compatibility

The DOS compatibility session has been improved in version 1.2. Programs can now be started from the groups of programs and from the File Manager; the DOS 4.00 screen modes with more than 25 lines are supported; and the amount of memory required by the system in the lower 640 KB has been reduced, allowing larger DOS programs to execute.

A major enhancement is the new dual boot capability. It is now possible to have both DOS and OS/2 installed on the same fixed-disk drive.

A major enhancement is the new dual boot capability.

Begin with DOS (version 3.20 or later) on the C: drive. All DOS files must be in a separate subdirectory. When OS/2 Version 1.2 is installed, it detects the presence of the DOS system and implements the dual boot feature, which saves the boot record and the DOS CONFIG.SYS and AUTOEXEC.BAT files before OS/2 is installed.

When changing operating systems, the BOOT command specifies /OS2 or /DOS depending on which system are being activated. The boot record, CONFIG.SYS, and AUTOEXEC.BAT, which were previously saved, are copied to the root directory, and the other versions of OS/2 are saved. The system then restarts itself. If this is done often, the BOOT command can be added to one of the program groups in the Desktop Manager.

Remember that the dual boot feature cannot be used if the High Performance File System (HPFS) is used on drive C.

File Manager

The File Manager, which is contained in the main program group, has been substantially improved. It is used to manage files and directories and to execute programs.

When it is started, the File Manager displays a directory map of the current drive and a pictorial mapping of the logical drives that are attached to the system. A copy of the File Manager can be started for each drive in the system.

There is a small icon of a file folder next to each directory name. If there is a plus or minus sign in this icon, then this directory contains subdirectories.

If the location of a file is unknown, select the File item in the action bar, then select Search in the File pull-down option box that is displayed. The Search window will be displayed where the name of the file is entered. You can also specify that the search can be directed to occur on a different drive or directory, if necessary.

Files and directories can be moved or copied by either dragging them with the mouse or by using the File pull-down with either the mouse or keyboard. Dragging the files or directories with the mouse is quicker, but the file or directory can be renamed during the operation with the pull-downs.

If files are dragged between drives, the files are copied. If they are dragged between directories on the same drive, they are moved. To

copy files to another directory on the same drive, hold the Ctrl key and press mouse button 2 while dragging the files.

There are several new ways to start a program in version 1.2. With a directory window displaying your data files and another open showing your programs, simply click on the data file, and drag it to the program file; the program will be started using that data file.

If the program title is displayed in one of the Desktop Manager windows, it can be started by dragging the data files from the File Manager to its title in the Program Window Group.

Programs can be associated with their data files. When this is done, double-clicking on the data file will start the program. If the data file has been associated with more than one program, a list of the programs is displayed and the program to execute must be selected.

As in the previous release, a program can be started by clicking on it or selecting the run option from the File pull-down.

Any installed font can be used with the File Manager in version 1.2. It is often easier to see the cursor if you select a larger fixed-space font, such as one of the Courier fonts. This will not, however, affect the font used in the various dialog boxes.

Printing Files

A file can be printed in much the same way that a program is executed. Open a directory window that contains the file, select the file, and drag it to the Print Manager icon. The system will prompt to enter the queue or printer and the file will be printed. A group of files can be printed the same way – just select a group of files and drag them to the Print Manager.

High Performance File System

The High Performance File System (HPFS) is an alternate way to format fixed-disk drives with OS/2 Version 1.2. It can be installed during system installation or later with the FORMAT command. It supports a file with either the standard DOS file names or long file names, which can be up to 255 characters in length. The HPFS is less performance-sensitive as files and disk drives increase in size.

The High Performance File System supports up to 16 partitions on a drive.

The HPFS supports up to 16 partitions on a drive. The partitions and files can be as large as 2 gigabytes. The HPFS is compatible with the File Allocation Table (FAT) system at the API level. This allows programs written for OS/2 Versions 1.0 and 1.1 as well as DOS programs running in the DOS compatiblity session to access HPFS files.

No earlier version of OS/2 nor any version of DOS recognizes a drive formatted as an HPFS drive. Therefore, booting with any system other than OS/2 Version 1.2 will not give access to either the HPFS drive or any logical drives that are located after it on the same physical disk drive.

For example, assume there is a system with a 120 MB fixed disk installed, partitioned as drives C:, D:, E:, and F:. If drive C: is formatted as an HPFS drive and OS/2 Version 1.2 is installed, when DOS is booted none of those drives can be accessed.

Extended Attributes

The version 1.2 file system supports extended attributes. These are user-defined data for a file that can be up to 64 KB in size. If files with extended attributes are moved or copied using DOS or a version of OS/2 other than 1.2, the attributes will be lost.

Long File Names

The HPFS also supports files having names as long as 255 characters. If these files are copied to a FAT-based drive, the names must be shortened. The original, long name is saved as an extended attribute of the file so that, when the file is copied back to an HPFS drive, the long name can reused.

If the COPY or XCOPY command is used to copy the file to a FAT-based device, and there is a file with the short name already on that device, it is replaced. If the copy is done with the File Manager, the user will be asked to change the short name.

System Editor

The system editor has been enhanced to run in a Presentation Manager window. It now conforms to the Systems Application Architecture™(SAA™) Common User Access (CUA) definition.

Help Manager

The Help Manager is formally called the Information Presentation Facility or IPF. It allows help panels to be developed independently of the flow and logic of an application program. A compiler included in the toolkit allows non-programmers to create the help panels, which are then combined with the application code. The IPF can also provide a link between programs for tutorial functions. The IPF can be used by both Presentation Manager and Dialog Manager applications. The IPF complies with the SAA CUA requirements.

Miscellaneous Improvements

Three new utilities work with picture files. PICPRINT allows the user to print metafiles and Picture Interchange Format (PIF) files. PICSHOW displays picture files on the screen. PICICHG converts PIF files into metafiles.

The Presentation Manager now has API calls that provide support for the help system and support for both FORTRAN and COBOL.

New device drivers are included with the system. A PostScript™ driver is provided for the IBM Page Printer II models 4216-030 and -031, and other PostScript printers. A driver is also provided for the Epson™ FS80 (9-wire) and LQ1500 (24-wire) printers.

FDISK now runs under the Presentation Manager. It displays all information about a fixed disk on one screen, making it much easier to use.

A compiler included in the toolkit allows non-programmers to create the help panels.

The system can now support 64,000 file handles, and a single process can access 32,000 file handles.

The FORMAT command has been changed to make it easier to remember its parameters. For example, to format a 720 KB diskette in a 1.44 MB drive, the command is "FORMAT A: /F:720KB". Of course, if the user still forgets the syntax, entering the command "HELP FORMAT" on the command line will activate the online command reference and display a description of the command and its parameters.

The quick reference card has a complete mapping of the Desktop Manager functions. The reverse side describes how the keys are used by each of these function.

ABOUT THE AUTHOR

W. Craig Chambers is a senior market support representative in the Desktop Systems Technical Support Center in Dallas. He joined IBM as a systems engineer in Pittsburgh in 1969 after receiving B.S.M.E. and M.S.M.E degrees from Purdue University. His previous assignments have included lead systems engineer at several large MVS/JES2/JES3 accounts, technical support for communications products such as the 3270 system, the 3270-PC, the Workstation Program, and ECF. He is presently supporting OS/2 Standard Edition. He has been a presenter for IBM's television broadcasts to its customers and has published several IBM technical bulletins.

An Application Developer's View of OS/2

Peter Kron
Aldus™ Corporation
Seattle, Washington

In this article Aldus shares their experiences in developing PageMaker® for OS/2 Presentation Manager. The article points out the OS/2 features the developers used to their advantage, and potential pitfalls that are not necessarily endemic to the operating system itself, but which can trip up an OS/2 developer who is accustomed to developing applications for Windows™ or the Macintosh®.

Aldus Corporation began shipping PageMaker for OS/2 Presentation Manager in September 1989, completing a process that had begun more than two years before, when the first OS/2 developer conferences were held. During this period, Aldus engineers worked closely with Microsoft and had access to numerous pre-release software development kits (SDKs).

Concurrent with this development effort, major new releases of the Macintosh and Windows versions of PageMaker were developed and released, providing a unique vantage point for assessing OS/2 as a development environment.

Background

The process of creating PageMaker for OS/2 benefited greatly from our experiences with the Windows and Macintosh versions of PageMaker. The existence of two versions had forced PageMaker to straddle the host application program interface (API) fence successfully for almost three years. A key element of that success has been the core/edge code concept – between 50 and 75 percent of PageMaker is shared by those two platforms. (PageMaker is written almost entirely in C.)

Because OS/2 and Windows share common hardware and software ancestry, OS/2 became more of a subdivision of the Windows edge than an entirely new, third edge. Our experience with Windows was certainly a great advantage in writing for OS/2 Presentation Manager (although in some instances we assumed similarities that were not true).

Our goal was to use as much core and Windows edge code as possible, preserving the "look and feel" of PageMaker and file compatibility. But we also wanted to take advantage of inherent features of OS/2 to improve productivity wherever possible.

OS/2 Presentation Manager

OS/2 Presentation Manager is really a mixture of two levels of API: Window Management and Graphics Imaging. They reflect dual parentage by Microsoft and IBM. Window Management is very similar to the Windows API, while Graphics Imaging bears strong resemblance to IBM mainframe graphics models.

Similarly, our code base has a window management edge (front end) and an imaging edge (back end) (Figure 1). The various core functions, such as data storage and text composition, fit generally in the middle. Because we had an established Windows edge for the front-end, we were able to modify it to handle the OS/2 API. One of the most common API differences is the addition of an "anchor block" handle to Presentation Manager calls; the Windows equivalents have no such argument. Changes of this nature could be dealt with by compiler macros. Other differences are limited to argument types and constants, and could be resolved by conditional datatype definitions.

Nonetheless, there are subtle differences that can trip up a developer. One of the most pervasive is the switch from a top-left-based origin in Windows to a bottom-left origin in Presentation Manager (Figure 2). This bias affects relative placement of child windows (such as rulers), common drawing code, and genera-

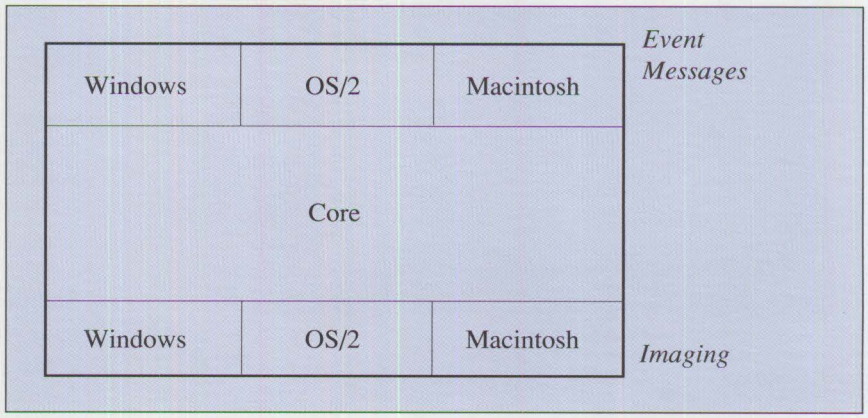


Figure 1. Code Structure

tion of bitmap images. The computation needed to convert from one form to the other is dependent upon current window height. While this computation is not difficult, it affects a surprising amount of code and will be the source of many bugs if not anticipated adequately.

For example, Aldus had isolated many differences between Windows and Macintosh in their respective "edges." If core code contained assumptions specific to one or the other, the problem soon became apparent during testing. But because both followed a top-left imaging model, dependencies on this orientation went unnoticed in the core code and subsequently caused various problems in porting to OS/2.

A second subtlety is that some of the windowing messages differ only slightly from each other, either in interpretation, parameters, or order of generation. These slight differences are more bothersome than outright new messages, which can be conditionally handled. For example, subtle changes in the interpretation of keystroke messages from Windows to OS/2 changed the basic assumptions of the routines – again requiring changes to a large amount of code spread throughout the program. The Presentation Manager scheme is in many respects an improvement, but just the difference can be bothersome. There is also a particular problem when dealing with stable, familiar code that has been ported to a new environment; sometimes the engineer is still thinking in terms of the previous environment, and subtle differences can be a real blind spot.

Imaging Models

The imaging model of Presentation Manager differs radically from that of Windows, causing the macros to be ineffective as a porting tool. It was possible to use conditional code for the imaging because the imaging code was already fairly well modularized at the back end of the application. Certain edge-specific modules required complete rewriting, however. (Conditional code was used primarily in modules that were already Windows-specific.) In a few key instances, we were able to take advantage of the opportunity to improve our core-edge boundary by defining new platform-independent service routines and bundling the OS/2-specific code there. Our main guideline was readability: if a specific section could be written as conditional code without too much loss of readability, we didn't feel it was necessary to define new service routines at that time.

A more difficult imaging problem was resolving the pixel-specific idiosyncrasies of the two graphics models. While both models are based on stroking path lines of a defined width, the two imaging engines do not always end up with the same pixels being affected. It took some work to get the algorithms recoded so that PageMaker graphics and text would butt up properly under Presentation Manager.

Displaying text is an extreme example of this problem, due to the complexity of the metrics associated with text fonts. In addition, because the representation of font names and attributes is significantly different under the Macintosh, Windows and Presentation Manager environments, font code had to be rewritten from scratch. Font technology in general has been a hot topic recently, and there will undoubtedly be more changes in these areas. The good news from all of this is that font issues are being studied more seriously than ever, and solutions are being found.

Solutions to all of these problems have to address four basic device types: 1:1 screen aspect ratios (such as VGA), other screen aspect ratios (such as EGA), PostScript printers, and raster printers. No im-

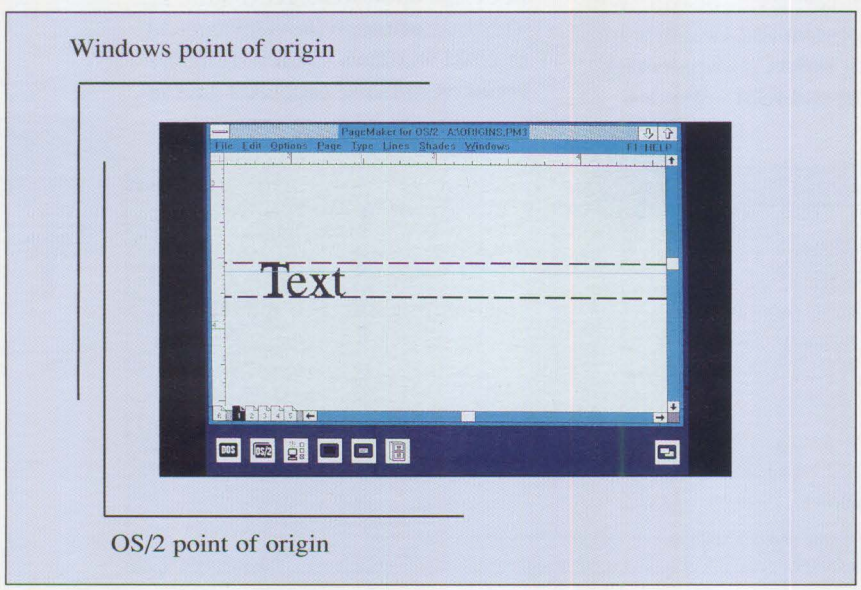


Figure 2. Points of Origin

aging model is perfectly device-independent, so the way in which the application uses the engine API can be very sensitive to the particular constraints imposed by both application requirements and the capabilities of the device (such as resolution and resident fonts).

There are several areas of technical difference in the two APIs. Presentation Manager's Graphics Programming Interface (GPI) has a much richer set of capabilities than Windows, including new curve primitives, pick-correlation of mouse actions, line attributes versus pen objects, and redraw of complex objects by the engine itself. The GPI also provides a (somewhat daunting) hierarchy of viewing transforms. For an application, such as PageMaker, which is to be available on other platforms without these features, it is a difficult task to take full advantage of these features while hiding the platform-specific implementations. This is an area we are still refining.

A final note of caution: the Windows Device Context is split into two handles under OS/2 Presentation Manager. Some analogs to the Windows API require the Presentation Manager Device Context, while others require the Presentation Space. Presentation Manager can also associate multiple Presentation Spaces with a Device Context, and fonts are associated with the Presentation Space. The result can be changes to some function-interface definitions to provide each function with the proper handle.

Developing Applications in OS/2

The four main features of OS/2 that affect development decisions are more memory, enriched graphics, multitasking, and multithreading. PageMaker does not currently utilize many of the graphic extensions provided by the GPI, but I will discuss what each of the other features brings to PageMaker for OS/2.

PageMaker for OS/2 uses multithreading, which is quite effective in improving the responsiveness of the application.

The larger address space reduces code swapping in PageMaker, resulting in an immediate boost in raw performance. This is about as close to a free lunch as can be found in the software business. In the design phase, we discussed various changes to our caching algorithms, based on the assumption of having more memory, but we implemented only some basic caching of bitmaps and font metrics. There is still lots of room for enhancements in this area – such as in tuning file management – but we didn't feel that the gains would be significant enough to mandate their inclusion in this revision.

PageMaker for OS/2 opens multiple publications, a feature, which is unique to this platform. Cutting and pasting between publications is consequently simpler for the user than in Windows. This feature was implemented by creating a separate process for each publication window. Multitasking – using separate processes – facilitates background operations such as flowing text and printing. While this feature does not alter the raw performance of the application, it has a dramatic effect on the productivity of users, by freeing them to work in other areas while the operation completes.

Achieving the same level of independent activity by implementing threads within one process would have required considerable synchronization of access to data structures. Using multiple processes is much more straightforward. There is some overhead in this approach, because cursor and other resources must be loaded separately; but because the code itself is all shared, using separate processes was our preferred choice. PageMaker creates shared memory – another OS/2 feature – which is used by the different processes to coordinate this activity.

PageMaker for OS/2 does use multithreading as well, which is quite effective in improving the responsiveness of the application. Three threads are always active – a message thread, a service thread, and a paint thread (see Figure 3). A separate message thread is needed to ensure that all input messages are handled quickly, because this is essential to overall Presentation Manager performance. Calling a subroutine to print a page while processing the PRINT command message would prevent Presentation Manager from dispatching any further messages to any applications, and so the hourglass would linger. Presentation Manager guidelines state that no message should require more than one-tenth of a second processing. To meet this guideline, long user operations in PageMaker – printing, importing data, and flowing text – are performed by the service thread. The message thread waits for another message, thus al-

lowing Presentation Manager to activate other applications. Program initialization is also done largely by the service thread, absorbing the idle time while the user invokes the NEW or OPEN dialog.

This synchronization is complicated, because the user may continue typing or mouse activity, which resumes the message thread while the service thread is still active. In this instance, PageMaker for OS/2 Presentation Manager filters these messages and only accepts certain basic ones, such as RESIZE and MINIMIZE. A private message is posted from the service thread to indicate completion of its task. Since user activity in this process is restricted, we give the user feedback by disabling menu items and displaying a "busy" cursor, which is distinct from the hourglass. This cursor will change when moved to other windows (because the user is free to switch to other applications), but the hourglass will not. In future versions we hope to handle a greater variety of messages while service threads are active.

Screen redraw is performed by a separate paint thread in PageMaker. There are two reasons for this: first, since PageMaker does not limit the number of objects appearing on a page, redraw can easily exceed the guideline of one-tenth of a second to process events. More importantly, using a separate thread allows a greater responsiveness to the user by allowing the redraw to be aborted. If a newly opened page is at the wrong view, the resize command can take effect immediately rather than completely redrawing the page, resizing and completely redrawing again. Dynamic scrolling – redrawing the screen as the user drags the scroll thumb – also occurs, because monitoring the scroll bar is handled by the message thread. (The rulers are also drawn by the message thread, because that is fast and gives immediate positional feedback to the user. The page redraw constantly tries to catch up.) Users greatly appreciate this new responsiveness.

While dynamic redraw can be implemented without multithreading, it places a much greater burden on the application developer to poll for possible messages at various points. Multithreading allows the concurrent activities to be separated more naturally in the code. The GPI engine itself anticipates the user multithreading and performs some of the necessary synchronization for the application, further simplifying the engineer's task.

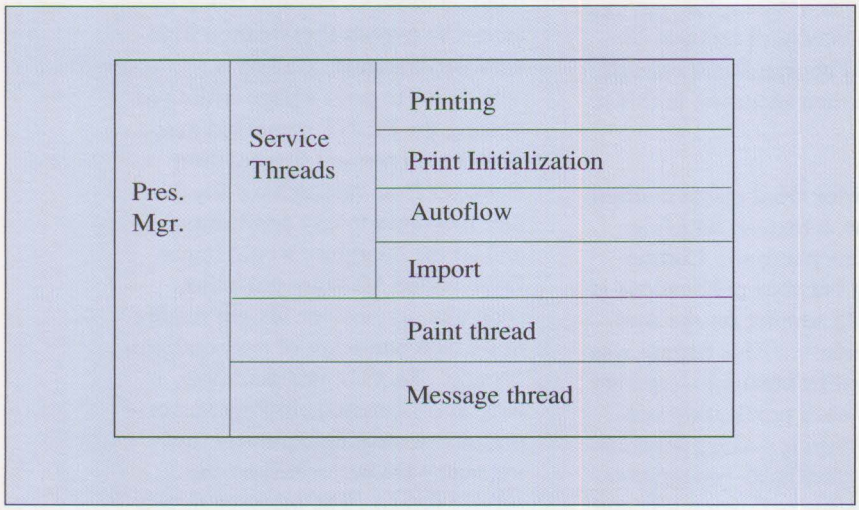


Figure 3. Thread Structure

Assessing the OS/2 Development Environment

In the early SDKs, it was advised that development be done under DOS and that OS/2 be used only for testing. As the SDKs matured, however, it became more advantageous to work in OS/2 entirely. Now we have come full circle – developing for Windows on OS/2, and rebooting to DOS for testing.

One of the first things developers get very accustomed to in OS/2 is having multiple command sessions. This is not only useful for doing background or simultaneous compiles, but also for preserving your current state. I find it very comfortable to quickly switch from one screen to another just to work in another directory with a new set of environment variables. Another advantage is being able to keep your train of thought while the compile takes place. Under DOS, it is common to juggle a mental list of minor editing changes to make once the compile completes – and not uncommon to lose time if something is forgotten.

The next thing to notice is that the dreaded "out of heap" message disappears from the C compiler. All the development tools run in protected mode, and therefore have access to considerably more memory for internal table storage. No longer is the developer interrupted by the need to break up modules artificially just to get them to compile. Linking a large application, especially when containing debug

code and debugger data, is an order of magnitude faster with OS/2 memory management than with using virtual-disk scratch files under DOS.

Debugging in OS/2, using CodeView®, Logitech's™ Multiscope™ or one of the other available debuggers, is much easier than in either Windows or the Macintosh, although it is not without deficiencies (such as being unable to interrupt a looping application). Source-level debugging with these tools is a great improvement over glass-TTY tools such as Windows SYMDEB. In addition, because of OS/2 multitasking the debugging tools intrude less on the target application. (PageMaker was never able to use CodeView in practice on Windows – 640 KB was just too small.)

QuickHelp (in the SDK) is also very effective in this environment. I frequently leave a QuickHelp window on the screen for reference while editing source code. It generally provides thorough coverage of the parameters and options available for any system call, eliminating the need to shuffle through the sizable manuals. The engineer can have the application, debugger, source files and reference manual all within a few keystrokes. These things do require memory, though: our development was done on 4 MB machines, but that is the bare minimum. 6 MB prevents a lot of swapping activity.

Last but not least, a protected memory environment is a great time-saver during development, just to isolate dumb mistakes and uninitialized pointers. Some potentially nasty bugs – the Ctrl-Alt-Del kind on Windows – have become trivial to find and fix when the hardware gives you a helping hand.

OS/2's architecture lays the groundwork for much tighter coordination among different tasks on a user's desktop.

So what are the problems with OS/2 development? For one thing, your favorite TSRs and tools still may not be there. Many tools will run in the DOS compatibility box, but cannot be invoked from protected-mode command files yet. The same may be true of various hardware or network options. You may have to reboot DOS until the driver support is there. Finally, Presentation Manager is robust about announcing errors, but finding the errors themselves can sometimes be tedious. There is room for a lot more diagnostic tools to help clean and tune the code. But on the whole, OS/2 is well worth investigation as a development environment.

Potential

PageMaker for OS/2 has been recognized in the press for making some of the benefits of OS/2 real to the end user. OS/2's architecture lays the groundwork for much tighter coordination among different tasks on a user's desktop. Some will be invoked by explicit user action or result from links between documents; some will result from background service tasks that are not visible to the average user except in the form of enhanced productivity. Networking, too, will take place at various levels of user visibility. As more applications like PageMaker become available on OS/2, a critical application mass will be reached, and this potential will be tapped.

ABOUT THE AUTHOR

Peter Kron, a principal software engineer at Aldus Corporation, was the technical lead in development of Aldus PageMaker for OS/2. He also worked on both the Windows and Macintosh versions of Aldus PageMaker. Peter received a degree in mathematics from Dartmouth College. Since then, he has worked extensively in the development of database software and publishing workstations.

Aldus Corporation
411 First Ave. South, Suite 200
Seattle, Washington 98104

Object-Oriented Programming with C and OS/2 PM – Is It Possible?

Hans J. Eisenhuth
SE International, Inc.
Boca Raton, Florida

This article aims to prove that object-oriented programming with OS/2 Presentation Manager (PM) and C is possible, even though C is not an object-oriented programming language. It also shows that object-oriented design and the right programming techniques have more influence on the object-orientedness of an application than the object-orientedness of the programming language itself.

The discussion about object-oriented programming (OOP) has increased dramatically with the introduction of OS/2 Presentation Manager and object-oriented programming languages such as C++®, Ada and Smalltalk.

Whenever you talk about the advantages of Presentation Manager as a base for software development you come across statements such as:

- You can only write object--oriented applications with a true object-oriented programming language like Smalltalk or C++.
- C is not an object-oriented programming language. Therefore PM applications written in C cannot be object-oriented.

These statements are also used as an excuse for the fact that the first application developed with C and PM does not show the expected object-orientedness results.

We will analyze these statements and find out whether they are justified. At the end we will see that object-oriented programming is achievable with C and PM. We just have to adapt our problem-solving techniques and change our way of thinking a bit.

Criteria for Object-Orientedness

Most of the literature about OOP briefly discusses the fundamentals before starting a detailed discussion of object-oriented programming based on an OOP programming language. The reader might therefore come to the conclusion that only real OOP languages allow object-oriented programming.

An exception to the rule is Bertrand Meyer's book *Object-Oriented Software Construction* (Prentice Hall, pages 60-63), in which he lists the "7 steps toward object-based happiness." These steps are criteria for determining the object-orientedness of an application.

These criteria will be used to check whether OS/2 applications written in C can be object-oriented.

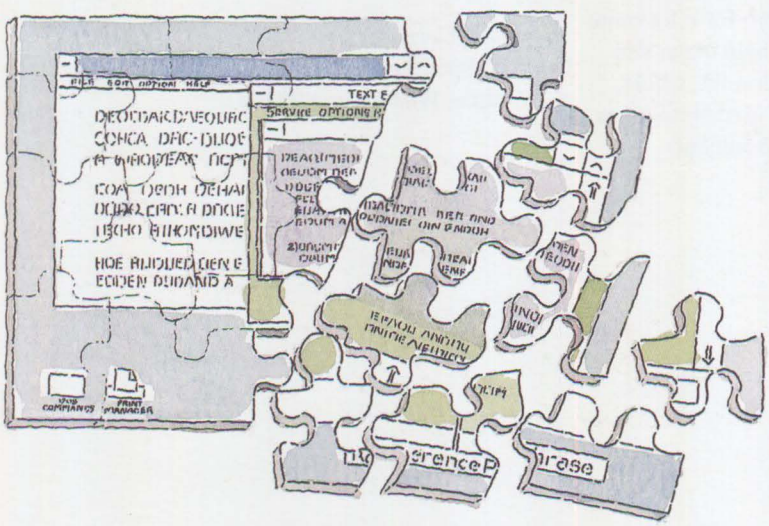
Object-Based Modularization: To achieve true object-orientedness, applications must be modularized on the basis of their data structures. However, the majority of application designers and developers still use the data flow-oriented design principles – that's why we still see so many flowcharts. This approach does *not* lead to an object-oriented design and therefore not to an object-oriented application.

Modern software engineering distinguishes between three design principles:

- Data flow-oriented design (DFD)
- Data structure-oriented design (DSD)
- Object-oriented design (OOD)

These design principles are used to determine the modules (highly independent program entities) of an application.

The data flow-oriented design is based on a "top-down" design technique, whereas the data structure-oriented design is based on a "bottom-up" design technique. The object-oriented design, as the newest principle, combines the techniques



of DFD and DSD to a "Yo-Yo"(top-down / bottom-up) technique. The different design principles are discussed extensively by Roger S. Pressman in his book *Software Engineering – A Practitioner's Approach (McGraw-Hill).*

Only the object-oriented design approach will lead to an object-oriented application with all its advantages. If we have a proper object-oriented oriented design we can implement systems with PM and C and get object-oriented applications, even though C is not an object-oriented language.

Abstract Data Types: Truly object-oriented applications allow the definition and use of abstract data types.

An abstract data type is specified as a list of services which allow the use and modification of data structures. These services are operations or features that allow the external view of objects without disclosing the internally used data structures.

The book, *Einführung in Software Engineering* (sorry, it's a book written in German) uses abstract data types to:

- Isolate important data structures from their usage
- Define prototypes of common data structures together with their access routines

Figure 1 shows how an abstract data type hides the data structures through an interface of access routines.

To implement abstract data types, a modern programming language must provide the following:

- Definition of abstract data types
- Creation of objects of an abstract data type
- Manipulation of an object exclusively through the use of services defined in the abstract data type
- Side-effect-free use of data

In Presentation Manager, an abstract data type is equal to a window class, and a window is equivalent to an object. The PM function **WinRegisterClass** defines a window class, and **WinCreateWindow** creates a window of a given window class.

Sending or posting a message to a window is equal to a service request. The window procedure of a window class determines how all objects of a class have to respond to service requests. That means the window procedure defines the behavior of all windows of a window class.

A good programming technique to hide the PM-specific parameters and functions for the creation of an object is to provide an object creation function for each window class. This should perform the class registration and object creation. The parameter list for the creation function must contain only application specific parameters. Figure 2 shows the layout of an object creation function.

A key requirement for abstract data types is that their description and implementation is absolutely free of side effects. A side effect occurs when an object uses or manipulates data that has been defined outside of the object's scope.

It is obvious that the exclusive use of instance variables in an application (be it object-oriented or not) reduces the maintenance effort dramatically. This requirement is automatically implemented in real OOP languages. In a PM/C application we have to use proper programming techniques to fulfill this requirement. We have to make sure that a window does *not* use any global variables or variables defined outside of its window procedure. The window data (or better called instance data) should be stored or

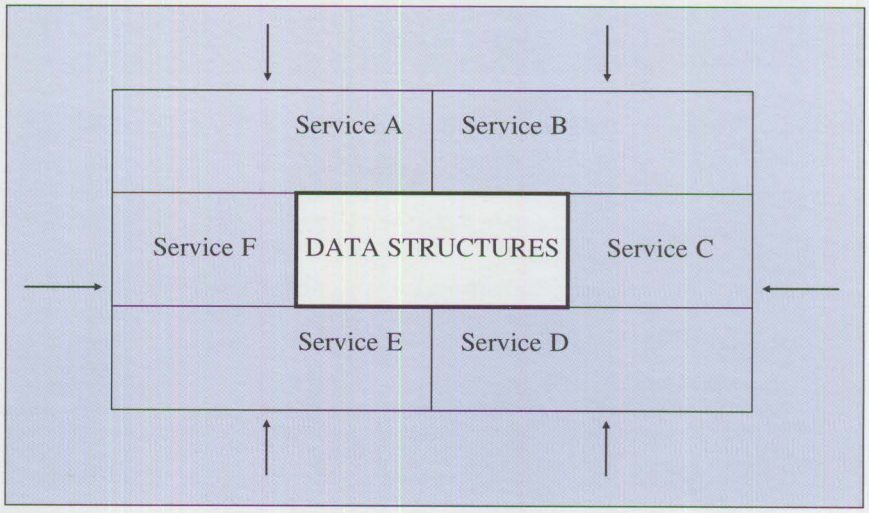


Figure 1. Abstract Data Types: Access routines hide the internal data structure

maintained with the help of window words.

OS/2 and C provide a wide (some people say, too wide) variety of memory management functions and good support for maintaining these window words. But application developers and designers have not taken enough notice of this technique so far. Figure 3 shows how an object-oriented window procedure should look.

If we use only properly implemented instance data and use only PM's message-passing philosophy to modify the instance data, we are able to implement abstract data types in a PM application written in C (PM/C).

Automatic Memory Management: For the purpose of de-allocating un-

```
HWND CreateObject ( Input Parameters) {

WinRegisterClass (.....);
Error code checking;

Prepare the input parameters so that they can be passed
properly to the new window object as a control parameter;

hwndObject = WinCreateWindow (...., control parameter,.);
error code checking;

return (hwndObject);
}
```

Figure 2. Object Creation Function: Used to hide complex PM functions

```
INT EXPENTRY WinProc (HWND hwnd, USHORT msg, LPARAM1 lp1, LPARAM2 lp2) {

      typedef struct_INSTANCE {
            USHORT usNumber;
            . . . .              /* Further instance data              */
            }INSTANCE;

      INSTANCE *pInstance;       /* Pointer to window's instance data  */

      switch (msg)
        case WM_CREATE:
            Allocate space for the instance data;
            Initialize the instance data;
            Store the pointer pInstance in a window word;
            Do further initialization;
            break;

        case WM_XXX:
            Retrieve pInstance from the window word;
            Perform the WM_XXX related behavior;
            break;

        . . . .

        case WM_DESTROY:
            Retrieve pInstance from the window word;
            Free the memory to which pInstance points;
            break;

        default:
            return (WinDefWindowProc (hwnd, msg, lp1, lp2));
        }
        return (0L);
}/*END of WinProc*/
```

Figure 3. Use of Instance Data

used objects, real object-oriented languages provide the function of a garbage collector, which browses periodically through the application's bound memory resources and frees unreferenced portions of memory.

As we have seen in the previous chapter, management of instance data for an object must be done "manually" in its window procedure. In the implementation of a window procedure we have to ensure that the instance data of a window is freed when the WM_DESTROY message is received by an object (Figure 3).

However, the hierarchical window structure in Presentation Manager makes sure that all dependent windows are destroyed whenever the parent window is destroyed. No programmer intervention is necessary.

Classes: A class is the implementation of an abstract data type in a highly independent and coherent fashion. To ensure this independence, the application must be modularized so that each module is equal to the implementation of an abstract data type.

Because C does not provide a native support for independent classes, we have to ensure independence through the usage of good programming techniques.

Let us have a look at PM's predefined system classes, such as classes for list boxes, entry fields or standard windows. The code for these window classes is held in separate Dynamic Link Libraries (DLLs). DLLs are perfectly suited to hold an independent implementation of an abstract data type. Therefore, we should also use this technique for application-specific classes.

Each Dynamic Link Library should contain:

- The window procedure(s)
- Related functions (such as the object creation function)
- All privately used functions
- The related PM resource definitions

This technique enforces class autonomy and data usage without side effects. Modification of a class which is held in a DLL is highly transparent for other applications that use objects of this class. Applications (or other classes) don't have to be recompiled or relinked, because the code of the modified window class is linked dynamically during runtime.

Presentation Manager also allows the creation of public new system window classes. This means that new window classes are registered at system start and applications no longer care about registering new classes.

But don't forget: to use this feature of OS/2 efficiently, you must have designed your application(s) in an object-oriented fashion.

Inheritance: Inheritance is a key issue in object-oriented programming. It leads to high code reusability and easier maintenance, because it allows the definition of a new class as an extension or restriction of an already existing class. Only the modified behavior must be described in the new class, while the unchanged behavior is inherited.

The implementation of inheritance in a PM/C application is based on subclassing. The subclassing technique allows extension or restriction of an object's functionality simply by intercepting the message flow to the object. Be aware that this technique changes the behavior of an object, but not of a class. The classic

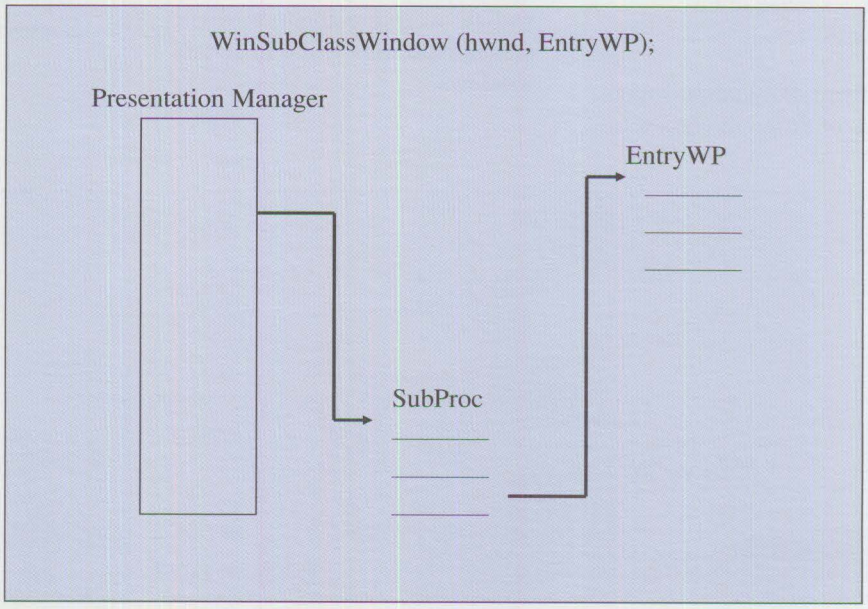


Figure 4. Message Flow to a Subclassed Window Procedure

example for subclassing is the implementation of a numeric entry field, which inherits the behavior of an unrestricted entry field (Figure 4).

The **WinSubClassWindow** function tells PM that all messages to a specific window (hwnd) should be passed first to an application-specific procedure (SubProc). SubProc analyzes all messages before it passes them to the "real" window procedure (EntryWP). The functionality of the entry field can now be extended, for example, by displaying a message if the user presses an alphabetic key, or restricted, for example, by filtering alphabetic WM_CHAR messages.

SubProc has the same structure as a window procedure. Therefore, we can create a new window class of numeric entry fields by creating a public window class with SubProc as a window procedure. Figure 5 shows the layout of the numeric entry field window procedure SubProc. Notice that SubProc calls EntryWP instead of the Window default procedure. This actually implements the inheritance in PM window classes.

Polymorphism: Polymorphism enables objects of different classes to respond to the same service request. A program entity requesting a service from an object doesn't care which class the object belongs to. It just issues the request. Of course, it is necessary that the class behavior is prepared to service the request in a commonly agreed manner, but the implementation may vary from class to class.

It is possible to develop fully object-oriented applications with C under the control of OS/2 Presentation Manager.

A Presentation Manager example is the message WM_PAINT. Any window of any class should repaint itself when it receives the WM_PAINT message. Presentation Manager just sends this message to all windows that have to be updated without worrying about the implementation of repainting in certain windows.

To allow polymorphism in a PM information system, we should define a range of message identifiers which are reserved system-wide for polymorphic use. Whenever a new class is developed, the development team has to ensure that the window procedure of this new class responds correctly to these polymorphic messages.

Multiple and Repeated Inheritance: This criterion of object-orientedness is just an extension of the previously discussed inheritance criterion. It allows an object to have more than one parent, from whom it inherits functionalities. This is a very specific requirement for an OOP language; however, it should be mentioned here for completeness.

In real OOP languages this requirement makes special language constructs necessary. Presentation Manager and C do not provide special language constructs for inheritance. The entire PM/C program processing is based on sending and receiving messages. This basic technique allows the definition of object classes with more than one parent.

```
MRESULT EXPENTRY SubProc (HWND hwnd, USHORT msg, LPARAM lp1, LPARAM lp2){
      Query original entry field WP EntryWP
      switch (msg){
        case WM_CHAR:
           if    character is alphabetic
                 {Display error message or beep
                 Return (0)}
           break;
      } /* end switch */
      return (EntryWP (hwnd, msg, lp1, lp2));
} /* End SubProc */
```

Figure 5. Subclass Window Procedure: Window procedure for numeric entry field class

Conclusion

Having examined the different criteria of object-orientedness, we can say that it is possible to develop fully object-oriented applications with C under the control of OS/2 Presentation Manager.

Contrary to the PM/C environment, real OOP languages provide language constructs that enforce object-oriented programming. These language constructs allow for more comfortable creation and structuring of classes, objects and instance data. They also check the correct use of objects at compilation time, which facilitates program debugging. Therefore, object-oriented programming becomes more obvious in languages like Smalltalk or C++.

The advantage of C and Presentation Manager is their conformance to SAA. It assures compatibility with current and future components of the SAA architecture and protects your hardware, software and programmer education investments.

Irrespective of the language you use for implementing an object-oriented application, it is much more important to apply the right object-oriented design methodology. If you have not designed your application in an object-oriented fashion, you will not successively implement an object-oriented application, even if you use a real OOP language.

If an application has been designed to be object-oriented, you will find that OS/2 Presentation Manager and C provide excellent capabilities to implement multitasking and SAA/CUA conforming applications in a cooperative processing environment. And that's where we want to go!

ABOUT THE AUTHOR

Hans J. Eisenhuth worked from 1986 to 1989 for IBM's International Technical Support Center (ITSC) in Boca Raton, where he developed and taught education classes for OS/2 and Presentation Manager application development and design. Since 1989 he has been president of SE International, Inc., which provides advanced software engineering education and consulting services. Hans holds an M.S. degree in computer science and a B.A. in mathematics and economics from the Technical University in Berlin, Germany.

SE International, Inc.
One Park Place
621 NW 23rd Street
Boca Raton, Florida 33487

Design Goals and Implementation of the New High Performance File System

Ray Duncan
Laboratory Microsystems Inc.
Los Angeles, California

Reprinted with permission from Microsoft® Systems Journal, Volume 4, #5
©1989, Microsoft Corporation

The High Performance File System (HPFS), which is making its first appearance in the OS/2 Version 1.2 operating system, had its genesis in the network division of Microsoft and was designed by Gordon Letwin, the chief architect of the OS/2 operating system. The HPFS has been designed to meet the demands of increasingly powerful PCs, fixed disks, and networks for many years to come and to serve as a suitable platform for object-oriented languages, applications, and user interfaces.

The HPFS is a complex topic because it incorporates three distinct yet interrelated file system issues. First, the HPFS is a way of organizing data on a random access block storage device. Second, it is a software module that translates file-oriented requests from an application program into more primitive requests that a device driver can understand, using a variety of creative techniques to maximize performance. Third, the HPFS is a practical illustration of an important new OS/2 feature known as installable file systems.

This article introduces the three aspects of the HPFS. But first, it puts the HPFS in perspective by reviewing some of the problems that led to the system's existence.

FAT File System

The so-called FAT file system, which is the file system used in all versions of the MS-DOS® operating system to date and in the first two releases of OS/2 (see *Note 1*) Versions 1.0 and 1.1, has a dual heritage in Microsoft's earliest programming language products and the Digital Research® CP/M® operating system – software originally written for 8080-based and Z-80-based microcomputers. It inherited characteristics from both ancestors that have progressively turned into handicaps in this new era of multitasking, protected mode, virtual memory, and huge fixed disks.

The FAT file system revolves around the File Allocation Table for which it is named. Each logical volume has its own FAT, which serves two important functions: it contains the allocation information for each file on the volume in the form of linked lists of allocation units (clusters, which are power-of-2 multiples of sectors) and it indicates which allocation units are free for assignment to a file that is being created or extended.

The FAT was invented by Bill Gates and Marc McDonald in 1977 as a method of managing disk space in the NCR version of stand-alone Microsoft® Disk BASIC. Tim Paterson, at that time an employee of Seattle Computer Products (SCP), was introduced to the FAT concept when his company shared a booth with Microsoft at the National Computer Conference in 1979. Paterson subsequently incorporated FATs into the file system of 86-DOS, an operating system for SCP's S-100 bus 8086 CPU boards. 86-DOS was eventually purchased by Microsoft and became the starting point for MS-DOS (see *Note 2*) Version 1.0, which was released for the original IBM PC in August 1981.

When the FAT was conceived, it was an excellent solution to disk management, mainly because the floppy disks on which it was used were rarely larger than 1 MB. On such disks, the FAT was small enough to be held in memory at all times, allowing very fast random access to any part of any file. This proved far superior to the CP/M method of tracking disk space, in which the information about the sectors assigned to a file might be spread across many directory entries, which were in turn scattered randomly throughout the disk directory.

When applied to fixed disks, however, the FAT began to look more like a bug than a feature. It became too large to be held entirely resident and had to be paged into memory in pieces; this paging resulted in many superfluous disk head movements as a program was reading through a file and degraded system throughput. In addition, because the information about free disk space was dispersed across many sectors of FAT, it was impractical to allocate file space contiguously, and file fragmentation became another obstacle to good performance. Moreover, the use of relatively large clusters on fixed disks resulted in a lot of dead space, since an average of one-half cluster was wasted for

each file. (Some network servers use clusters as large as 64 KB.)

The FAT file system's restrictions on naming files and directories are inherited from CP/M. When Paterson was writing 86-DOS, one of his primary objectives was to make programs easy to port from CP/M to his new operating system. He therefore adopted CP/M's limits on filenames and extensions so the critical fields of 86-DOS File Control Blocks (FCBs) would look almost exactly like those of CP/M. The sizes of the FCB filename and extension fields were also propagated into the structure of disk directory entries. In due time, 86-DOS became MS-DOS and application programs for MS-DOS proliferated beyond anyone's wildest dreams. Since most of the early programs depended on the structure of FCBs, the 8.3 format for filenames became irrevocably locked into the system.

During the last couple of years, Microsoft and IBM have made valiant attempts to prolong the useful life of the FAT file system by lifting the restrictions on volume sizes, improving allocation strategies, caching pathnames, and moving tables and buffers into expanded memory. But these can only be regarded as temporizing measures, because the fundamental data structures used by the FAT file system are simply not well suited to large random access devices.

The HPFS solves the FAT file system problems mentioned here and many others, but it is not derived in any way from the FAT file system. The architect of the HPFS started with a clean sheet of paper and designed a file system that can take full advantage of a multitasking environment, and that will be able to cope with any sort of disk device

	FAT File System	**High Performance File System**
Maximum filename length	11 (in 8.3 format)	254
Number of dot (.) delimeters allowed	One	Multiple
File attributes	Bit flags	Bit flags plus up to 64 KB of free-form ASCII or binary information
Maximum path length	64	260
Minimum disk space overhead per file	Directory entry (32 bytes)	Directory entry (length varies) + Fnode (512 bytes)
Average wasted space per file	1/2 cluster (typically 2 KB or more)	1/2 sector (256 bytes)
Minimum allocation unit	Cluster (typically 4 KB or more)	Sector (512 bytes)
Allocation info for files	Centralized in FAT on home track	Located nearby each file in its Fnode
Free disk space info	Centralized in FAT on home track	Located near free space in bitmaps
Free disk space described per byte	2 KB (1/2 cluster at 8 sectors/cluster)	4 KB (8 sectors)
Directory structure	Unsorted linear list, must be searched exhaustively	Sorted B-Tree
Directory location	Root directory on home track, others scattered	Localized near seek center of volume
Cache replacement strategy	Simple LRU	Modified LRU, sensitive to data type and usage history
Read-ahead	None in MS-DOS 3.3 or earlier, primitive read-ahead optional in MS-DOS 4	Always present, sensitive to data type and usage history
Write-behind	Not available	Used by default, but can be defeated on per-handle basis

likely to arrive on microcomputers during the next decade.

HPFS Volume Structure

HPFS volumes are a new partition type – type 7 – and can exist on a fixed disk alongside of the several previously defined FAT partition types. IBM-compatible HPFS volumes use a sector size of 512 bytes and have a maximum size of 2199 GB (2^{32} sectors). Although there is no particular reason why floppy disks can't be formatted as HPFS volumes, Microsoft plans to stick with FAT file systems on floppy disks for the foreseeable future. (This ensures that users will be able to transport files easily between MS-DOS and OS/2 systems.)

An HPFS volume has very few fixed structures (Figure 1). Sectors 0-15 of a volume (8 KB) are the BootBlock and contain a volume name, 32-bit volume ID, and a disk bootstrap program. The bootstrap is relatively sophisticated (by MS-DOS standards) and can use the HPFS in a restricted mode to locate and read the operating system files wherever they might be found.

Sectors 16 and 17 are known as the SuperBlock and the SpareBlock respectively. The SuperBlock is only modified by disk maintenance utilities. It contains pointers to the free space bitmaps, the bad block list, the directory block band, and the root directory. It also contains the date that the volume was last checked out and repaired with CHKDSK/F. The SpareBlock contains various flags and pointers that will be discussed later; it is modified, although infrequently, as the system executes.

The remainder of the disk is divided into 8 MB bands. Each band has its own free space bitmap in which a bit represents each sector. A bit is 0 if the sector is in use and 1 if the sector is available. The bitmaps are located at the head or tail of a band so that two bitmaps are adjacent between alternate bands. This allows the maximum contiguous free space that can be allocated to a file to be 16 MB. One band, located at or toward the seek center of the disk, is called the directory block band and receives special treatment (more about this later). Note that the band size is a characteristic of the current implementation and may be changed in later versions of the file system.

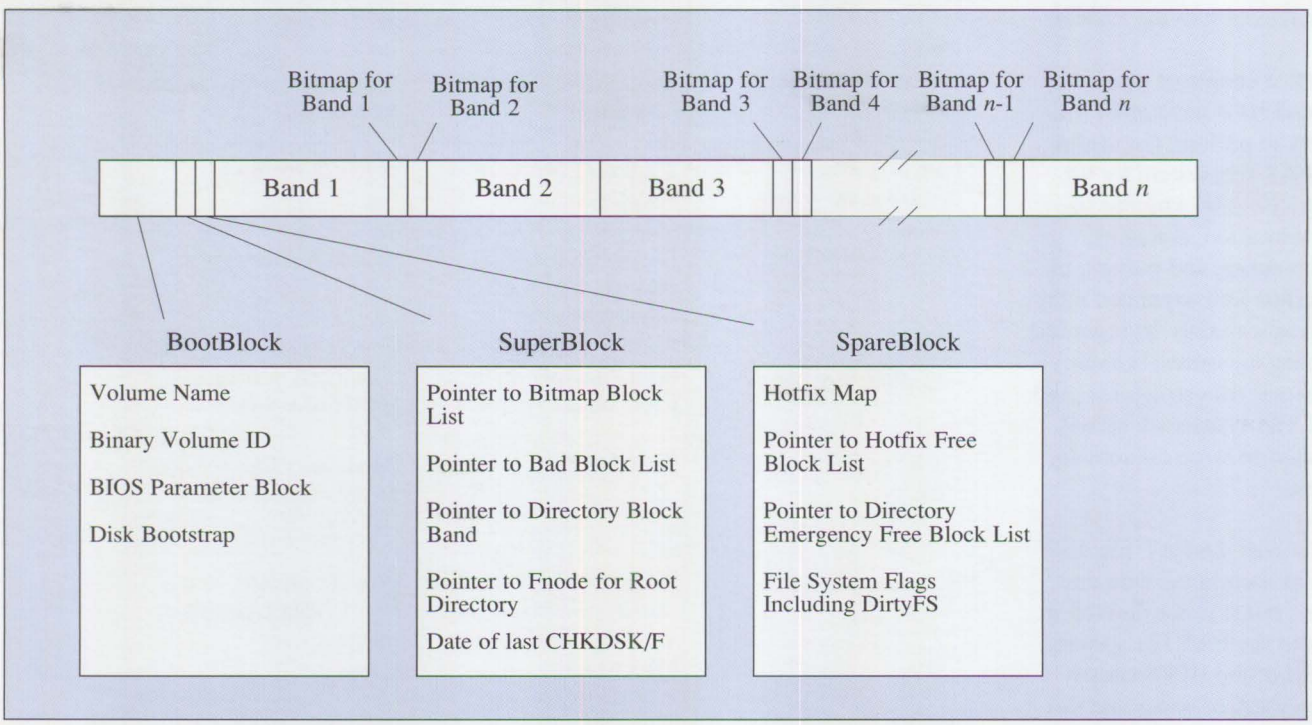


Figure 1. This figure shows the overall structure of an HPFS volume. The most important fixed objects in such a volume are the BootBlock, the SuperBlock, and the SpareBlock. The remainder of the volume is divided into 8 MB bands. There is a freespace bitmap for each band and the bitmaps are located between alternate bands; consequently, the maximum contiguous space which can be allocated to a file is 16 MB.

Files and Fnodes

Every file or directory on an HPFS volume is anchored on a fundamental file system object called an Fnode (pronounced "eff node"). Each Fnode occupies a single sector and contains control and access history information used internally by the file system, extended attributes and access control lists (more about this later), the length and the first 15 characters of the name of the associated file or directory, and an allocation structure (Figure 2). An Fnode is always stored near the file or directory that it represents.

The allocation structure in the Fnode can take several forms, depending on the size and degree of contiguity of the file or directory. The HPFS views a file as a collection of one or more runs or extents of one or more contiguous sectors. Each run is symbolized by a pair of doublewords – a 32-bit starting sector number and a 32-bit length in sectors (this is referred to as run-length encoding). From an application program's point of view, the extents are invisible; the file appears as a seamless stream of bytes.

The space reserved for allocation information in an Fnode can hold pointers to as many as eight runs of sectors of up to 16 MB each. (This maximum run size is a result of the band size and free space bitmap placement only; it is not an inherent limitation of the file system.) Reasonably small files or highly contiguous files can therefore be described completely within the Fnode (Figure 3).

HPFS uses a new method to represent the location of files that are too large or too fragmented for the Fnode and consist of more than eight runs. The Fnode's allocation structure becomes the root for a B+ Tree of allocation sectors, which in turn contain the actual pointers to the file's sector runs (see Figure 4 and the section, "B-Trees and B+ Trees"). The Fnode's root has room for 12 elements. Each allocation sector can contain, in addition to various control information, as many as 40 pointers to sector runs. Therefore, a two-level allocation B+ Tree can describe a file of 480 (12*40) sector runs, with a theoretical maximum size of 7.68 GB (12*40*16 MB) in the current implementation (although the 32-bit

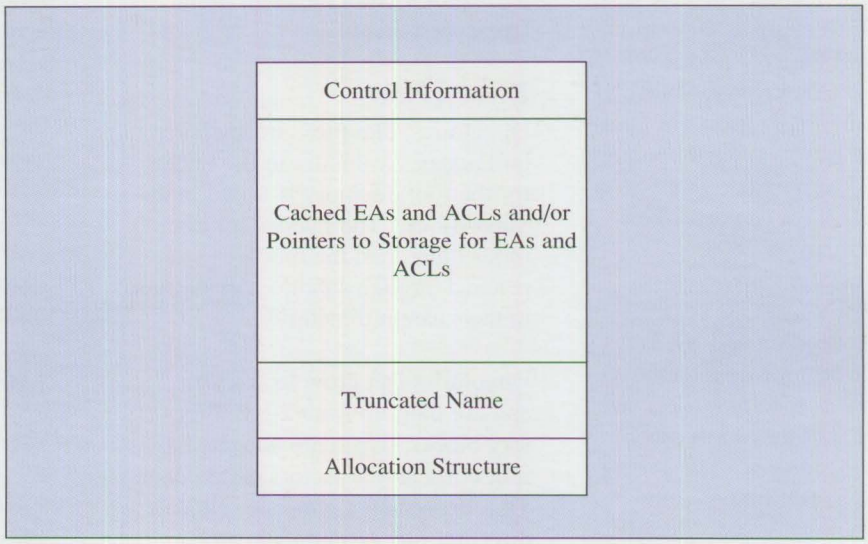


Figure 2. This figure shows the overall structure of an Fnode. The Fnode is the fundamental object in an HPFS volume and is the first sector allocated to a file or directory. It contains control and access history information used by the file system, cached EAs and ACLs or pointers to same, a truncated copy of the file or directory name (to aid disk repair programs), and an allocation structure which defines the size and location of the file's storage.

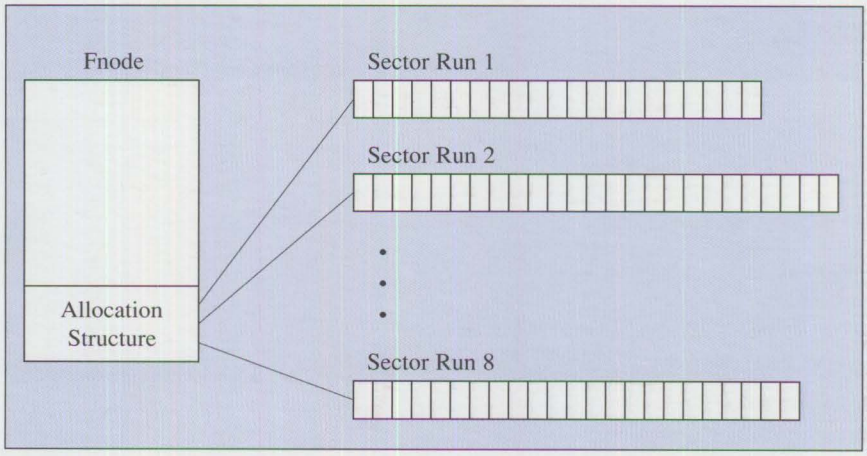


Figure 3. The simplest form of tracking for the sectors owned by a file is shown. The Fnode's allocation structure points directly to as many as eight sector runs. Each run pointer consists of a pair of 32–bit doublewords: a starting sector number, and a length in sectors.

signed offset parameter for DosChgFilePtr effectively limits file sizes to 2 GB).

In the unlikely event that a two-level B+ Tree is not sufficient to describe a highly fragmented file, the file system will introduce additional levels in the tree as needed. Allocation sectors in the intermediate levels can hold as many as 60 internal (nonterminal) B+ Tree nodes, which means that the descriptive ability of this structure rapidly grows to numbers that are nearly beyond comprehension. For example, a three-level allocation B+ Tree can describe a file with as many as 28,800 (12*60*40) sector runs.

Run-length encoding and B+ Trees of allocation sectors are a memory-efficient way to specify a file's size and location, but they have other significant advantages. Translating a logical file offset into a sector number is extremely fast: the file system just needs to traverse the list (or B+ Tree of lists) of run pointers until it finds the correct range. It can then identify the sector within the run with a simple calculation. Run-length encoding also makes it trivial to extend the file logically if the newly assigned sector is contiguous with the file's previous last sector; the file system merely needs to increment the size doubleword of the file's last run pointer and clear the sector's bit in the appropriate freespace bitmap.

Directories

Directories, like files, are anchored on Fnodes. A pointer to the Fnode for the root directory is found in the SuperBlock. The Fnodes for directories other than the root are reached through subdirectory entries in their parent directories.

Directories can grow to any size and are built up from 2 KB directory blocks, which are allocated as four consecutive sectors on the disk. The file system attempts to allocate directory blocks in the directory band, which is located at or near the seek center of the disk. Once the directory band is full, the directory blocks are allocated wherever space is available.

Each 2 KB directory block contains from one to many directory entries. A directory entry contains several fields, including time and date stamps, an Fnode pointer, a usage count for use by disk maintenance programs, the length of the file or directory name, the name itself, and a B-Tree pointer. Each entry begins with a word that contains the length of the entry. This provides for a variable amount of flex space at the end of each entry, which can be used by special versions of the file system and allows the directory block to be traversed very quickly (Figure 5).

The number of entries in a directory block varies with the length of names. If the average filename length is 13 characters, an average directory block will hold about 40 entries. The entries in a directory block are sorted by the binary lexical order of their name fields (this happens to put them in alphabetical

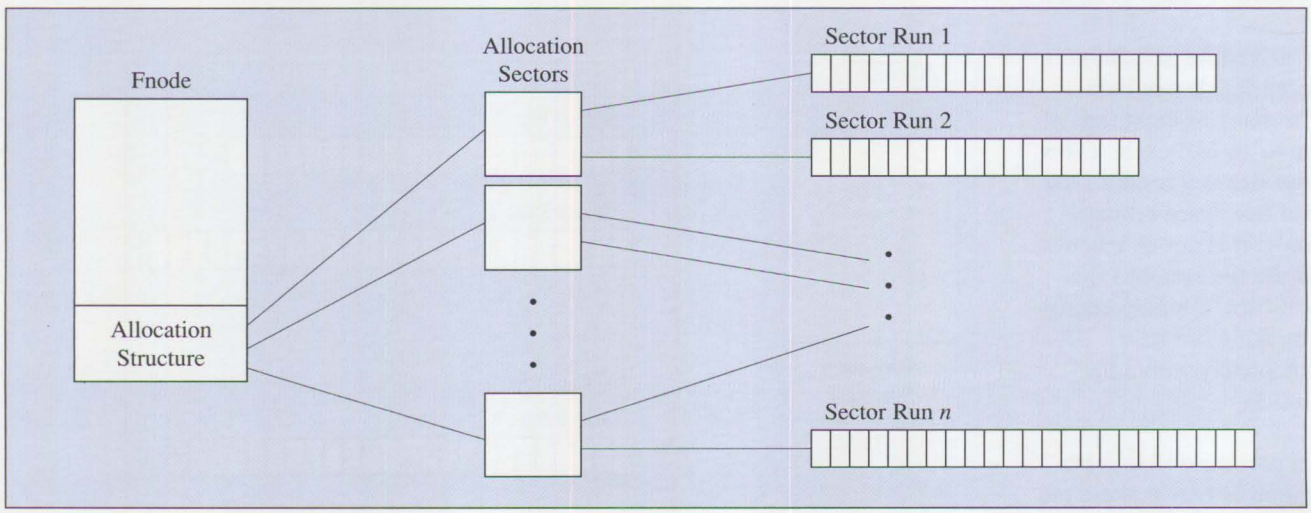


Figure 4. This figure demonstrates the technique used to track the sectors owned by a file with 9–480 sector runs. The allocation structure in the Fnode holds the roots for a B+ Tree of allocation sectors. Each allocation sector can describe as many as 40 sector runs. If the file requires more than 480 sector runs, additional intermediate levels are added to the B+ Tree, which increases the number of possible sector runs by a factor of sixty for each new level.

order for the U.S. alphabet). The last entry in a directory block is a dummy record that marks the end of the block.

When a directory gets too large to be stored in one block, it increases in size by the addition of 2 KB blocks that are organized as a B-Tree (see "B-Trees and B+ Trees"). When searching for a specific name, the file system traverses a directory block until it either finds a match or finds a name that is lexically greater than the target. In the latter case, the file system extracts the B-Tree pointer from the entry. If there is no pointer, the search failed; otherwise the file system follows the pointer to the next directory block in the tree and continues the search.

A little back-of-the-envelope arithmetic yields some impressive statistics. Assuming 40 entries per block, a two-level tree of directory blocks can hold 1,640 directory entries and a three-level tree can hold an astonishing 65,640 entries. In other words, a particular file can be found (or shown not to exist) in a typical directory of 65,640 files with a maximum of three disk hits

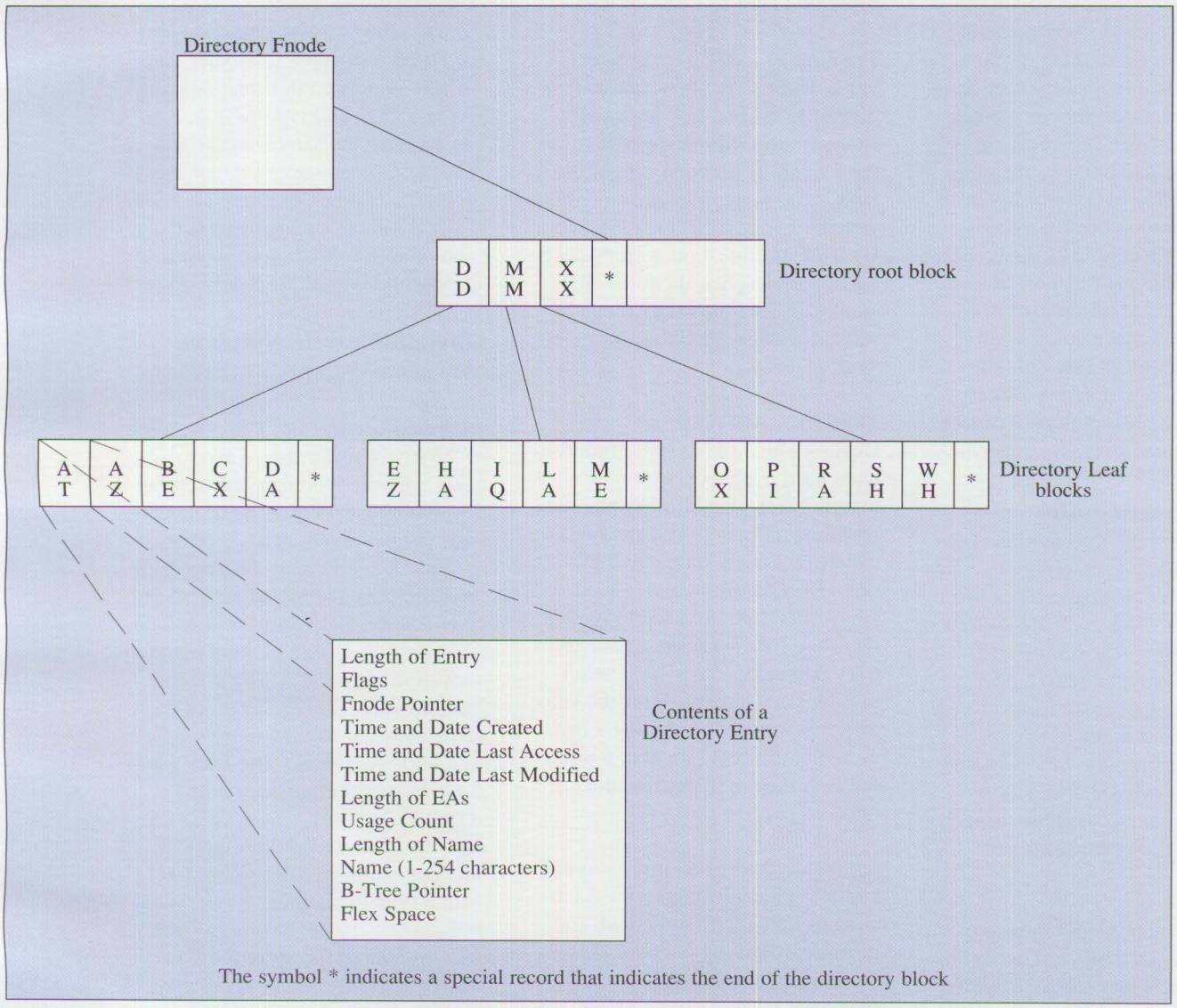


Figure 5. Here, directories are anchored on an Fnode and are built up from 2 KB directory blocks. The number of entries in a directory block varies because the length of the entries depends on the filename. When a directory requires more than one block, the blocks are organized as a B-Tree. This allows a filename to be located very quickly with a small number of disk accesses, even when the directory grows very large.

the actual number of disk accesses depending on cache contents and the location of the file's name in the directory block B-Tree. That's quite a contrast to the FAT file system, where in the worst case more than 4,000 sectors would have to be read to establish that a file was or was not present in a directory containing the same number of files.

The B-Tree directory structure has interesting implications beyond its effect on open and find operations. A file creation, renaming, or deletion may result in a cascade of complex operations, as directory blocks are added or freed or names are moved from one block to the other to keep the tree balanced. In fact, a rename operation could theoretically fail for lack of disk space even though the file itself is not growing. In order to avoid this sort of disaster, the HPFS maintains a small pool of free blocks that can be drawn from in a directory emergency; a pointer to this pool of free blocks is stored in the SpareBlock.

Extended Attributes

File attributes are information about a file that is maintained by the operating system outside the file's overt storage area. The FAT file system supports only a few simple attributes (read only, system, hidden, and archive) that are actually stored as bit flags in the file's directory entry; these attributes are inspected or modified by special function calls and are not accessible through the normal file open, read, and write calls.

The HPFS supports the same attributes as the FAT file system for historical reasons, but it also supports a new form of file-associated, highly generalized information called Extended Attributes (EAs).

Each EA is conceptually similar to an environment variable, taking the form

name=value

except that the value portion can be either a null-terminated (ASCIIZ) string or binary data. In OS/2 1.2, each file or directory can have a maximum of 64 KB of EAs attached to it. This limit may be lifted in a later release of OS/2.

The storage method for EAs can vary. If the EAs associated with a given file or directory are small enough, they will be stored right in the Fnode. If the total size of the EAs is too large, they are stored outside the Fnode in sector runs, and a B+ Tree of allocation sectors can be created to describe the runs. If a single EA gets too large, it can be pushed outside the Fnode into a B+ Tree of its own.

The kernel API functions DosQFileInfo and DosSetFileInfo have been expanded with new information levels that allow application programs to manipulate extended attributes for files. The new functions DosQPathInfo and DosSetPathInfo are used to read or write the EAs associated with arbitrary pathnames. An application program can either ask for the value of a specific EA (supplying a name to be matched) or can obtain all of the EAs for the file or directory at once.

Although application programs can begin to take advantage of EAs as soon as the HPFS is released, support for EAs is an essential component in Microsoft's long-range plans for object-oriented file systems. Information of almost any type can be stored in EAs, ranging from the name of the application that owns the file to names of dependent files to icons to executable code. As the HPFS evolves, its facilities for manipulating EAs are likely to become much more sophisticated. It's easy to imagine, for example, that in future versions the API might be extended with EA functions that are analogous to DosFindFirst and DosFindNext and EA data might get organized into B-Trees.

I should note here that in addition to EAs, the LAN Manager version of HPFS will support another class of file-associated information called Access Control Lists (ACLs). ACLs have the same general appearance as EAs and are manipulated in a similar manner, but they are used to store access rights, passwords, and other information of interest in a networking multiuser environment.

Installable File Systems

Support for installable file systems has been one of the most eagerly anticipated features of OS/2 Version 1.2. It will make it possible to access multiple incompatible volume structures – FAT, HPFS, CD ROM, and perhaps even UNIX® on the same OS/2 system at the same time, will simplify the life of network implementors, and will open the door to rapid file system evolution and innovation. Installable file systems are, however, only relevant to the HPFS insofar as they make use of the HPFS optional. The FAT file system is still embedded in the OS/2 kernel, as it was in OS/2 1.0 and 1.1, and will remain there as the compatibility file system for some time to come.

An installable file system driver (FSD) is analogous in many ways to a device driver. An FSD resides on the disk in a file that is structured like a dynamic-link library (DLL),

typically with a SYS or IFS extension, and is loaded during system initialization by IFS= statements in the CONFIG.SYS file. IFS= directives are processed in the order they are encountered and are also sensitive to the order of DEVICE= statements for device drivers. This lets you load a device driver for a nonstandard device, load a file system driver from a volume on that device, and so on.

Once an FSD is installed and initialized, the kernel communicates with it in terms of logical requests for file opens, reads, writes, seeks, closes, and so on. The FSD translates these requests – using control structures and tables found on the volume itself – into requests for sector reads and writes for which it can call special kernel entry points called File System Helpers (FsHlps). The kernel passes the demands for sector I/O to the appropriate device driver and returns the results to the FSD (Figure 6).

The procedure used by the operating system to associate volumes with FSDs is called dynamic mounting and works as follows. Whenever a volume is first accessed, or after it has been locked for direct access and then unlocked (for example, by a FORMAT operation), OS/2 presents identifying information from the volume to each of the FSDs in turn until one of them recognizes the information. When an FSD claims the volume, the volume is mounted and all subsequent file I/O requests for the volume are routed to that FSD.

Performance Issues

The HPFS attacks potential bottlenecks in disk throughput at multiple levels. It uses advanced data structures, contiguous sector allocation, intelligent caching, read-ahead, and deferred writes in order to boost performance.

First, the HPFS matches its data structures to the task at hand: sophisticated data structures (B-Trees and B+ Trees) for fast random access to filenames, directory names, and lists of sectors allocated to files or directories, and simple compact data structures (bitmaps) for locating chunks of free space of the appropriate size. The routines that manipulate these data structures are written in assembly language and have been

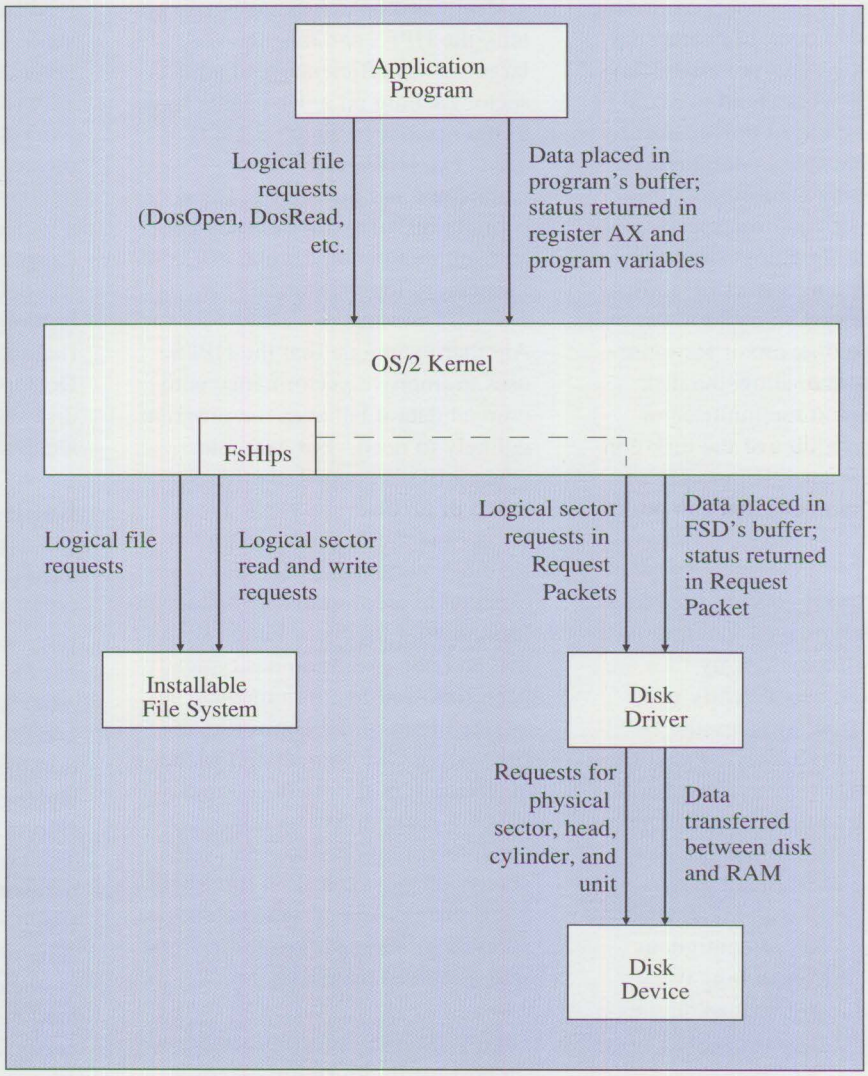


Figure 6. This is a simplified sketch of the relationship between an application program, the OS/2 kernel, an installable file system, a disk driver, and the physical disk device. The application issues logical file requests to the OS/2 kernel by calling the entry points for DosOpen, DosRead, DosWrite, DosChgFilePtr, and so on. The kernel passes these requests to the appropriate installable file system for the volume holding the file. The installable file system sectors translates the logical file requests into requests for reads or writes of logical sectors and calls a kernel File System Helper (FsHlp) to pass these requests to the appropriate disk driver. The disk driver transforms the logical sector requests into requests for specific physical units, cylinders, heads, and sectors, and issues commands to the disk adapter to transfer data between the disk and memory.

painstakingly tuned, with special focus on the routines that search the freespace bitmaps for patterns of set bits (unused sectors).

Next, the HPFS's main goal – its prime directive, if you will – is to assign consecutive sectors to files whenever possible. The time required to move the disk's read/write head from one track to another far outweighs the other possible delays, so the HPFS works hard to avoid or minimize such head movements by allocating file space contiguously and by keeping control structures such as Fnodes and freespace bitmaps near the things they control. Highly contiguous files also help the file system make fewer requests of the disk driver for more sectors at a time, allow the disk driver to exploit the multisector transfer capabilities of the disk controller, and reduce the number of disk completion interrupts that must be serviced.

Of course, trying to keep files from becoming fragmented in a multitasking system in which many files are being updated concurrently is no easy chore. One strategy the HPFS uses is to scatter newly created files across the disk – in separate bands, if possible – so that the sectors allocated to the files as they are extended will not be interleaved. Another strategy is to preallocate approximately 4 KB of contiguous space to the file each time it must be extended and give back any excess when the file is closed.

If an application knows the ultimate size of a new file in advance, it can assist the file system by specifying an initial file allocation when it creates the file. The system will then search all the free space bitmaps to find a run of consecutive sectors large enough to hold the file. That failing, it will search for two runs that are half the size of the file, and so on.

The HPFS relies on several different kinds of caching to minimize the number of physical disk transfers it must request. Naturally, it caches sectors, as did the FAT file system. But unlike the FAT file system, the HPFS can manage very large caches efficiently and adjusts sector caching on a per-handle basis to the manner in which a file is used. The HPFS also caches pathnames and directories, transforming disk directory entries into an even more compact and efficient in-memory representation.

Another technique that the HPFS uses to improve performance is to preread data it believes the program is likely to need. For example, when a file is opened, the file system will preread and cache the Fnode and the first few sectors of the file's contents. If the file is an executable program or the history information in the file's Fnode shows that an open operation has typically been followed by an immediate sequential read of the entire file, the file system will preread and cache much more of the file's contents. When a program issues relatively small read requests, the file system always fetches data from the file in 2 KB chunks and caches the excess, allowing most read operations to be satisfied from the cache.

Finally, the OS/2 operating system's support for multitasking makes it possible for the HPFS to rely heavily on lazy writes (sometimes called deferred writes or write behind) to improve performance. When a program requests a disk write, the data is placed in the cache and the cache buffer is flagged as dirty (that is, inconsistent with the state of the data on disk). When the disk becomes idle or the cache becomes saturated with dirty buffers, the file system uses a captive thread from a daemon process to write the buffers to disk, starting with the oldest data.

In general, lazy writes mean that programs run faster because their read requests will almost never be stalled waiting for a write request to complete. For programs that repeatedly read, modify, and write a small working set of records, it also means that many unnecessary or redundant physical disk writes may be avoided. Lazy writes have their dangers, of course, so a program can defeat them on a per-handle basis by setting the write-through flag in the OpenMode parameter for DosOpen, or it can commit data to disk on a per-handle basis with the DosBufReset function.

Fault Tolerance

The HPFS's extensive use of lazy writes makes it imperative for the HPFS to be able to recover gracefully from write errors under any but the most dire circumstances. After all, by the time a write is known to have failed, the application has long since gone on its way under the illusion that it has safely shipped the data into disk storage. The errors may be detected by the hardware (such as a "sector not found" error returned by the disk adapter), or they may be detected by the disk driver in spite of the hardware during a read-after-write verification of the data.

The primary mechanism for handling write errors is called a hotfix. When an error is detected, the file system takes a free block out of a reserved hotfix pool, writes the data to that block, and updates the hotfix map. (The hotfix map is simply a

series of pairs of doublewords, with each pair containing the number of a bad sector associated with the number of its hotfix replacement. A pointer to the hotfix map is maintained in the SpareBlock.) A copy of the hotfix map is then written to disk, and a warning message is displayed to let the user know that all is not well with the disk device.

Each time the file system requests a sector read or write from the disk driver, it scans the hotfix map and replaces any bad sector numbers with the corresponding good sector holding the actual data. This look aside translation of sector numbers is not as expensive as it sounds, since the hotfix list need only be scanned when a sector is physically read or written, not each time it is accessed in the cache.

One of CHKDSK's duties is to empty the hotfix map. For each replacement block on the hotfix map, it allocates a new sector that is in a favorable location for the file that owns the data, moves the data from the hotfix block to the newly allocated sector, and updates the file's allocation information (which may involve rebalancing allocation trees and other elaborate operations). It then adds the bad sector to the bad block list, releases the replacement sector back to the hotfix pool, deletes the hotfix entry from the hotfix map, and writes the updated hotfix map to disk.

Of course, write errors that can be detected and fixed on the fly are not the only calamity that can befall a file system. The HPFS designers also had to consider the inevitable damage to be wreaked by power failures, program crashes, malicious viruses and Trojan horses, and those users who turn off the machine without selecting Shutdown in the Presentation Manager Shell. (Shutdown notifies the file system to flush the disk cache, update directories, and do whatever else is necessary to bring the disk to a consistent state.)

The HPFS defends itself against the user who is too abrupt with the Big Red Switch by maintaining a Dirty FS flag in the SpareBlock of each HPFS volume. The flag is only cleared when all files on the volume have been closed and all dirty buffers in the cache have been written out or, in the case of the boot volume (since OS2.INI and the swap file are never closed), when Shutdown has been selected and has completed its work.

During the OS/2 boot sequence, the file system inspects the DirtyFS flag on each HPFS volume and, if the flag is set, will not allow further access to that volume until CHKDSK has been run. If the DirtyFS flag is set on the boot volume, the system will refuse to boot; the user must boot OS/2 in maintenance mode from a diskette and run CHKDSK to check and possibly repair the boot volume.

In the event of a truly major catastrophe, such as loss of the SuperBlock or the root directory, the HPFS is designed to give data recovery the best possible chance of success. Every type of crucial file object – including Fnodes, allocation sectors, and directory blocks – is doubly linked to both its parent and its children and contains a unique 32-bit signature. Fnodes also contain the initial portion of the name of their file or directory. Consequently, CHKDSK can rebuild an entire volume by methodically scanning the disk for Fnodes, allocation sectors, and directory blocks, using them to reconstruct the files and directories and finally regenerating the freespace bitmaps.

Application Programs and the HPFS

Each of the OS/2 releases thus far have carried with them a major discontinuity for application programmers who learned their trade in the MS-DOS environment. In OS/2 1.0, such programmers were faced for the first time with virtual memory, multitasking, interprocess communications, and the protected mode restrictions on addressing and direct control of the hardware and were challenged to master powerful new concepts such as threading and dynamic linking. In OS/2 Version 1.1, the stakes were raised even further. Programmers were offered a powerful hardware-independent graphical user interface but had to restructure their applications drastically for an event-driven environment based on objects and message passing.

In OS/2 Version 1.2, it is time for many of the file-oriented programming habits and assumptions carried forward from the MS-DOS environment to fall by the wayside. An application that wishes to take full advantage of the HPFS must allow for long, free-form, mixed-case filenames and paths with few restrictions on punctuation and must be sensitive to the presence of EAs and ACLs. After all, if EAs are to be of any use, it won't suffice for applications to update a file by renaming the old file and creating a new one without also copying the EAs.

But the necessary changes for OS/2 Version 1.2 are not tricky to make. A new API function, DosCopy, helps applications create backups – it essentially duplicates an existing file together with its EAs. EAs can

also be manipulated explicitly with DosQFileInfo, DosSetFileInfo, DosQPathInfo, and DosSetPathInfo. A program should call DosQSysInfo at run time to find the maximum possible path length for the system, and ensure that all buffers used by DosChDir, DosQCurDir, and related functions are sufficiently large. Similarly, the buffers used by DosOpen, DosMove, DosGetModName, DosFindFirst, DosFindNext, and like functions must allow for longer filenames. Any logic that folds cases in filenames or tests for the occurrence of only one dot delimiter in a filename must be rethought or eliminated.

The other changes in the API will not affect the average application. The functions DosQFileInfo, DosFindFirst, and DosFindNext now return all three sets of times and dates (created, last accessed, last modified) for a file on an HPFS volume, but few programs are concerned with time and date stamps anyway. DosQFsInfo is used to obtain volume labels or disk characteristics just as before, and the use of DosSetFsInfo for volume labels is unchanged. There are a few totally new API functions such as DosFsCtl (analogous to DosDevIOCtl but used for communication between an application and an FSD), DosFsAttach (a sort of explicit mount call), and DosQFsAttach (determines which FSD owns a volume); these are intended mainly for use by disk utility programs.

In order to prevent old OS/2 applications and MS-DOS applications running in the DOS box from inadvertently damaging HPFS files, a new flag bit has been defined in the EXE file header that indicates whether an application is HPFS-aware. If this bit is not set, the application will only be able to search for, open, or create files on HPFS volumes that are compatible with the FAT file system's 8.3 naming conventions. If the bit is set, OS/2 allows access to all files on an HPFS volume, because it assumes that the program knows how to handle long, free-form filenames and will take the responsibility of conserving a file's EAs and ACLs.

Summary

The HPFS solves all of the historical problems of the FAT file system. It achieves excellent throughput even in extreme cases – many very small files or a few very large files– by means of advanced data structures and techniques such as intelligent caching, read-ahead, and write-behind. Disk space is used economically because it is managed on a sector basis. Existing application programs will need modification to take advantage of the HPFS's support for extended attributes and long filenames, but these changes will not be difficult. All application programs will benefit from the HPFS's improved performance and decreased CPU use whether they are modified or not. *This article is based on a prerelease version of the HPFS that was still undergoing modification and tuning. Therefore, the final release of the HPFS may differ in some details from the description given here.*

B-Trees and B+ Trees

Most programmers are at least passingly familiar with the data structure known as a binary tree. Binary trees are a technique for imposing a logical ordering on a collection of data items by means of pointers, without regard to the physical order of the data.

In a simple binary tree, each node contains some data, including a key value that determines the node's logical position in the tree, as well as pointers to the node's left and right subtrees. The node that begins the tree is known as the root; the nodes that sit at the ends of the tree's branches are sometimes called the leaves.

To find a particular piece of data, the binary tree is traversed from the root. At each node, the desired key is compared with the node's key; if they don't match, one branch of the node's subtree or another is selected based on whether the desired key is less than or greater than the node's key. This process continues until a match is found or an empty subtree is encountered (Figure 7a).

Such simple binary trees, although easy to understand and implement, have disadvantages in practice. If keys are not well distributed or are added to the tree in a non-random fashion, the tree can become quite asymmetric, leading to wide variations in tree traversal times.

In order to make access times uniform, many programmers prefer a particular type of balanced tree known as a B-Tree. For the purposes of this discussion, the important points about a B-Tree are that data is stored in all nodes, more than one data item might be stored in a node, and all of the branches of the tree are of identical length (Figure 7b).

The worst-case behavior of a B-Tree is predictable and much better than that of a simple binary tree, but the maintenance of a B-Tree is correspondingly more complex. Adding a new data item, changing a key value, or deleting a data item may result in the splitting or merging of

a node, which in turn forces a cascade of other operations on the tree to rebalance it.

A B+ Tree is a specialized form of B-Tree that has two types of nodes: internal, which only point to other nodes, and external, which contain the actual data (Figure 7c).

The advantage of a B+ Tree over a B-Tree is that the internal nodes of the B+ Tree can hold many more decision values than the intermediate-level nodes of a B-Tree, so the fan out of the tree is faster and the average length of a branch is shorter. This makes up for the fact that you must always follow a B+ Tree branch to its end to get the data for which you are looking, whereas in a B-Tree you may discover the data at an intermediate node or even at the root.

Note 1: As used herein, "OS/2" refers to the OS/2 operating system jointly developed by Microsoft and IBM.

Note 2: "MS-DOS" refers to the Microsoft MS-DOS operating system and not to such products generally.

ABOUT THE AUTHOR

Ray Duncan is the president of Laboratory Microsystems Inc., a software house specializing in interpreters and compilers for the Forth programming language.

Laboratory Microsystems, Inc.
12555 West Jefferson Boulevard
Suite 202
Los Angeles, California 90066

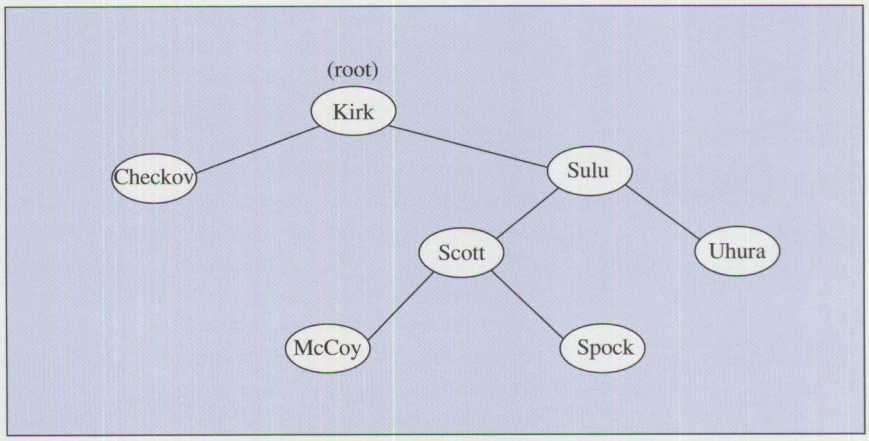


Figure 7a. To find a piece of data, the binary tree is traversed from the root until the data is found or an empty subtree is encountered.

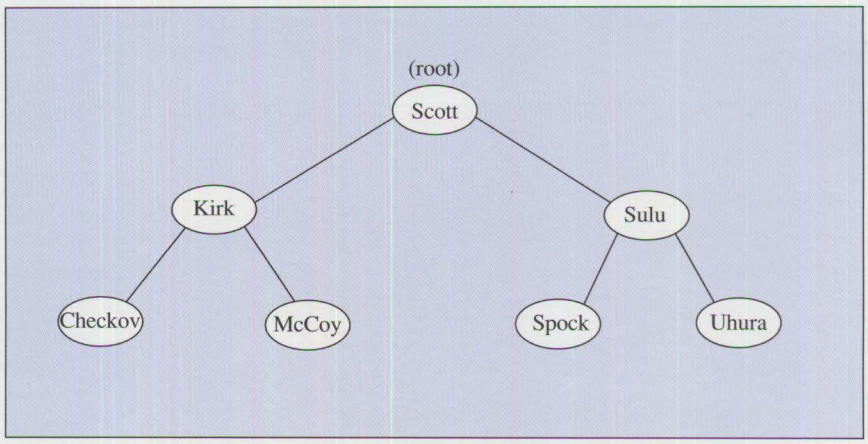


Figure 7b. In a balanced B-Tree, data is stored in nodes, more than one data item can be stored in a node, and all branches of the tree are the same length.

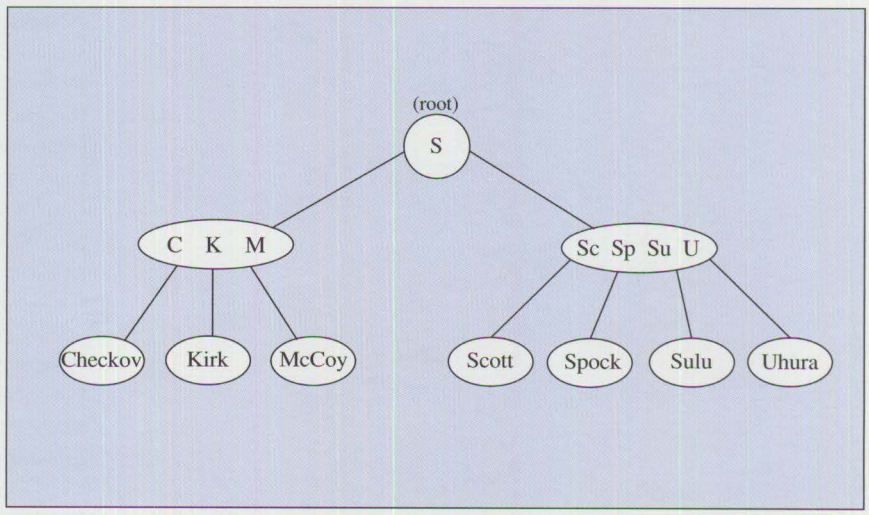


Figure 7c. A B+ Tree has internal nodes that point to other nodes, and external nodes that contain actual data.

OS/2 EE 1.2 Database Manager – Remote Data Services

Romelia Flores
IBM Corporation
Dallas, Texas

This article discusses the Communications Manager APPC configuration process for Remote Data Services. In addition, it contains information for installing Remote Data Services.

Database Manager consists of two major features: Database Services and Query Manager. Database Services is the engine or heart of the Database Manager. Database Services processes Structured Query Language (SQL) statements that are embedded within an application. It also contains a set of Application Programming Interfaces (APIs) that allow an application programmer to change the processing environment for Database Manager (configuration files, etc.), and gain access to utilities (import, export, etc.). Query Manager is the end-user interface provided with OS/2 Database Manager. Query Manager is itself an application program that interfaces with Database Services in order to externalize the functions of Database Services.

One of the major enhancements to the OS/2 Database Manager in Extended Edition 1.2 is the addition of Remote Data Services (RDS) components – RDS Requester and RDS Server. These components are part of the Database Services engine. They make it possible for an application program residing on an OS/2 workstation, or for the end user of such an application, to gain transparent access to data residing in an OS/2 Database on a remote workstation.

RDS uses the Advanced Program-to-Program Communication (APPC) interface of OS/2 Communications Manager (CM), thus providing support of several different communications media by simply adjusting CM profiles. IBM Token-Ring, IBM PC Network, Ethernet (DIX V2.0 and IEEE 802.3), X.25, and SDLC are supported. The use of APPC also paves the way for the implementation of Distributed Database Services to the other Systems Application Architecture (SAA) platforms.

Although the actual manner in which data is transported in the RDS environment is transparent to the end user, the installation and configuration for the RDS environment requires knowledge of APPC configuration. *The RDS Requester and RDS Server components do not use the Local Area Network (LAN) Requester component of OS/2 EE 1.2 or the LAN Server 1.2 product.* However, the LAN Requester or LAN Server can reside on the same LAN or workstation as the RDS Requester or RDS Server.

RDS is designed for multiuser access to a database. Therefore multi-

ple RDS Requesters can simultaneously access the same database on an RDS Server. A separate process is started on the RDS Server on behalf of each remote connection to a Server database. For example, if two programs on an RDS Requester connect to a database on the same RDS Server, two processes will be started. Database Manager enforces serialized execution of SQL statements at the thread level, not the process level; therefore, concurrent access is possible.

This article focuses on the Communications Manager APPC configuration process for RDS and provides other information for installing RDS. Information on APPC, specific APPC parameters, and how to configure Communications Manager for APPC is provided in the *IBM Operating System/2 Extended Edition Systems Administrator's Guide for Communications*, S01F-0261.

SQL LAN-Only Option

The implementation of RDS provides a special Transmission Service Mode named SQL LAN-Only Option (SQLLOO), which yields a performance enhancement for the LAN environments. SQLLOO does not apply to the SDLC or X.25 environments.

The Subsystems Management and remote operator facilities provided by Communications Manager are not available when SQLLOO is used. This means that a user will not be able to view or deactivate any SQLLOO sessions or link stations. Hence, the recommended approach for initially setting up RDS is to use APPC by creating a Transmission Service Mode profile (named anything other than SQLLOO) and specifying this Transmission Service Mode profile name in the Partner Logical Unit (LU) profile. When successful access to a remote database is achieved, the user can modify the Transmission Service mode profile and the Partner LU profile to specify SQLLOO. (The remote workstation must also be uncataloged and recataloged with the SQLLOO.)

Data Link Control (DLC) Considerations

When using a transmission mode other than SQLLOO for communication, the link stations defined in the DLC profile are used. When using SQLLOO for communication, the link stations defined in the DLC profile are not used. Instead, the SQLLOO uses link stations defined in the IEEE 802.2 profile. It uses 80% of the remaining link stations after subtracting the DLC and NETBIOS link stations. When configuring your system, make sure that the parameter settings for link stations provide some extra link stations that can be used by SQLLOO, because it is the last function loaded that requires link stations. For example, if the IEEE 802.2 profile has the link stations parameter set to 8, and your NETBIOS and DLC link stations are each set to 4, that does not leave any link stations available for SQLLOO. SQLLOO needs one link station available on the RDS Requester and one link station per RDS Requester available on the RDS Server.

APPC Configuration

Because RDS uses the APPC interfaces of OS/2 Communications Manager, APPC profiles will need to be configured appropriately. The profiles that need to be configured depend on whether the workstation is an RDS Requester or RDS Server. By configuring additional partner profiles, a Server may also perform Requester functions. In this article this configuration will be referred to as RDS Requester/Server. The profiles needed in each situation are shown in Figure 1. An "X" in the figure signifies that more than one of these profiles may be needed to gain access to multiple Servers, or to use different transmission service modes or initial session limits.

Some of the complexity of configuring the RDS environment is relieved through the use of Basic Configuration Services (BCS). During installation, BCS provides de-

Profile	RDS Requester	RDS Server	(Additional) RDS Server/ Requester
Workstation	1	1	
IEEE 802.2	1	1	
SNA Base	1	1	
DLC	1	1	
Local LU	1	1	
Partner LU	X	1	X
Transmiss. Service Mode	X	1	X
Initial Session Limits	X	0	X
Remotely Attached TPs	0	2	
Conversation Security	0	1	

Figure 1. CM Profiles for RDS

fault configuration files required to have one RDS Requester communicate with one RDS Server in the IBM Token-Ring environment. BCS determines which type of configuration file to create by prompting the user to specify whether each workstation will be an RDS Requester, RDS Server, or both Requester and Server. The CM Advanced Configuration Services can be used to modify the BCS configuration files for different connectivity, to access multiple RDS Servers, and to specify other changes to the environment.

Following is a description of the default values provided in the configuration files when configuring RDS via BCS for the IBM Token-Ring Network.

Workstation Profile: This profile contains information about the workstation, auto-start options, and message/error log sizes (Figure 2).

If RDS is configured to take advantage of the SQLLOO, and applications residing on this workstation do not require APPC, the Workstation Profile parameter to enable autostart options can be modified to not load Sytems Network Architecture (SNA)/APPC.

SNA Base Profile: This profile contains parameters pertaining to APPC (Figure 3).

The Physical Unit (PU) Name and Network Name are used by RDS when recording information in the OS/2 Error Log and generating alerts. If configuring strictly for the LAN environment, the PU Name and Network name parameters can be left with the default values. Otherwise, the naming scheme put in place by the Network Administrator should be used. Auto-Activate Attach Manager must be set to YES on the RDS Server workstation in order to accept requests initiated by RDS Requesters.

DLC Profile: This profile contains information about adapters being used for APPC or 3270 terminal emulation (Figure 4).

The DLC profile should be configured for the type of connectivity used for RDS. If SQLLOO is not used, the RDS Server should have one link station defined for every RDS Requester that accesses it.

NETBIOS Profile: This profile contains parameters pertaining to the NETBIOS interface (Figure 5).

The 802.2 link station value must be greater than or equal to the sum of the DLC link stations and the NETBIOS link stations. The maximum number of link stations in this profile should be taken into account during configuration.

IEEE 802.2 Profile: This profile contains specific information about the LAN logical link control (Figure 6).

Workstation Profile	BCS RDS Requester	BCS RDS Server
Services to load	SNA/APPC	SNA/APPC
Autostart	Yes	Yes

Figure 2. Workstation Profile

SNA Base Profile	BCS RDS Requester	BCS RDS Server
Autoact. Attach Manager	No	Yes
PU Name	PU000000	PU000000
Network Name	BLANK	BLANK

Figure 3. SNA Base Profile

DLC Profile /IBM TR Network DLC	BCS RDS Requester	BCS RDS Server
Maximum Link Stations	4	4

Figure 4. DLC Profile

NETBIOS Profile	BCS RDS Requester	BCS RDS Server
Maximum Link Stations	4	4

Figure 5. NETBIOS Profile

IEEE 802.2 Profile	BCS RDS Requester	BCS RDS Server
Adapter Number / Version	0 – 16/4 /A	0 – 16/4 /A
Adapter Address	XXXXXXXXXXXX	XXXXXXXXXXXX
Maximum Link Stations	12	41

Figure 6. IEEE 802.2 Profile

BCS configures this profile for use with the IBM Token-Ring Network 16/4 Adapter /A. BCS prompts the user to specify if the Universally Administered LAN adapter address is to be used. If the user specifies no, BCS prompts the user for the Local LAN adapter address (Locally Administered Address (LAA) of the workstation). If a different adapter type is being used, this profile should be modified appropriately. When modifying the default values for the Maximum Link Stations, take into account the previous section entitled "DLC Considerations."

LU Profile: This profile is used to configure an LU that will reside on the local workstation (Figure 7). The name specified when creating the profile will become the local LU Alias in this profile.

BCS creates a profile named LU1 for the RDS Requester and LU2 for the RDS Server. Both LU1 and LU2 are created on a workstation configured as an RDS Requester/Server. The appropriate Local LU Profile name and Partner LU Profile name should be used when cataloging the RDS Server workstation at the RDS Requester workstation.

Partner LU Profile: This profile is used to describe a path to configure a partner LU residing on a remote workstation (Figure 8). The name specified when creating the profile will become the partner LU Alias in the profile.

As mentioned earlier, the recommended approach when initially configuring the RDS environment is to use a Transmission Service Mode profile with a name other than SQLLOO or the implicit Transmission Service Mode. This name should be specified in the Partner LU Profile.

- RDS Requester

  BCS will create one partner LU profile and will prompt the user for the LAN destination address (LAA or Universally Administered Address of the partner workstation) to be specified in this profile.

  A Partner LU Profile is required on the RDS Requester for each RDS Server to be accessed. These can be created via Communications Manager's Advanced Configuration Services, using the BCS Partner LU as a model profile. When multiple applications on the RDS Requester will be accessing the same RDS Server, the Session Limit in the Partner LU profile should be increased by two for each additional concurrent connection. This value should be increased by two, because two Transaction Programs are executing on behalf of each RDS Requester.

- RDS Server

  Implicit Partner LU Profiles (indicated by a '*' at the beginning of the Partner LU Alias Name) can be used at the RDS Server so that requests can be accepted from any RDS Requester. Explicit Partner LU Profiles would require a profile for each RDS Requester which would gain access to this RDS Server. The Server has the Partner LU session limit set to the maximum in order to allow as many Requesters as possible to gain access. The implicit Transmission Service Mode *SQLLOO is also used. If *SQLLOO is not used, subsystem management allows the view-

	BCS RDS Requester	**BCS RDS Server**
LU Alias	LU1	LU2
LU Name	LU1	LU2
Default LU	No	No
LU Local Address	00	00
LU Session Limit	255	255
Maximum Number of TPs	0	0

Figure 7. LU Profiles

	BCS RDS Requester	**BCS RDS Server**
Partner LU Alias	LU2	*PLU
Fully Qualified Name	BLANK.LU2	BLANK.BLANK
LU Alias	LU1	LU2
Net. Adap. Numb./ LAA	XXXXXXXXXXXX	000000000000
PLU Session Limit	1	255
Conversation Security	Yes	Yes
CS Verified	No	No
Transmission Service Mode	SQLLOO	*SQLLOO
Initial Session Limits	LU1ISL	BLANK

Figure 8. Partner LU Profiles

ing of active sessions, with the Partner LU Alias Names appearing as @I000000, @I000001, etc.

- RDS Requester/Server

  Both LU2 and *PLU profiles are created on a workstation configured as an RDS Requester/Server.

Transmission Service Mode Profile: This profile contains session characteristics for a specific conversation (Figure 9). The name specified when creating the profile becomes the Mode name.

BCS creates a Mode name of SQLLOO on the RDS Requester and *SQLLOO on the RDS Server. Again, the recommended approach for initially configuring the RDS environment is to create a new profile with a name other than SQLLOO or *SQLLOO. SQLLOO or *SQLLOO may be used as the model profile for setting up these new profiles and then deleted to avoid minor inconsistencies from occurring if they are not referenced in any Partner LU Profiles.

Both SQLLOO and *SQLLOO are created on a workstation configured as an RDS Requester/Server. BCS sets the session limit to 1 in the transmission service mode profile to avoid starting the SNASVCMG mode to negotiate the session limit.

Initial Session Limits Profile: This profile contains parameters used during Node/Link activation (Figure 10).

BCS creates an Initial Session Limits Profile named LU1ISL. If SQLLOO is not utilized, the value for automatically started sessions can be adjusted to automatically start a session during Node/Link activation instead of waiting to start the session until the application performs a "Start Using" for a particular database. When using the SQLLOO, the initial session limits cannot be modified via Communications Manager's Subsystem Management.

LU1ISL is also created on a workstation configured as an RDS Requester/Server.

Remotely Attached TPs: These are used to describe transaction programs (TPs) that can be started from a remote workstation (Figure 11).

BCS creates two Remotely Attached TP Profiles corresponding to the TPs provided for RDS on the RDS Server. The profile names provided by BCS for the two RDS Transaction Programs are SQLAPPLA and SQLSNMGR. The profile names can be modified and the filespecs should be set with the drive letter where Database Man-

	BCS RDS Requester	BCS RDS Server
Mode Name	SQLLOO	*SQLLOO
Minimum RU Size	256	256
Maximum RU Size	1920	1920
Receive Pacing Limit	4	4
Session Limit	1	255

Figure 9. Transmission Service Mode Profiles

	BCS RDS Requester
ISL Profile Name	LU1ISL
Minimum Number Cont. Winners Source	0
Maximum Number Cont. Winners Target	0
Number of Automatically Started Sessions	0

Figure 10. Initial Session Limits Profile

BCS RDS Server		
TP Profile	**SQLAPPLA Profile**	**SQLSNMGR Profile**
Service TP	Yes	Yes
TP filespec	X:\SQLLIB\SQLCIAA.EXE	X:\SQLLIB\SQLCNSM.EXE
First character	07	07
TP name	6DB	6SN
Sync level	None	None
Conv. type	Basic	Basic
Conv. Sec. Req.	Yes	Yes
TP operation	Non–queued attach started	Non–queued attach started
TP rcv timeout	0	0
Program type	Background	Background
TP startup parms	BLANK	BLANK

Figure 11. Remotely Attached TPs

ager resides. The RDS TPs are Service TPs that take advantage of the flexibility provided by the BASIC conversation verbs and therefore have a BASIC conversation type. These TPs run in the background and require no user intervention at the RDS Server.

SQLAPPLA is the Server Application Agent TP which connects to a database at the RDS Server on behalf of the RDS Requester.

SQLSNMGR is the Server Node Manager TP. The RDS Requester starts a second conversation when Ctrl+Break is entered to transmit an interrupt request to the RDS Server. This interrupt request will cause RDS at the Server to start the SQLSNMGR which "cleans up" the currently active database operation for that process, rolling it back if possible.

Notice that the RDS TPs are both non-queued; therefore, multiple instances of these TPs can be concurrently active. Thus, multiple RDS Requesters can gain access to an SQL Database concurrently. There are a couple of ways to limit the number of occurrences of the RDS TPs. These include setting the Maximum number of TPs in the LU profile, or setting the Maximum number of concurrent connections to a particular database (MAXAPPLS configuration parameter) in the Database Configuration file.

The *Information and Planning Guide for OS/2 Extended Edition 1.2*, G360-2650, can be used to deduce the maximum number of RDS Requesters that can attach to an RDS Server.

For example, if you have a 16 MB RDS Server and you know that 4.5 MB of memory is required for the base Operating System, CM (APPC), and DBM, you will have 11.5 MB of memory left for RDS Requesters. Since 0.18 MB is required per remote database connection, the maximum number of remote connections is 64.

The amount of memory required for each additional remote database connection will be changed in the next version of the OS/2 Information and Planning Guide to 0.15 MB. When this new number is used, then a recalculation yields a maximum of 76 remote connections.

These numbers represent the "optimum" amount of memory required, and they also represent a medium workload. These calculations do not include the amount of memory required for a DOS Compatibility Box, 3270, Gateway, Query Manager, LAN Requester, or any other applications executing in the system.

These numbers strictly represent the Database Manager limitations. The Communications Manager limitations, such as the maximum number of active link stations, should also be taken into account.

User Profile Management and Conversation Security: User Profile Management (UPM) is a new component of the OS/2 Extended Edition 1.2 Base Operating System. UPM is utilized to define unique user IDs and group IDs (collection of user IDs) on each workstation. These IDs are allowed access to that particular workstation, and are then used by other OS/2 Extended Edition components to perform extra security functions.

User IDs and group IDs are defined by accessing User Profile Management Services from the Desktop Manager, selecting User Profile Management, and selecting Manage Users/Group from the Manage pull-down screen. The user can also access UPM via the OS/2 Command Prompt by specifying "UPMACCTS."

UPM user IDs are used by DBM to prevent unauthorized access to a database and its contents. Database Manager has new levels of authority and privileges replacing the old database password. Authorities include SYSADM and DBADM. Privileges are granted to a particular user ID, group ID, or to public via the GRANT and REVOKE SQL statements.

User IDs are used to logon or logoff a user to or from a particular workstation. When Query Manager is started, the user is automatically prompted for a user ID and password. The user may also logon or logoff by accessing User Profile Management Services from the Desktop Manager and selecting LOGON or LOGOFF; or via the OS/2 Command Prompt by specifying "LOGON userid /P=password" or "LOGOFF userid".

A user may specify a default user ID and password to be utilized at the next connection to a particular node by specifying "LOGON userid /P=password /N=node". A user can also log off a particular node by specifying "LOGOFF userid /N=node."

The default user ID and password supplied upon installation is 'USERID' and 'PASSWORD'. The default group ID defined is 'GROUPID' and it contains the user ID 'USERID'.

Because Database Manager security is based on UPM IDs, UPM security should be used instead of APPC

Conversation Security. APPC Conversation Security is used to link to UPM by setting the first column of the first user ID equal to '*' and the remaining Conversation Security IDs equal to BLANK.

The RDS Server should contain an ID in UPM for each user who will access databases residing on this workstation.

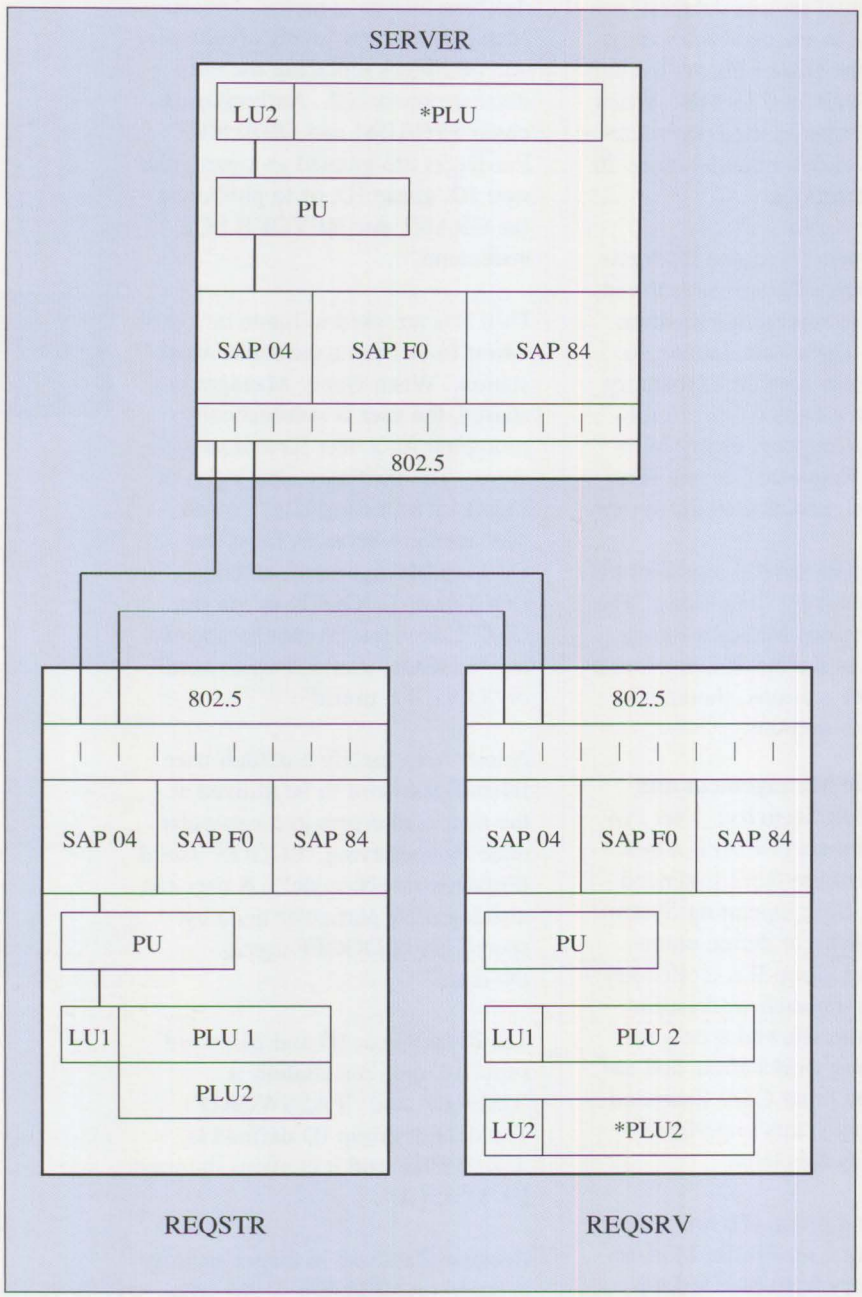


Figure 12. Sample Configuration

Sample Configuration

Figure 12 is a representation of three workstations configured for RDS. The configuration for each workstation is as follows: one workstation is an RDS Server (SERVER), one workstation is an RDS Requester (REQSTR) that will need access to two RDS Servers, and one workstation is an RDS Requester/Server (REQSRV) which has access to both local and remote databases.

SERVER has one local LU defined and uses an implicit partner LU in order to permit multiple RDS requesters to gain access to databases residing on this workstation.

REQSTR has one local LU defined with two partner LUs in order to gain access to multiple RDS Servers. PLU1 is associated with SERVER and PLU2 is associated with REQSRV.

REQSRV has two local LUs defined. LU1 is utilized as a Requester LU which has a partner LU of PLU2 (SERVER). LU2 is used as an RDS Server LU with an implicit partner LU in order to permit multiple RDS Requesters to gain access to databases residing on this workstation.

If SQLLOO is used as the transmission service mode, then link stations associated with Service Access Point (SAP) 84 are used.

Database Manager Structure

OS/2 Database Manager 1.2 (like the other OS/2 Extended Edition components) can be installed on any logical fixed disk drive by using the OS/2 Common Install facility. Installation options provide for the installation of both Database Services and Query Manager, just Database

Services, or Query Manager (which requires that Database Services has already been installed). The installation of the RDS components is part of the Database Services installation. Database Manager determines which components of Database Services to install by the responses given by the user during the installation procedure. For example, the question "Will other workstations need access to databases on your workstation?" allows Database Manager to determine whether or not the RDS Server component should be installed.

The SQLLIB subdirectory contains Database Manager executables along with other necessary information such as the Database Manager Configuration File, the subdirectory containing the Database System Directory, and the subdirectory containing the Database Manager Node Directory.

During database creation the user has the ability to specify a particular fixed disk drive for the database. A new subdirectory is created on the fixed disk (volume) specified called SQLxxxxx, where xxxxx is a number such as 00001 which uniquely identifies this subdirectory to Database Manager. Another subdirectory called SQLDBDIR contains the Database Volume Directory for this particular fixed-disk drive.

Databases are created locally at the RDS Server workstation, and then the RDS Requester can access them remotely. A database directory cannot be created remotely.

The tree diagram in Figure 13 shows the structure of subdirectories created by Database Manager at installation and database creation. The X represents any logical fixed-disk drive. Database Manager may reside on a different logical fixed-disk drive than actual databases.

Node Directories

When installing OS/2 Database Manager 1.2 for RDS, the user is prompted to supply a node name or workstation name. This node name is stored in the Database Manager configuration file as Workstation Name. Node names are used by Database Manager to tie together information placed in the Database Node Directory and System Database Directory for a particular workstation. In the RDS DOS Requester environment, node names are used to identify a workstation to NETBIOS.

The Database Node Directory (SQLNODIR) contains information such as the Local LU, Partner LU, and Mode which reference the LU Alias, Partner LU Alias, and Transmission Service Mode profiles respectively.

When a reference to a particular database is made, Database Manager scans the System Database Directory and determines the entry type for that database. If the entry

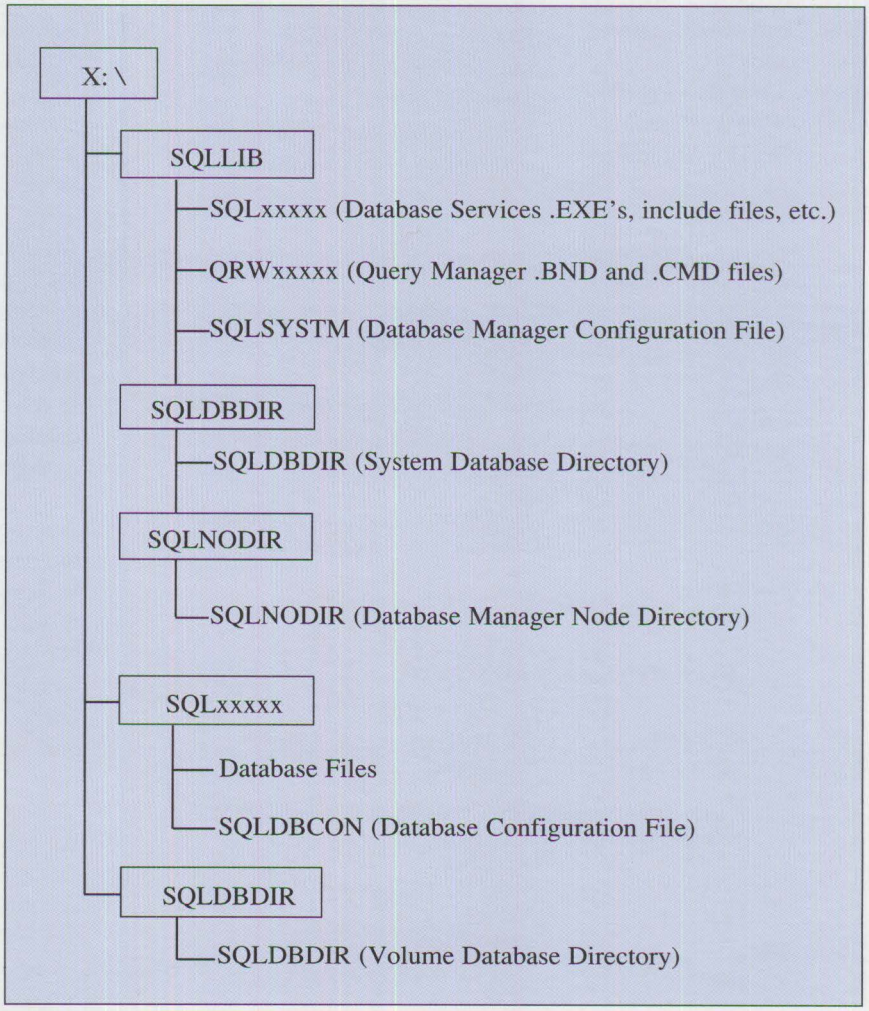


Figure 13. Database Manager Subdirectory Structure

type is "remote," then Database Manager looks at the "node-name" specified and proceeds to scan the Database Node Directory in order to determine which communications profiles should be used to gain access to the remote workstation.

Entries are made to the Database Node Directory when a node is cataloged. Interfaces for cataloging a node are provided via Query Manager and the Database Services API (sqlgcatn, sqlguncn).

When using RDS, a node must be cataloged on the RDS Requester workstation. When cataloging the node on the RDS Requester, specify the Node Name (workstation name specified at installation) for the remote workstation, Local LU Profile Name, Partner LU Profile Name, and Transmission Service Mode Profile name.

Only users defined as local administrators or administrators can catalog nodes.

The sample node entry in Figure 14 uses the BCS default profiles and defines a workstation as RDSN01 (RDS Node 01).

Database Directories

There are two types of Database Directories: System Database Directory and Volume Database Directory. The system directory resides in a subdirectory named SQLDBDIR inside the SQLLIB subdirectory. This directory will exist on every type of database workstation – standalone, RDS Requester, RDS Server, or RDS Requester/Server. This directory provides the information necessary for Database Manager to determine if a database is local or remote.

Node Directory Information	RDS Requester Contents
Node Name	RDSN01
Local LU	LU1
Partner LU	LU2
Mode Name	SQLLOO
Comment	Default
Comment Codepage	Default

Figure 14. Sample Node Directory

Database Directory Information	RDS Requester	RDS Server	
	System	System	Volume
Database Name	SAMPLE	SAMPLE	SAMPLE
Drive	N/A	C	C
Alias	SAMP01	SAMPLE	SAMPLE
Entry Type	REMOTE	INDIRECT	HOME
Comment	Default	Default	Default
Comment Codepage	Default	Default	Default
Node Name	RDSN01	N/A	N/A

Figure 15. Sample Database Directory

The volume database directory resides on each drive where a database is created. It is kept in a subdirectory named SQLDBDIR. This directory provides the information necessary for Database Manager to locate a database on a particular drive.

Both the system and volume database directories contain a number of fields: database name, drive, alias, entry type, comment, comment codepage, and node name.

The entry type field contains a value of home, indirect, or remote. Home entry type can only exist in a volume database directory. Indirect and remote entry types can only exist in a system database directory.

A home entry type indicates that the database resides on the same drive as the volume database directory.

An indirect entry type indicates that the database resides locally and the system database directory contains a reference to the correct volume database directory for that database. In the corresponding volume database directory entry, the entry type for this database will be home.

A remote entry type indicates that the database resides on a remote workstation. The node name for this database corresponds to a node name entry in the Node directory which provides the information needed by Database Manager to gain access to the remote workstation.

An entry is made to the System Database Directory when a database is cataloged. Interfaces for cataloging a database are provided via Query Manager and the Database Services API (sqlgcatd, sqlguncd). The volume database directory does

not have an interface, but is maintained by Database Manager when local databases are created or dropped.

In order to utilize RDS to access a remote database, the database must be cataloged on the RDS Requester as remote. Database Manager will take care of cataloging the database as local on the RDS Server when the database is created and the volume database directory will contain an entry type of home.

Only users defined as local administrators or administrators can catalog databases.

The sample database directory in Figure 15 shows the different Database Directories and the information required on both the RDS Requester and RDS Server in order to gain access to the database remotely. The node name specified on the RDS Requester ties this entry to the previous sample node directory entry for the appropriate communication information.

Performance Considerations

Database Manager contains two configuration files with parameters that should be taken into account when tuning Database Manager and the RDS Environment.

Database Configuration: The Database Configuration File (SQLDBCON) contains parameters that pertain to resources allocated for a particular database. Therefore, each database that is created contains its own database configuration file. Interfaces for modifying this configuration file are provided via Query Manager and the Database Services API (sqlgeudb). Some of the parameters that affect performance are:

- Buffer pool size (buffpage) – The buffer pool contains database records which have been read and changed. Records remain in the buffer pool until the database is no longer in use, a commit is issued, or the space is needed in the buffer pool for other records. Updated records are then written to disk.

- Sort list heap (sortheap) – This area contains private segments allocated per application program for sort buffers for each sort (ORDER BY, DISTINCT) in the application. A large sort buffer improves performance of sorting operations.

Database Manager Configuration: The Database Manager Configuration File (SQLSYSTM) contains parameters that pertain to resources allocated across all databases on a system. Therefore, only one Database Manager configuration file exists per workstation, and it is stored in SQLLIB. Interfaces for modifying the Database Manager configuration file are provided via Query Manager and the Database Services API (sqlgusys). Some of the parameters that affect performance are:

- Maximum number of remote connections (numrc) – Used to estimate storage required for CM buffers.

- Communication heap size (comheapsz) – Allocated on the RDS Requester and RDS Server for each requesting application. Record blocks are allocated from this heap. Whether or not blocking is to be used is specified during the precompile or bind of an application. A block is allocated for each cursor that is blocked. Record blocking affects performance because it reduces network traffic by buffering resulting data. Record blocking will only occur if specified at precompile or bind, if comheap storage is available, and if a cursor is read-only. Because comheap is allocated for each requesting application, set comheap size for the process with the greatest number of open cursors.

- Maximum requester I/O block size (rqrioblk) – One block (of size rqrioblk) is allocated in the comheap of the RDS Requester when database connection occurs. This block is used for communication between the RDS Requester and RDS server. This block is freed upon database connection termination.

  Additional blocks are allocated for each record blocking cursor in the application. The rqrioblk value is used to determine the record block size on both the RDS Requester and RDS Server. These blocks are freed when the associated cursor is closed.

- Maximum server I/O block size (svrioblk) – One block (of size svrioblk) is allocated in the comheap of the RDS Server when database connection occurs. This block is used for communication between the RDS Requester and RDS Server. This block is freed upon database connection termination.

- RDS heap size (rsheapsz) – Allocated on both RDS Requester and RDS Server. It provides a workarea containing information for managing comheap row blocks. One heap is allocated per application program on the RDS Requester. One heap is allo-

cated per remote application program on the RDS Server.

Starting RDS

The RDS environment is ready for utilization once:

- Appropriate Communications Manager configuration files have been configured and verified.
- Database Manager has been configured.
- Databases have been created and cataloged as appropriate.
- The applications to be executed remotely have been bound or are dynamically bound during execution into the RDS Server database.

  In an RDS environment, the distribution of an application differs in that a bind file (.BND) or a database is not required on each workstation. For remote database access the database exists at the RDS Server. .BND files may exist at either an RDS Requester or an RDS Server, depending on which workstation the bind is performed at. Access of the application (.EXE) file from each requesting workstation is necessary.
- User IDs have been created in UPM at the RDS Server (and at the RDS Requester if Query Manager is being used), and appropriate privileges have been granted.

The following steps should be performed on the RDS Server:

- Start Communications Manager
- Start the Database Manager by issuing the STARTDBM command at the OS/2 Command Prompt.

The following steps should be performed on each RDS Requester:

- Start Communications Manager
- Start the Database Manager by issuing the STARTDBM command at the OS/2 Command Prompt. This step is not required if the application performs this function. Query Manager performs this function.
- Start the application.

Conclusion

The SAA relational function provided by RDS is OS/2 to OS/2 "Remote Unit of Work". Remote Unit of Work allows access to a single Relational Database Management Systems within a unit of work. Access to different Relational Database Management Systems may be obtained with separate units of work.

Distributed Unit of Work allows access to multiple Relational Database Management Systems within a unit of work and requires a two-phase commit. Because OS/2's APPC does not implement Syncpoint, a two-phase commit has not been implemented for OS/2 Database Manager. For a description of the different SAA Relational Database phases, refer to announcement letters for the Distributed Relational Data in SAA (288-545) and IBM Relational Productivity Family Enhancements Overview (288-546).

As access to the other Relational Database Management Environments such as DB2, SQL/DS and OS/400 Database Manager is developed, the objectives of transparent access to data, performance, location autonomy, system managed integrity, and security will be emphasized. RDS implements these objectives through the use of of APPC and SQLLOO, node and database catalog techniques, and association of authorities/privileges with UPM user IDs.

ABOUT THE AUTHOR

Romelia Flores is an advisory market support representative in the OS/2 EE Technical Support department in the Desktop Systems Support Center. She received a computer science degree from the University of Texas at Austin. Romelia joined IBM in 1982 as a programmer for the IBM Displaywriter system in the Electronic Document Distribution area and the IBM 3270 Emulation Program in the Server-Requester Programming Interface area. She is currently focusing on the Remote Data Services feature of OS/2 EE 1.2 and the C language interface for Database Services.

OS/2 EE Database Manager Precompiler API

Nancy Miller
IBM Corporation
Dallas, Texas

In OS/2 Database Manager, a "precompiler" is a tool that translates SQL statements embedded in the code of a high-level language into compilable code containing calls that permit the program to access the Database Manager. The precompiler Application Programming Interface (API) provides a way for the programmer to write a precompiler to support any high-level language. This API extends the power of the programming languages by giving them greater database function, and extends the accessibility of the Database Manager.

Structured Query Language (SQL) is the programming interface for defining and manipulating OS/2 Extended Edition Database Manager data. It is used by embedding SQL statements into a program written in a high-level language (referred to as a "host language"). However, since SQL is not a part of any standard high-level language, a precompiler is needed to translate SQL statements into compilable "host language" statements.

To meet the requirements of each particular host language, a separate precompiler must be provided for each language. OS/2 Extended Edition 1.1 Database Manager supplies a precompiler for the IBM C/2™ language. Three additional languages are supported in OS/2 EE 1.2: IBM COBOL/2, IBM Pascal/2™, and IBM FORTRAN/2™. OS/2 EE also has an application program interface (API) to Database Manager's Precompiler Services. This enables precompilers to be written for any other language – languages that may be needed for your application development, but for which IBM has not provided a precompiler.

A precompiler can be written in any high-level programming language, and certainly does not have to be written in the language that it will be processing. The task of designing the precompiler will be easier if the language chosen supports string manipulation, pointers, dynamic allocation of variables, and user-defined data structures. Most important, the language used must have the capability to call Precompiler Services.

Process Overview

The discussion of the precompiler API begins with an overview of the precompilation process. Then each phase of the process and the structures involved will be covered in more depth.

The function of the Database Manager precompiler is divided into two parts so the processing of a host language program can be separated from the processing of the SQL statements embedded within the program. These two parts are:

- The language-specific precompiler function (hereafter called "the precompiler")

- Precompiler Services (a set of routines provided by Database Manager)

The precompiler first scans the program, searching for SQL statements embedded in the host language. When an SQL statement is found, it is "preprocessed," which means that any language-specific matter (such as comments), and any host variables and non-blank white spaces (such as carriage returns and line feeds) are removed. The statement is then passed to Precompiler Services. Precompiler Services processes the statement, returning the completion status to the precompiler along with instructions for making changes to the original source file.

After the precompiler has processed the original program, a new program is produced called the "modified source." The original SQL statements may remain in the program as comments, and code is inserted by the precompiler to make calls to Database Manager Run-time Services. Run-time Services is a group of routines that access Database Services, the part of Database Manager that actually builds and modifies databases. All SQL statements can be translated into some combination of calls to Run-time Services routines, which will be examined more closely later.

An access plan and a bind file may also be products of the precompilation process, depending on the precompiler options utilized. An access plan contains the compiled form of the embedded SQL statements and is stored in the database specified when the precompiler was invoked. There is an access plan associated with every precompiled source module. If an access plan is created at precompile time, the application can be run only against the database specified during precompilation.

Alternately, the creation of an access plan can be deferred and a bind file created instead. The bind file contains SQL statements and host variable definitions from the precompiled program. It is language independent, and will permit the application to be bound later to other databases without recompiling.

Figure 1 illustrates the flow of the precompilation process.

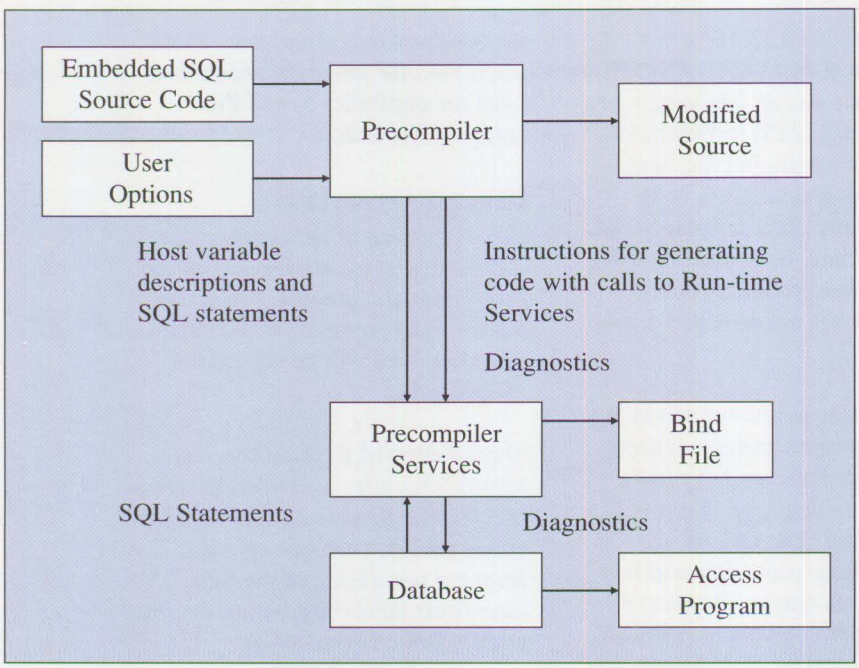


Figure 1. Precompilation Process

Division of Tasks

There are numerous tasks involved in turning a database application (for example, one containing SQL statements) into actions by Database Manager against a database. These tasks are divided into a number of steps, from the application program to the Database Manager.

Application Program: The application program must:

- Contain correct SQL statements

- Obtain any necessary user information

Precompiler: The responsibilities of the precompiler are to:

- Manage host variables

- Translate the source file into a modified source file in the host language, by doing the following:

  — Initially process SQL statements before sending them to Precompiler Services, removing any host language-specific matter

  — Include original SQL statements as comments in modified source

— Create any required data structures

— Respond to information returned by Precompiler Services to construct Run-time Services function calls to be inserted into modified source

Precompiler Services: The responsibilities of Precompiler Services are to:

- Validate each SQL statement, check authority of users to issue the statement, compile it, and store it in an access section (which will later become part of an access plan)
- Identify the tasks required to carry out the SQL statement and create a list of tasks called a task array (the task array is used by the precompiler in building Run-time Services function calls)
- Determine how all host variables are used and provide a usage code for each variable
- Build either an access plan, which is the sum of all the access sections; or, if deferred binding is chosen, store the processed SQL statements in a bind file
- Pass coded diagnostic information to the precompiler

Run-time Services: Run-time Services provides support for communicating with Database Services. Its responsibilities are to:

- Initialize and validate the SQL Communications Area (SQLCA) and SQL Description Area (SQLDA)
- Modify the input and output SQLDA
- Call Database Services to process compiled SQL statements
- Pass dynamic SQL statements to Database Services

Database Services: Database Services responds to calls made by Run-time Services, executing sections of an access plan to make the database changes being requested.

Data Structures

How is information passed between the executing program, the precompiler, Precompiler Services, and Run-time Services? Much of it is passed using specific standard data structures.

SQLCA: The SQL Communications Area (SQLCA) is used to send diagnostic information from the Database Manager to the host program. One of the primary pieces of information it contains is the SQLCODE, an integer return code signifying the completion status of a function call. It is interpreted as shown in Figure 2.

The rest of the SQLCA contains more specific diagnostic information, depending on the contents of the SQLCODE.

All Precompiler Services routines use the SQLCA, so it must be defined and allocated by the precompiler before calling Precompiler Services. The return codes in the SQLCA should be checked after every Precompiler Services function.

SQLDA: The SQL Data Area (SQLDA) is used to transfer data values between Run-time Services and the host program at run time. The precompiler does not manage this structure directly; this task is done by Run-time Services. However, for every host variable found in an SQL statement, the precompiler must provide the following information for use in the SQLDA:

- SQLTYPE – an assigned number that describes the data type of the host variable and whether or not it can contain nulls
- SQLLEN – the length of a variable in bytes; or, in the case of decimal type, the precision and scale
- SQLDATA – a pointer containing the address of a variable
- SQLIND – a pointer containing the address of a null indicator variable if one is used

Token Array: The precompiler must assign a token ID for each host variable. It is a four-byte integer and must be unique for each

Code	Meaning
0 (zero)	Successful execution (there may be warning conditions)
Positive	Successful execution, but with an exception condition
Negative	Error condition

Figure 2. SQLCODE

host variable. Precompiler Services refers to variables using the token ID.

The Token Array is a one-dimensional array of logically paired four-byte integers. The first pair is the header, which contains both the number of pairs available in the array for token IDs and the number of pairs that actually contain tokens. The remaining pairs in the array contain the token ID as the first element, and a usage code as the second element.

There is a token ID in the array for every instance of a variable in an SQL statement. It must be filled in by the precompiler. The precompiler must also supply the number of tokens in the array as the second element in the header.

The usage codes are filled in by Precompiler Services after a compile request. They are listed in Figure 3.

Usage Code	Meaning
SQLA_INPUT_HVAR (0)	Input host variable
SQLA_INPUT_WITH_IND (1)	Input host variable with indicator variable
SQLA_OUTPUT_HVAR (2)	Output host variable
SQLA_OUTPUT_WITH_IND (3)	Output host variable with indicator variable
SQLA_INDICATOR (4)	Indicator variable
SQLA_INVALID_USE (5)	Host variable does not match use
SQLA_USER_SQLDA (6)	User-defined SQLDA name
SQLA_INVALID_ID (7)	Host variable token ID is not valid

Figure 3. Usage Codes from the Token Array

Function Flag	Action Generated
SQLA_START	Generate a call to SQLASTRT.
SQLA_DECLARE	Begin or end parsing host variable declarations.
SQLA_INCLUDE	Generate code for an SQLCA or SQLDA
SQLA_ALLOC_INPUT	Allocate an input SQLDA using SQLAALOC.
SQLA_ALLOC_OUTPUT	Allocate an output SQLDA using SQLAALOC.
SQLA_SETS	Register a host variable using SQLASETS.
SQLA_USDA_INPUT	Register an input user-defined SQLDA.
SQLA_USDA_OUTPUT	Register an output user-defined SQLDA.
SQLA_CALL	Generate a call to SQLACALL.
SQLA_DEALLOC	Generate a call to SQLADLOC.
SQLA_STOP	Generate a call to SQLASTOP.
SQLA_SQLERROR	Generate code for WHENEVER SQLERROR.
SQLA_SQLWARNING	Generate code for WHENEVER SQLWARNING.
SQLA_NOT_FOUND	Generate code for WHENEVER NOT_FOUND.

Figure 4. Function Flags from the Task Array

Task Array: Like the Token Array, the Task Array is a one-dimensional array of logically paired four-byte integers, and the first logical pair serves as the header. It contains the number of pairs available in the Task Array (a value provided by the precompiler) and the number of pairs being used (a value filled in by Precompiler Services after an SQL statement is compiled).

The remainder of the array contains the tasks that the precompiler must complete. This list of tasks is returned by Precompiler Services after it processes an SQL statement. The first element is a function flag that represents a function the precompiler must perform; the second element contains data necessary to perform the function. Figure 4 is a list of the function flags and the precompiler actions they generate.

Option Array: The Option Array is used by the precompiler to pass options to Precompiler Services. It contains an eight-byte header followed by several pairs of four-byte integers. The header contains the amount of space allocated for the array and the number of options actually used. The remainder of the array contains option information regarding date and time format, isolation level, record blocking, and

information regarding whether or not an access plan and a bind file are to be generated.

Initialization Tasks

Before the precompiler actually begins to process a source program, it must perform several initialization tasks. These tasks include:

- Defining an SQLCA
- Installing a control-break handler so that if the user decides to terminate precompilation by pressing Ctrl+Break, the program will terminate properly
- Processing command line arguments, which include the source file name, modified source file name, bind file name, database name, access plan name, and any user options
- Opening the source file for input and the modified source file for output
- Preparing the Option Array
- Calling the Precompiler Services routine SQLAINIT to initialize the precompilation process
- Testing the return code from SQLAINIT
- Processing the program ID – a string of characters inserted into the modified source will be used for coordination between the executable program and access plan at run time

SQLAINIT is one of four Precompiler Services routines. (The other routines are SQLAAHVR, SQLACMPL, and SQLAFINI.) SQLAINIT initializes the Precompiler Services data structures and must be completed before calling any other Precompiler Services routines. It accepts as input the access plan name, database name, password, bind file name (if any) and the Option Array. It also issues a START USING DATABASE. An SQLCA containing any warnings or errors is returned to the precompiler, along with the program ID.

The precompiler maintains a list of all host variables.

Processing the Source File

After initialization has successfully completed, the precompiler is ready to begin processing the source file. It begins by scanning the file, copying everything verbatim to the modified source until it encounters an SQL statement. When SQL statements are found, there are a number of tasks to be performed. All comments, host variables, and non-blank white spaces (tabs, carriage-returns, line-feeds) must be replaced by blanks (X'20') on a character-by-character basis. Any quoted strings must be recognized by the precompiler and left untouched.

Usually, all host variables are declared between BEGIN DECLARE SECTION and END DECLARE SECTION statements. The precompiler maintains a list of all host variables, registering each host variable with Precompiler Services by using the call to SQLAAHVR (Add Host Variable). The name, length, datatype, and token ID assigned to the variable are sent as parameters to SQLAAHVR.

When the precompiler identifies host variables in an SQL statement and replaces them with blanks, it must then look up each variable's token ID in its list of host variables, and place the token ID in the Token ID Array. The colon that precedes host variables in SQL statements is left in so that Precompiler Services can identify the position of host variables. As a final step in preprocessing the SQL statement, the precompiler adds a spare byte, which will be used in compiling the statement.

When the statement and the Token ID Array are prepared, the precompiler invokes the Precompiler Services SQLACMPL (Compile) routine. Precompiler Services compiles the SQL statement by parsing it and determining exactly what tasks must be performed to carry out the statement. If there are no errors, it stores the statement in a bind file (if one is being created) and creates the Task Array. It also determines exactly how each host variable is being used, and places the usage code in the Token ID array for each occurrence of a variable.

The precompiler then checks the SQLCODE in the SQLCA for errors or messages. If no errors occurred, the precompiler begins checking the task array to see what tasks must be performed in the modified source in order to execute the SQL statement. The precompiler must then generate code in the host language to include the required Run-time Services function calls.

There are eight Run-time Services function calls. Each SQL statement can be broken down into some com-

bination of these calls. They include:

- SQLASTRT – starts run-time statement execution, and is always required. SQLASTRT obtains a semaphore to make sure thread access is serialized in order to enforce synchronization.

- SQLAALOC – allocates an input or output SQLDA. It is used if the statement contains host variables.

- SQLASETV – used only if the statement contains host variables, and follows the call to SQLAALOC. It initializes fields of an SQLDA SQLVAR element.

- SQLAUSDA – sets a pointer to the address of a user-defined input or output SQLDA. It is used in the execution of dynamic SQL statements.

- SQLASETS – also used to execute dynamic SQL statements, this call sets a pointer to the length and address of a host variable used to store the text of the statement.

- SQLACALL – a function that calls Database Services with the specific access section to execute. It is the one function that actually accesses Database Services and is always used for executing an SQL statement.

- SQLADLOC – deallocates SQLDAs previously allocated by SQLAALOC.

- SQLASTOP – marks the end of the execution of the SQL statement. It releases the semaphore originally acquired by SQLASTRT.

After completing processing of the source file (or if a fatal error occurs), the precompiler terminates Precompiler Services with a call to SQLAFINI. Once this call is issued, no other calls can be accepted by Precompiler Services until another SQLAINIT is issued. SQLAFINI saves or discards the access program and bind file, if one was created, and returns information via the SQLCA, indicating whether or not the access program and bind file were saved.

Assuming no errors occurred, the modified source file is now completed and ready to be compiled in the host language.

For additional information on the Database Manager precompiler API, see the following publications:

IBM OS/2 Extended Edition V1.1 Database Manager Programming Guide and Reference, 90X7772.

IBM OS/2 Extended Edition V1.2 Database Manager Programming Guide and Reference, S01F-0269.

ABOUT THE AUTHOR

Nancy Miller is an associate market support representative in the OS/2 EE technical support group in Dallas. She is a member of the Database Manager support team. She also provides technical support for GDDM-PCLK, Graphics Workstation Program and GDDM-OS/2 Link. Nancy has a B.A. in art and English and is completing an M.S. in mathematics and computer science.

UNION, INTERSECT, EXCEPT

Patty Brayton
IBM Corporation
Dallas, Texas

This article is intended as a tutorial for the relational database operators UNION, INTERSECT, and EXCEPT. Not only will the theoretical concepts be discussed, but also the manner in which these functions are implemented in OS/2 Extended Edition Database Manager. Included in this article will be examples of each of the operators.

It is assumed that the reader understands basic relational database concepts, especially relations (referred to in this article as "tables"), keys, and how the data is stored in a relational database (attributes and tuples, referred to as "columns" and "rows" here).

For instructional purposes, the tables in Figure 1 are used for each of the examples in this article.

UNION and UNION ALL

UNION is most commonly used to combine the results from two or more separate queries that would individually create a "result table." When the UNION operator is used, the result consists of all the rows (meeting the criteria of the SELECT statement) appearing in any or all of the individual result tables. Using the UNION operator, two result tables, T1 and T2, are combined to create a new result table that contains the set of all rows in T1 or T2 (or both) that meet the specified conditions in the initial queries (with any duplicate rows eliminated).

This could, of course, be expanded to include more than two tables to be UNIONed. *Note:* In OS/2 Extended Edition Database Manager's implementation, the tables being UNIONed together must SELECT an equal number of columns, and the corresponding columns must be the same type. The columns of the UNION table are not given names by Query Manager. When Query Manager is used to perform the query, the column headings are labeled "EXPRESSION 1," "EXPRESSION 2," etc., rather than "LASTNAME," "ID," and so forth.

The UNION operator can be represented by the diagram in Figure 2.

Notice that the resulting table, represented by T1 UNION T2, contains all the data in both T1 and T2.

EMPLOYEE_INFO TABLE

LASTNAME	INITIAL	ID	DEPT
BROWN	E.	12345	H57
BROWNING	E.	78342	J34
POLK	J.	43567	J34
BOVER	D.	87634	N76
JOHNSON	F.	85461	N76
SMYTHE	W.	45342	J34

DEPT_INFO TABLE

LASTNAME	ID	DEPT	MANAGER
BROWN	12345	H57	Y
BROWNING	78342	J34	N
POLK	43567	J34	Y
BOVER	87634	N76	Y
JOHNSON	85461	N76	N
SMYTHE	45342	J34	N

PERS_INFO TABLE

LASTNAME	ID	MARRIED	NUM_OF_DEPS
BROWN	12345	N	1
BROWNING	78342	Y	3
POLK	43567	N	0
BOVER	87634	Y	5
JOHNSON	85461	N	1
SMYTHE	45342	Y	2

Note: In all of the tables, ID is defined as a character field.

Figure 1. Example Tables

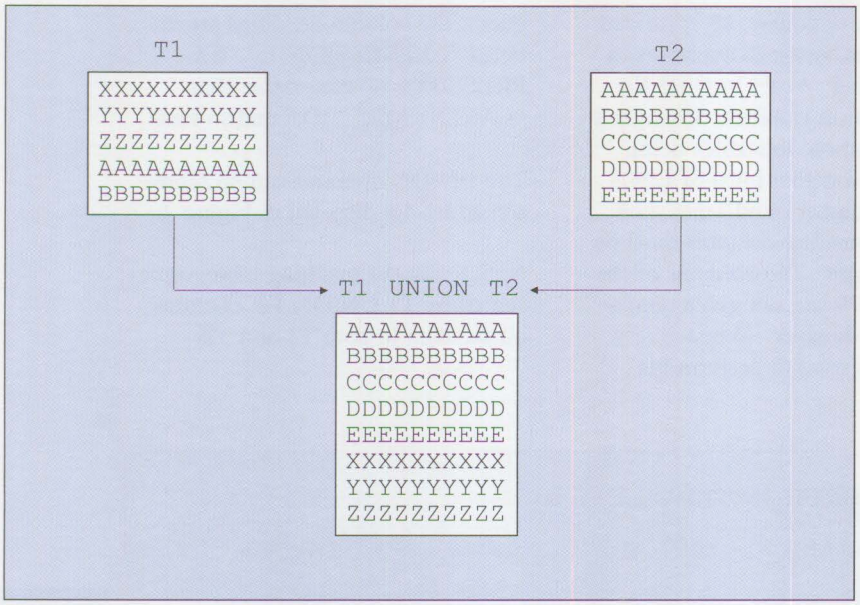


Figure 2. The UNION Operator

Here are some examples to help clarify how the UNION operator works.

Example 1: The following query will display all the last names of employees who are managers:

```
SELECT LASTNAME FROM DEPT_INFO
   WHERE MANAGER = 'Y'
```

If this query is run, the following is displayed:

LASTNAME
BROWN
POLK
BOVER

If a list of all non-married employees is needed, the following query could be used:

```
SELECT LASTNAME FROM PERS_INFO
   WHERE MARRIED = 'N'
```

Running this query would cause the following to be displayed:

LASTNAME
BROWN
POLK
JOHNSON

Now, the two SELECT statements can be UNIONed together to form a single query to produce a list of all employees that are either managers or are not married:

```
SELECT LASTNAME FROM DEPT_INFO
   WHERE MANAGER = 'Y'
UNION
SELECT LASTNAME FROM PERS_INFO
   WHERE MARRIED = 'N'
```

the result table would be:

EXPRESSION 1
BOVER
BROWN
JOHNSON
POLK

Note that the items in the result table are in alphabetical order. One of the consequences of using the UNION operator is that to remove duplicate rows, the rows in the result table are sorted in ascending order.

Example 2: Suppose that a query is needed to find all the married people (in all departments) and all the people who work in Department J34 (married or not). The following query performs the desired function:

```
SELECT LASTNAME, ID FROM
   EMPLOYEE_INFO
         WHERE DEPT = 'J34'
UNION
SELECT LASTNAME, ID FROM
   PERS_INFO
         WHERE MARRIED = 'Y'
```

When this query is executed, the result table is:

EXPRESSION 1	EXPRESSION 2
BOVER	87634
BROWNING	78342
POLK	43567
SMYTHE	45342

Example 3: In the previous examples, the columns that were SELECTed were the same in both statements joined by the UNION operator. This is not required as long as corresponding columns are the same data type. For example,

```
SELECT LASTNAME FROM PERS_INFO
   WHERE NUM_OF_DEPS > 2
UNION
SELECT ID FROM DEPT_INFO
   WHERE MANAGER = 'Y'
```

would result in:

EXPRESSION 1
12345
43567
87634
BOVER
BROWNING

UNION ALL

Recall that the UNION function causes deletion of any duplicate rows that would appear as a result of the operation. UNION ALL does not do this. When UNION ALL is used, any duplicate rows that occur appear in the final table. As a result, the rows in the table are not sorted, but rather appear in the order in which they were entered into the original tables. Consider Example 1 again, but with the UNION ALL operator this time.

```
SELECT LASTNAME FROM DEPT_INFO
   WHERE MANAGER = 'Y'
UNION ALL
SELECT LASTNAME FROM PERS_INFO
   WHERE MARRIED = 'N'
```

This query produces the following result table:

EXPRESSION 1
BROWN
POLK
BOVER
BROWN
POLK
JOHNSON

Note the duplicates and the lack of sorting that occurs with the UNION ALL operator.

INTERSECT

INTERSECT is another set operator that can be used to merge two or more tables in a relational database. When INTERSECT is used, the result table consists of rows appearing in *all* of the specified tables. (Recall that UNION result tables consist of rows appearing in any or all of the specified tables.) In other words, the intersection of two tables, T1 and T2, is the set of all rows belonging to both T1 *and* T2. INTERSECT (like UNION) will cause any duplicate rows to be eliminated from the final result table. *Note:* In OS/2 Extended Edition Database Manager's implementation, the tables to be INTERSECTed must SELECT an equal number of columns, and the corresponding columns must be the same type. The columns of the INTERSECT table are not given names by Query Manager. Again, similar to UNION, the column headings are displayed simply as "EXPRESSION 1," "EXPRESSION 2," etc., when Query Manager is used to perform the query.

The INTERSECT operator can be represented by the diagram in Figure 3.

The result table (T1 INTERSECT T2) contains only those rows contained in *both* T1 and T2.

To demonstrate the difference between UNION and INTERSECT, the examples used in the UNION section of this article have been modified to utilize the INTERSECT operator in the following examples:

Example 4: Recall from the UNION example, the query:

```
SELECT LASTNAME FROM DEPT_INFO
   WHERE MANAGER = 'Y'
```

generates the following result table:

LASTNAME
BROWN
POLK
BOVER

The query:

```
SELECT LASTNAME FROM PERS_INFO
   WHERE MARRIED = 'N'
```

generates:

LASTNAME
BROWN
POLK
JOHNSON

When the two SELECT statements are INTERSECTed, the following query is created:

```
SELECT LASTNAME FROM DEPT_INFO
   WHERE MANAGER = 'Y'
INTERSECT
SELECT LASTNAME FROM PERS_INFO
   WHERE MARRIED = 'N'
```

which generates the result table:

EXPRESSION 1
BROWN
POLK

Notice that only the names found in both the first and the second result tables are included in the INTERSECT table, and that they are alphabetized. As in UNION, the INTERSECT function causes the data rows in the result table to be sorted in ascending order so any duplicate rows can be deleted. Therefore, each row found in a result table generated by the INTERSECT operator is unique.

Example 5: The query:

```
SELECT LASTNAME, ID FROM
   EMPLOYEE_INFO
         WHERE DEPT = 'J34'
INTERSECT
SELECT LASTNAME, ID FROM
   PERS_INFO
         WHERE MARRIED = 'Y'
```

causes the following table to be generated:

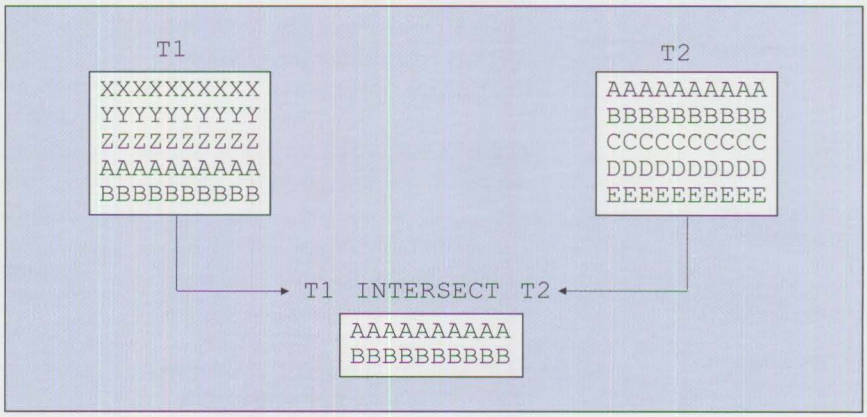


Figure 3. The INTERSECT Operator

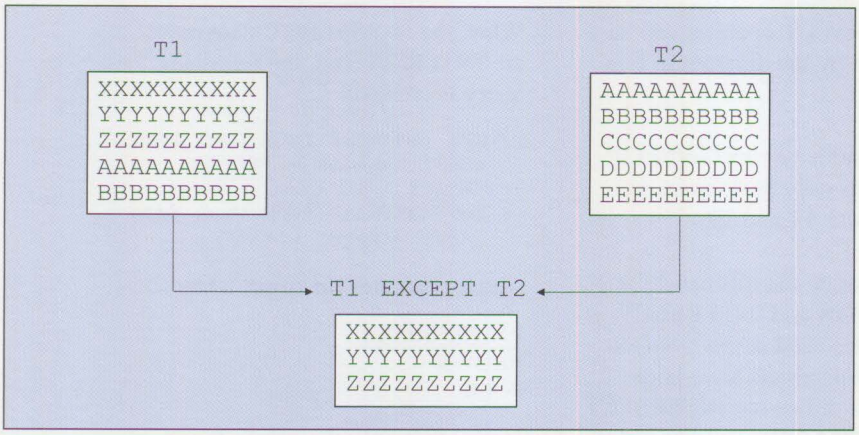


Figure 4. The EXCEPT Operator

EXPRESSION 1	EXPRESSION 2
BROWNING	78342
SMYTHE	45342

Again, notice that the rows found in the INTERSECTed table are only those that are present in *both* tables generated by the SELECT statements.

Example 6: Unlike the UNION operator, the columns SELECTed in each of the statements INTERSECTed are required to be the same. For example, if the following query is executed:

```
SELECT LASTNAME FROM PERS_INFO
   WHERE NUM_OF_DEPS > 2
INTERSECT
SELECT ID FROM DEPT_INFO
   WHERE MANAGER = 'Y'
```

Query Manager then displays this empty table:

EXPRESSION 1

Whereas, the following query:

```
SELECT LASTNAME FROM PERS_INFO
   WHERE NUM_OF_DEPS > 2
INTERSECT
SELECT LASTNAME FROM DEPT_INFO
   WHERE MANAGER = 'Y'
```

generates the non-empty table:

EXPRESSION 1
BOVER

This makes sense because the INTERSECT operator is defined as generating only those rows of data that appear in *both* INTERSECTed tables.

EXCEPT

Given two tables T1 and T2, the EXCEPT operator generates a result table that consists of all rows in table T1, but *not* in table T2. That is, EXCEPT corresponds to the relational operator DIFFERENCE.
Note: When the EXCEPT operator is used between two SELECT statements, the two statements must SELECT the same number of columns, and the corresponding columns must be the same type. The selected column(s) will be given a heading of "EXPRESSION 1," "EXPRESSION 2," etc., (by Query Manager) as when using the UNION or INTERSECT operators.

The EXCEPT operator can be represented by the diagram in Figure 4.

The result table represents T1 EXCEPT T2 – all the rows contained in table T1, but not contained in T2.

Example 7: This query generates a table that contains all employees who are married.

```
SELECT LASTNAME FROM PERS_INFO
   WHERE MARRIED = 'Y'
```

The table looks like this:

LASTNAME
BROWNING
BOVER
SMYTHE

To generate a list of all non-managers, use the following query:

```
SELECT LASTNAME FROM DEPT_INFO
   WHERE MANAGER = 'N'
```

The generated table is:

LASTNAME
BROWNING
JOHNSON
SMYTHE

Now, to find out how many married people are non-managers, use this query:

```
SELECT LASTNAME FROM PERS_INFO
   WHERE MARRIED = 'Y'
EXCEPT
SELECT LASTNAME FROM DEPT_INFO
   WHERE MANAGER = 'Y'
```

The result:

EXPRESSION 1
BROWNING
SMYTHE

Notice that the condition has been changed in the second SELECT statement (from 'N' to 'Y'). Remember that all rows generated by the second SELECT statement are deleted from the table generated by the first SELECT statement to form the final result table. To find all the non-managers, we want to have a table with all managers deleted. EXCEPT generates a table in which all rows are in the first table but *not* in the second.

Example 8: This query produces a list of all non-married employees who work in department J34:

```
SELECT LASTNAME, ID FROM
   EMPLOYEE_INFO
         WHERE DEPT = 'J34'
EXCEPT
SELECT LASTNAME, ID FROM
   PERS_INFO
         WHERE MARRIED = 'Y'
```

The report generated looks like this:

EXPRESSION 1	EXPRESSION 2
POLK	43567

Example 9: To have a meaningful result, the columns in both SELECT statements should be the same. The rows generated by the second SELECT statement are deleted from the rows generated by the first SELECT statement. Consequently, if the columns SELECTed are different, nothing is deleted from the first result table. See the following example for clarity:

```
SELECT LASTNAME FROM PERS_INFO
   WHERE NUM_OF_DEPS > 2
EXCEPT
SELECT ID FROM DEPT_INFO
   WHERE MANAGER = 'Y'
```

The table generated by the first SELECT statement is:

LASTNAME
BROWNING
BOVER

The table produced by the second SELECT statement is:

ID
12345
43567
87634

Obviously, there are no rows in the second table that are also in the first table, so nothing is deleted from the first table. Therefore, the EXCEPT table generated is:

EXPRESSION 1
BOVER
BROWNING

Notice that in all these examples the output is sorted in ascending order so any duplicate rows can be deleted.

Additional Information

For additional information, the following resources are recommended:

- C. J. Date, *An Introduction to Database Systems*, Volume 1, Addison Wesley.
- *IBM Operating System/2 Extended Edition Database Manager SQL Reference*
- *IBM Operating System/2 Extended Edition Database Manager Programming Guide and Reference*
- *IBM Systems Application Architecture Common Programming Interface Database Reference*

ABOUT THE AUTHOR

Patty Brayton is an associate market support representative in the OS/2 EE Desktop Systems Technical Support Center in Dallas. She is a member of the Database Manager Support Team. Before coming to IBM in 1988, she was an instructor of electrical engineering and computer science at the University of Oklahoma. Patty received her bachelor's degree in mathematics from the University of Oklahoma, her master's degree in computer science education from Central State University in Edmond, Oklahoma, and has done additional graduate work in computer science at the University of Oklahoma.

Writing a Database Manager COBOL/2 Program

Chris Jagger
IBM Corporation
Dallas, Texas

This article describes the interface provided for writing IBM COBOL/2 applications to gain access to the IBM OS/2 EE 1.2 Database Manager (DBM). It also includes a sample of such an application.

An IBM COBOL/2 application program interfacing to the OS/2 EE DBM follows the same basic format, with appropriate divisions, as any COBOL/2 application program.

The IDENTIFICATION (spelled out) DIVISION is required. The name of the program, PROGRAM-ID, must be stated here. Optional information such as Author, Date Written, Date Compiled, and so forth, can be included. This information is treated as comments by the compiler.

The ENVIRONMENT DIVISION, which contains information relating to the physical characteristics of the machine being used, is not required. It may be useful, however, to include the CONFIGURATION SECTION to document the SOURCE-COMPUTER and the OBJECT-COMPUTER for the program.

The DATA DIVISION contains the descriptions of all the data used in a program. The WORKING-STORAGE SECTION can be thought of as the work space for a program. Storage is allocated according to the data described in this section. The LINKAGE SECTION describes external data to be made available to the program from another program (the calling program), but does not allocate the storage. The storage is allocated in the calling program.

The PROCEDURE DIVISION holds all executable statements that direct the processing logic of a program.

Data is moved between a program and an OS/2 database through the use of Structured Query Language (SQL) and host variables. SQL is the language used to access data in a relational database. When a program contains any SQL statements, it must be processed by a precompiler. It is the procompiler that replaces the SQL statements with the appropriate COBOL/2 function calls, thus producing a modified source file that is ready to be compiled.

Host Variables

Host variables are variables referenced in an SQL statement. Just as there are specific "types" of data allowed in the columns of a database, there are also specific SQL data "types" allowed to describe host variables. The SQL data types supported by the COBOL/2 interface to OS/2 EE 1.2 Database Manager are listed in Figure 1.

No user-defined data types are allowed. Neither arrays nor the FLOAT column type are supported in COBOL/2. Level number 77 may be used interchangeably with 01 except when declaring VARCHAR and LONG VARCHAR data types.

Numerics

When using the PIC clause to define a numerical host variable, both the "S" (signed) and the "9" (number) must be used. If the "S" is left off, the following error message

DBM Column Type	SQL Type	Description	COBOL/2 Data Type
SMALLINT	500	16-bit, signed integer	01 NAME PIC S9(4) COMP-5.
INTEGER	496	32-bit, signed integer	01 NAME PIC S9(4) COMP-5.
DECIMAL(p,s)	484	Packed decimal	01 NAME PIC S9(n)V9(m) COMP-3.
CHAR(n) 1 <= n <= 254	452	Fixed-length character string	01 NAME PIC X(n).
VARCHAR(n) 1 <= n <= 4000	448	Varying-length character string	01 NAME. 49 NAME-1 PIC S9(4) COMP-5. 49 NAME-2 PIC X(n).
LONG VARCHAR 1 <= n <= 32700	456	Long varying-length	01 NAME. 49 NAME-1 PIC S9(4) COMP-5. 49 NAME-2 PIC X(n).

Figure 1. SQL Data Types Supported

will be displayed from the precompiler: "The token '9' found in a host variable declaration is not valid." The program will not precompile.

COMP should also be used as shown. COMP-5 differs from COMP (or COMPUTATIONAL, meaning BINARY format) in that the values stored are not limited to the number of decimal digits indicated in the PICTURE (PIC) clause, but to the largest binary number for which the allocated storage has space.

Decimals

When defining a DECIMAL(p,s), the following rules apply:

Sample:

```
01 NAME PIC S9(n)V9(m) COMP-3.
```

- The value for 's' equals the value for 'm'
- The value for 'p' equals the value for 'n' + 'm'
- The value for 'm' is optional
- Values for 'n' and 'm' must be greater than or equal to zero (0)
- Value for 'n' + 'm' cannot exceed 31
- V denotes the decimal point and must be used
- PACKED-DECIMAL can be used instead of COMP-3

Variable-Length Strings

A variable-length string consists of a collective name, a length item, and a value item. See VARCHAR(n) and LONG VARCHAR(n) in Figure 1. However, when referring to a variable-length string in an SQL statement, use the collective name.

VARCHAR(n), LONG VARCHAR(n), as well as CHAR(n), can be used to describe database columns with the FOR BIT DATA attribute. These columns can hold binary data or other forms of data not used by Database Manager. This gives an application, such as a graphics package, the ability to store and access its data using the database. The application could use the data for its own purposes, such as displaying an image.

Date, Time and Timestamp

DATE, TIME and TIMESTAMP data types are represented internally to Database Manager as a series of packed decimal digits. When retrieved, Database Services converts them to fixed-length character strings (Figure 2).

Null Columns

The OS/2 EE Database Manager has a special way of dealing with nullable columns, and it does this with null indicators. A null indicator is required for each host variable used to hold data from a nullable column. The null indicator is always two-bytes long.

Example:

```
01 NCOLUMN  PIC X(5).
01 NCOLUMN-I PIC S9(2) COMP-5.
```

If the null indicator contains a negative integer, the column value is null. If, however, the null indicator contains a zero or positive integer, the corresponding column value is not null. The null indicator and its associated host variable act as a pair. They do not need to be declared together (as they are in the preceeding example), but they must be referenced together in an SQL statement. Place the null indicator variable adjacent to the host variable column field, separated by one or more spaces.

Example:

```
EXEC SQL SELECT nullcol INTO
  :ncolumn :ncolumn-i
END-EXEC.
```

Naming Host Variables

Though it is the precompiler that identifies these host variables and prepares them for use in exchanging data with an OS/2 database, the COBOL/2 compiler also sees them, so certain rules must be followed in naming these variables:

- Use the correct COBOL/2 syntax
- Variable names can be up to 30 characters in length
- SQL and EXEC are reserved
- Word FILLER not allowed
- Hyphens (without spaces) are allowed

DBM Column Type	SQL	Description	COBOL/2 Data Type
DATE	384	10-byte character string	01 NAME PIC X(10).
TIME	488	8-byte character string	01 NAME PIC X(8).
TIMESTAMP	392	26-byte character string	01 NAME PIC X(26).

Figure 2. DATE, TIME and TIMESTAMP Data Types

Declaring Host Variables

All host variables need to be declared in an SQL DECLARE SECTION. An SQL DECLARE SECTION is simply a group of host variable declarations preceded by a BEGIN DECLARE SQL statement and followed by an END DECLARE SECTION SQL statement.

Example:

```
EXEC SQL
  BEGIN DECLARE SECTION
    END-EXEC.
01 ID      PIC S9(4).
01 JOB     PIC X(5).
01 YEARS PIC S9(4)
                    COMP-5.
01 SALARY PIC S9(7)V9(2)
                    COMP-3.
EXEC SQL
  END DECLARE SECTION
    END-EXEC.
```

The following rules apply:

- End each host variable declaration with a period
- Figurative constants (ZERO, SPACE) not allowed
- COBOL COPY not supported
- Follow COBOL/2 continuation rules to continue lines or strings
- Valid clauses allowed: PICTURE, USAGE, VALUE
- Use X as picture character, S9 for numeric
- Standard COBOL/2 comments allowed

SQL Data Type Summary

The precompiler determines the appropriate SQL data type when it finds a host variable DECLARE SECTION. Database Manager uses this data type to convert the data exchanged between the program and Database Services. To avoid abends, use the data descriptions in the charts provided.

Database Services Function Calls (APIs)

Besides defining the host variables, variables needed for calls to the Database Services Application Program Interfaces (APIs) must also be declared in the DATA DIVISION. These variables are not declared in an SQL DECLARE SECTION.

One such call is START USING DATABASE. Each program accessing an OS/2 database must use this call to connect to a particular database. Figure 3 shows a sample DATA DIVISION declaring the necessary variables when using the START USING DATABASE API followed by the syntax used to call the API from the PROCEDURES DIVISION.

Each Database Services function call should be preceded with two underscores. Then the compiler uses fully segmented addresses when passing "by reference" parameters. However, Query Manager will accept function calls without the underscores.

For each DBM API, the *OS/2 EE 1.2 Database Manager's Programming Guide and Reference, Volume II*, S01F-0269, lists the variable information as required, as well as the syntax for applications executing in the OS/2 PC environment and in the DOS environment (APIs supported by the DOS Database Requester).

When defining variables for the DBM APIs, all pointers are four-bytes long, and all two-byte numeric variables used as value parameters must be declared as COMP-5.

A list of the various APIs supported in COBOL/2 for both an OS/2 EE 1.2 environment and a DOS Database Requester environment is given in Figure 4.

```
DATA DIVISION.
77 SPARE1       PIC 9(4)       COMP-5 VALUE 0.
77 DB-LENGTH    PIC 9(4)       COMP-5 VALUE 6.
77 SPARE2       POINTER.
77 DATABASE     PIC X(5)       VALUE "SAMPLE"

* Put character in first byte.
77 D-USE        PIC 9(4)       COMP-5.
77 U            PIC X REDEFINES D-USE.
. . .
PROCEDURES DIVISION.
* Use indicator - S for Shared, X for Exclusive
MOVE "S" TO U.
CALL "__SQLGSTRD" USING DATABASE
                        SPARE2
                        SQLCA
                        BY VALUE D-USE
                        BY VALUE DG-LENGTH
                        BY VALUE SPARE1.
```

Figure 3. START USING DATABASE API

DBM Include Files

The last items in the DATA DIVISION to focus on are the DBM Include Files. These files contain necessary function definitions and error codes for the various environment, utility and miscellaneous commands, as well as system constants. The actual files used in a program will depend on the functions used in that program. Figure 5 describes each file.

There are two ways to define these DBM structures in an application: COPY or EXEC SQL INCLUDE statement. A COPY statement can be used on any of these files; an INCLUDE statement can be used only on SQLDA/SQLCA. Deciding which method to use for SQLDA/SQLCA depends on when the program needs the information. An EXEC SQL statement is resolved at precompile time while the COPY statement is resolved at compile time. Use the COPY statement to include the other files.

Procedure Division

The PROCEDURE DIVISION is where data is processed and manipulated. It is here that executable SQL statements are embedded into COBOL/2 code and calls are made to the Database Services APIs.

Some rules to keep in mind here are:

- Precede function calls with two underscores (_ _).
- Host variables in an SQL statement must begin with a colon (:).
- Statements can begin in Area A (8-11) or Area B (12-72) *Note: To follow Systems Application Architecture™ (SAA™) guidelines, use Area B.*

Environment Routines Supported	OS/2 Environment	RDS DOS Requestor
Start/Stop Database Manager	X	
Start/Stop Using Database	X	X
Database Restart	X	X
Log On/Off	X	X
Install Signal Handler	X	
Interrupt	X	X
Create/Drop Database	X	
Change Database Comment	X	X
Catalog/Uncatalog Database	X	X
Catalog/Uncatalog Node	X	
Migrate Database	X	X
Open/Close Directory Scan	X	X
Open/Close Node Scan	X	
Get Next Directory Entry	X	X
Get Next Node Entry	X	
Dereference Address	X	X
DB Appl. Remote Interface	X	X

Utility Routines Supported	OS/2 Environment	RDS DOS Requestor
Backup/Restore	X	
Import/Export	X	X
Runstats	X	X
Reorganize Table	X	X
Update DBM/DB Config	X	
Reset DBM/DB Config	X	
Return Copy DBM/DB Config	X	
Collect Database Status	X	
Get User Status	X	
Get Next DB Status Block	X	
Free DB Status Resources	X	
Get Administrative Authority	X	X
Get Address	X	X

Miscellaneous Routines Supported	OS/2 Environment	RDS DOS Requestor
Bind	X	
Retrieve Error Message	X	X

Figure 4. Supported APIs

SQL – Contains all the system constants for the DBM, SQLCA and SQLDA.

SQLENV – Contains structures, constants, and return codes for the DBM environment commands.

SQLUTIL – Contains structures, constants, and return codes for the DBM utility commands.

SQLCODES – Defines the possible values that can be returned to the SQLCODE variable in the SQLCA.

SQLCA – Defines the SQL Communications Area structure. This structure holds information about the most recently executed function call or SQL statement. The value of the SQLCODE variable in the SQLCA is used to determine whether an SQL statement or function call executed successfully or not. This file should be included in all programs and the SQLCODE should be checked after each statement or call:

SQLCODE =	0	Successful execution
	positive	Successful, but with a warning
	negative	Error condition
	100	Not found

The following sample shows how to evaluate the value of the SQLCODE to determine what action to take if 1) the SQL statement completed successfully, 2) the end of the table is reached, or 3) any other value appears:

```
EVALUATE SQLCODE  WHEN 0 CONTINUE
                  WHEN 100 PERFORM MOVE "1" TO END-OF-TABLE-SW
                                   GO TO 100-EXIT END-PERFORM
                  WHEN OTHER PERFORM DBERROR
END-EVALUATE.
```

SQLDA – Defines the variable-length SQL DESCRIPTOR AREA structure. This structure is used to pass data between Database Services and the application program. It is most often used with Dynamic SQL where the SQL statement is not fully known when the application is written or the objects referenced by the SQL statement do not exist at precompile.

The information in the SQLDA depends on its use. In a PREPARE and DESCRIBE statement, the SQLDA provides information to an application about the prepared statement. In an OPEN, EXECUTE, and FETCH statement, the SQLDA describes the host variables that can be used to allocate the storage needed.

DSQCOMM – This structure is required when a program uses the Query Manager Callable Interface and provides two services. It receives return code, reason code and message ID information from Query Manager (much like the SQLCA structure). Each call must check the DSQ_RETURN CODE to determine the success of the call:

0 =	Successful execution
4 =	Successful execution, but with a warning
8 =	Command did not execute correctly
16 =	Severe error, QM session terminated

The DSQCOMM also provides a working storage area for the Query Manager Callable Interface. A DSQCOMM structure is necessary for each QM session started.

Figure 5. Database Manager Include Files

- SQL statements follow the same line continuation rules as COBOL/2. However, the keyword pair EXEC SQL must be on one line.

Precompiling the Program

Once the program is written, use SQLPREP to precompile, making certain to include the file extension for COBOL (.SQB) on the command line. In this step, the precompiler converts the SQL statements into a form that can be compiled and turns each SQL statement into a comment. The precompiler creates a new file with a .CBL extension.

The program can be bound to the database at this time, or the bind can be deferred. If the program is bound during precompile, it can be bound to only one database. If, however, the bind is deferred, the program can later be bound to more than one database. Another advantage to deferring the bind is that when it is time to create a new access plan, after a REORG and RUNSTATS, only the bind will need to be redone. Otherwise an application would need to be precompiled, compiled and linked again.

Example:

```
SQLPREP INCREASE.SQB SAMPLE /B
```

This command precompiles the program INCREASE deferring the bind (see *Note 1* at the end of this article). Note that even though the bind is being deferred, the database name is still required or the precompile will not be successful. This example creates a .CBL and a .BND file with the name INCREASE.

Compiling and Linking the Program

After the precompile, the application may be compiled using any necessary compiler directives. For applications using the COBOL/2 interface to the OS/2 EE Database Manager, using the /NOTRUNC option is suggested (see *Note 2*). This option takes advantage of the full range of COMP-5 integers. This step creates the object file. The compiler produces a new file with a .OBJ extension.

Example:

```
PCOBOL INCREASE.CBL /NOTRUNC
```

This command compiles the program INCREASE and creates a .OBJ file with the name INCREASE.

The next step is to link the object file with the proper libraries. To link a program to run under OS/2, the PCOBOL and DOSCALLS libraries must be used. Three libraries provided by Database Manager may also need to be included: SQL_DYN.LIB (dynamic), SQL_STAT.LIB (static), and/or DSQCI.LIB (Query Manager Callable Interface). /NOP is a required option when linking a program to run under OS/2 (see *Note 2*). Using this option ensures that the link command will not try to pack neighboring logical code segments into one physical segment.

If an application consists of several modules, there are two ways to link it: as a single .EXE file, or as a main .EXE that calls one or more .DLL (Dynamic Link Library) files at runtime. Linking produces the executable program module; an example is shown in Figure 6.

Binding The Program

If binding was deferred, the program .BND file needs to be bound to a database before it will execute (see *Note 1* on the next page).

Example:

```
SQLBIND INCREASE.BND SAMPLE
```

This binds the INCREASE.BND file to the SAMPLE database.

Example:

```
    LINK INCREASE.OBJ
```

```
,,,\PCOBOL\PCOBOL.LIB+
   \OS2\DOSCALLS.LIB+
   \SQLLIB\SQL_STAT.LIB+
   \SQLLIB\SQL_DYN.LIB+
   \NOP;
```

This command links the program INCREASE with all the libraries listed. The three commas mean that it will produce a .EXE file (the executable module) and a .MAP file (a list of the segments of the program and groups) with the same name as the .OBJ file (INCREASE). The "response" information (surrounded by the box) can be stored in a file of any name. To use it with the link command, it must be preceded by an "at" sign (@). If this information was saved in a file called DBM.LNK, the link command would look like this:

```
LINK INCREASE.OBJ @DBM.LNK
```

Figure 6. Linking a Program

If there is more than one .BND file to be bound to the database, a file needs to be created that lists the bind file names. To use this file with the SQLBIND command, it must be preceded by an "at" sign (@).

Sample Program

Figures 7a, 7b and 7c show a program that uses the COBOL/2 interface to access information in the OS/2 EE Database Manager database "SAMPLE" to increase by five percent the salary of employees who have been with the company more than 10 years. This program is meant for illustrative purposes only. Comments are indicated with asterisks.

Note 1: Several other options can be used with the SQLPREP and the SQLBIND commands. These are documented in the *IBM OS/2 1.2 EE V1.2 Commands Reference* manual, S01F-0282.

Note 2: Several other options can be used with the PCOBOL and LINK commands. These are documented in the *IBM COBOL/2 Compile, Link and Run* manual, 84X1673.

ABOUT THE AUTHOR

Christine F. Jagger is an advisory market support representative in the Desktop Systems Support Center for IBM's Marketing and Services Group in Dallas. She currently supports OS.2 EE 1.2 Database Manager as leader of the COBOL/2 team. Chris has been in a technical support role since joining IBM in 1977. Before coming the the OS/2 EE Database team, she was a member of the RT/PC AIX technical support team.

```
*********************************
*********************************
IDENTIFICATION DIVISION.
PROGRAM-ID. INCREASE.

*********************************
* DATABASE              :OS/2 EE SAMPLE
* ENVIRONMENT           :STANDALONE
* PURPOSE               :UPDATE STAFF  TABLE:
*                       :INCREASE SALARY BY 5% THOSE EMPLOYEES WHO
*                       :WHO HAVE BEEN HERE MORE THAN 10 YEARS.
* DEPENDENCIES          :THE SAMPLE DATABASE PROVIDED WITH OS/2 EE
*                       :MUST BE INSTALLED. TO INSTALL TYPE: SQLSAMPL
*  COMPLIER OPTIONS     :/NOTRUNC /ANS85

*********************************
*********************************
DATA DIVISION.

*********************************
WORKING-STORAGE SECTION.

*********************************
COPY SQLCA.
COPY SQLCODES.

*********************************
* Fields used for Start Using Database
*********************************
  77 D-USE              PIC 9(4) COMP-5
  77 U REDEFINES D-USE  PIC X.
  77 SPARE2             POINTER.
  77 DATABASE-NAME      PIC X(8) VALUE "SAMPLE".
  77 SPARE1             PIC 9(4) COMP-5 VALUE 0.
  77 DB-NAME-LENGTH     PIC 9(4) COMP-5 VALUE 6.
```

Figure 7a. Sample DBM COBOL/2 Program

```
**********************************
* Fields used to Retrieve Error Message
**********************************
  77 BUFFER-SIZE        PIC 9(4) COMP-5 VALUE 512.
  77 LINE-WIDTH         PIC 9(4) COMP-5 VALUE 0.
  77 BUFFER             PIC X(512).
**********************************
**********************************
PROCEDURE DIVISION.
**********************************
* Install signal handler to handle CRTL-BREAK
**********************************
    CALL "_SQLGISIG" USING SQLCA.
**********************************
* See START-DB for start using database
**********************************
    PERFORM START-DB.
**********************************
* Update the appropriate rows
**********************************
    EXEC SQL
      UPDATE STAFF
           SET SALARY = SALARY + (SALARY * .05)
           WHERE YEARS > 10
    END-EXEC.
**********************************
* Check the SQLCODE for errors
**********************************
    EVALUATE SQLCODE   WHEN 0 CONTINUE
                       WHEN OTHER PERFORM DBERROR
    END-EVALUATE.
**********************************
* Stop using the database
**********************************
    CALL "_SQLGSTPD" USING SQLCA.
    GOBACK.
**********************************
* Start Using Database
**********************************
    START-DB.
**********************************
* Use Database in Shared Mode
**********************************
    MOVE "S" TO U.
    CALL "_SQLGSTRD" USING DATABASE-NAME
                           SPARE2
                           SQLCA
                           BY VALUE D-USE
                           BY VALUE DB-NAME-LENGTH
                           BY VALUE SPARE1.
```

Figure 7b. Sample DBM COBOL/2 Program

```
**********************************
* Look to see if database needs to be Restarted
**********************************
     EVALUATE SQLCODE   WHEN 0 CONTINUE
                        WHEN -1015 PERFORM RESTART
                        WHEN OTHER PERFORM DBERROR
     END-EVALUATE.
**********************************
* Restart database if necessary
**********************************
   RESTART.
     CALL "_SQLGREST" USING   DATABASE-NAME
                              SPARE2
                              SQLCA
                              BY VALUE DB-NAME-LENGTH
                              BY VALUE SPARE1
     EVALUATE SQLCODE   WHEN 0 PERFORM START2
                        WHEN OTHER PERFORM DBERROR
     END-EVALUATE.
**********************************
   START2.
     CALL "_SQLGSTRD"   USING  DATABASE-NAME
                               SPARE2
                               SQLCA
                               BY VALUE D-USE
                               BY VALUE DB-NAME-LENGTH
                               BY VALUE SPARE1
     EVALUATE SQLCODE   WHEN 0 CONTINUE
                        WHEN OTHER PERFORM DBERROR
     END-EVALUATE.
**********************************
* Display error #
**********************************
   DBERROR.
     DISPLAY "SQL ERROR, SQLCODE = "SQLCODE.
**********************************
* Retrieve Error Message and Display it
**********************************
     CALL "_SQLGINTP" USING   BUFFER
                              SQLCA
                              BY VALUE LINE-WIDTH
                              BY VALUE BUFFER-SIZE.
DISPLAY "SQL ERROR MESSAGE = "BUFFER.
GOBACK.
```

Figure 7c. Sample DBM COBOL/2 Program

Database Manager Programming with Procedures Language 2/REXX

Jeffrey W. Fisher
IBM Corporation
Dallas, Texas

Procedures Language 2/REXX is a powerful OS/2 Extended Edition Version 1.2 tool for creating programs utilizing the Database Manager. The Database Manager and Query Manager interfaces for Procedures Language covered in this article provide the reader with basic knowledge and examples of writing programs to create business applications and database management tools. Several sample programs illustrate each interface.

Procedures Language Overview

Procedures Language 2/REXX is a powerful structured programming language that is included with OS/2 Extended Edition Version 1.2. It is designed for ease of use with very English-like syntax. Examples of instructions are SAY, PULL, and IF-THEN-ELSE. SAY and PULL are used respectively to display characters to the screen and to read characters from the screen. Several commands are included for structured programming.

Procedures Language has a free-format syntax. One or more instructions can span multiple lines, start in any column, and use mixed (upper and lower) case. A good programming style uses these techniques to enhance readability.

Procedures Language includes many built-in functions to save programmer's time. Several have very powerful string-parsing capabilities. Others deal with data formatting.

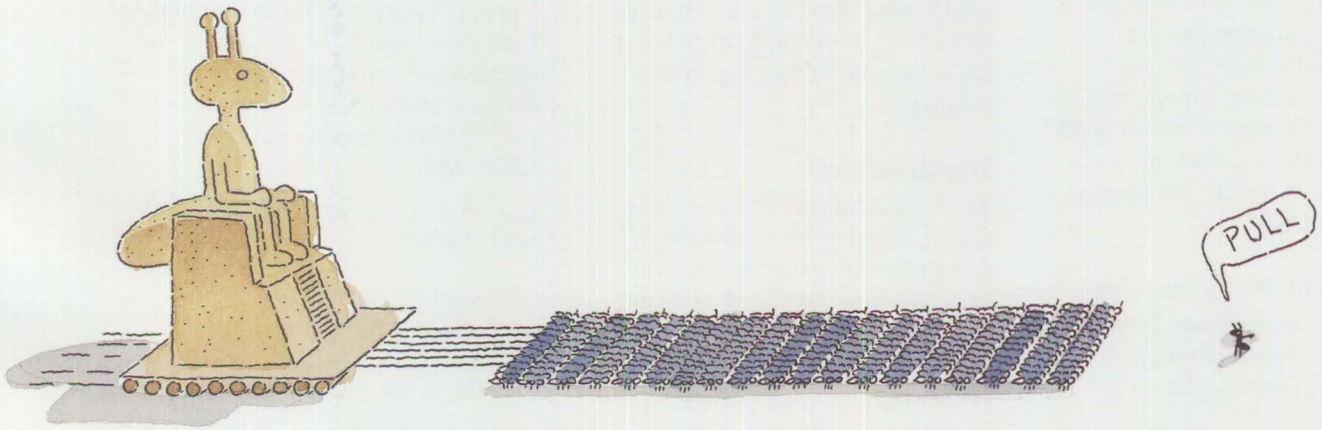
One important feature of Procedures Language is that all variables are considered typeless. All data is treated as character strings, although math can be performed on any variables that contain valid numerics. Designers of highly numeric-intensive applications should consider the specific capabilities of Procedures Language when choosing the most appropriate language for each part of an application.

In addition, there is no need to predeclare variables (as would be done, for example, in a COBOL working-storage section); as soon as these variables are referenced in the program, they are added to the Procedures Language's variable pool. A variable pool is a table maintained internally by Procedures Language. It identifies all known variables and their values. This ease of definition makes it particularly easy to use the powerful database and dialog interfaces available to Procedures Language users.

Procedures Language contains a variety of tracing options for run-time debugging. The TRACE facility can be used to display each line of the program as it executes.

Procedures Language is particularly suited to:

- Command Procedures
- Application Driver
- Application Programming
- Structured Query Language (SQL) Statement Prototyping



- Application Prototyping
- Database Administration Tools

Command Procedures

Command procedures (.CMD files), as used in previous versions of OS/2, are still available. However, in any situation where a command file would be used in OS/2 Extended Edition, a Procedures Language program can be substituted. This gives users significant new capabilities and flexibility.

Both Procedures Language programs and command procedures use a ".CMD" extension. Procedures Language programs are differentiated from OS/2 command procedures because its a Procedures Language requirement that the first line begins with a comment. Procedures Language is not supported for DOS, DOS Compatibility Box, or OS/2 SE users.

OS/2 optionally runs a command procedure at boot time named STARTUP.CMD. Users gain additional flexibility by using Procedures Language to write this procedure.

Application Driver

Procedures Language can be used as a front end for an application and as a streaming tool for individual application programs. For example, a Procedures Language program could be used to automatically download host files, import them to the Database Manager, run an application program, and then check its return codes or handle any exceptional conditions.

Application Programming

Procedures Language can be used for business application programming. It can format output reports, and can use both Query Manager and Dialog Manager as presentation interfaces. Because Procedures Language programs contain typeless variables, it may not always be suitable for applications requiring complex numerical calculations. A higher-level language should be considered depending on application requirements.

Application Prototyping

If the programmer has decided to use a higher-level language, the Procedures Language can still be used during development for prototyping the application. Simple screen flows and representative data could be created easily and demonstrated to end users. End users could then approve the application flow earlier in the development cycle. Most or all of the SQL, Query Manager commands, and any Dialog Tag Language statements written for the prototype could then be reused in the actual application code.

Database Administration Tools

Procedures Language supports nearly all of the Database Manager API calls. A complete list of supported API calls is available in the *OS/2 Extended Edition Version 1.2 Database Manager Programming Guide and Reference* (S01F-0269). Procedures Language offers a particularly convenient means of housing these calls, with much simpler syntax requirements than higher-level languages.

Implications

The implications of some of these features need to be considered by the application designer. In terms of application programming to the Database Manager, Procedures Language differs from higher-level languages in the ways that code is developed and prepared for execution. Because Procedures Language programs run interpretively, compiles, links and binding to the Database Manager are not supported. During development, the programmer can edit a program in one window, save it, and immediately run it in another window.

Procedures Language programs support only dynamic SQL. Therefore, a Database Manager Access Plan is not used. This must be taken into consideration if the application has performance requirements that an access plan would be required to support.

Procedures Language Variables

To understand how the various interfaces communicate with Procedures Language, it is necessary to review basic rules of Procedures Language variables.

Each Procedures Language program has its own unique pool of variables. These variables do not need to be declared unless it is necessary to assign a value to them before they are used. The following statement both declares variable STMT and assigns a value to that variable for later use:

```
STMT = "Select * from STAFF"
```

Variable values can also be entered from the screen. The following statement asks the user for a database name to be used and assigns it to the variable named DBNAME:

```
say "Please enter the DB name"
pull DBNAME
```

With compound variables, multidimensional arrays may be created. For example, the variable ANSWERS is one-dimensional. How-

`/* Name : COMPILE.cmd */`
`/* Platform : OS/2 Extended Edition 1.2 */`
`/* Purpose : Performs a PCOBOL compile */`	

`say 'Enter the name of your program'` `pull progname`	Ask the user for the program name to compile. PULL the name of the program. The variable PROGNAME now contains the program name.
`say 'Beginning compile of' progname`	Display a message stating that the compile is beginning.
`address cmd 'PCOBOL' progname`	Pass the PCOBOL command to the OS/2 command processor, with the program name.
`say 'Compile complete'`	When the compile is complete, notify the user.
`exit`	The Procedures Language EXIT command ends the program and returns control to OS/2.

Figure 1. A COBOL Compile Utility.

ever, ANSWERS.1 is the first entry of the array ANSWERS, ANSWERS.2 is the second entry, etc. Some of the following sample programs use compound variables to receive data from the Database Manager utilities.

External Command Passing

Commands can be passed to environments external to the Procedures Language with the ADDRESS command. The following command passes a "DIR" command to the OS/2 command processor:

```
ADDRESS CMD 'DIR'
```

Some commands, such as the Procedures Language EXIT, are common to both Procedures Language and external environments. EXIT is used to end a Procedures Language program. Passing this command to OS/2 also causes the program to end and causes the window to close if running in a window.

Figure 1 shows an example of using Procedures Language to create a simple utility to assist in a repetitive task. The sample program asks the user for a program name and then passes it to the COBOL/2 compiler. The code is in the left column: the explanation of the code is in the right column. In some later examples where it is not necessary to show all of the code in a sample program, single periods represent lines of code not shown.

All Procedures Language programs are required to begin with a comment. Comments begin with /* and end with */. This signifies to OS/2 that it will process a Procedures Language program, rather than a Command procedure.

Complex commands with multiple parameters can be simplified to the end user by writing a Procedures Language program to collect, edit, and pass the commands in proper sequence. Figure 1 could be easily modified to perform a precompile, compile, link, and bind for a COBOL/2 program. Procedures Language can even help the user with the relative complexities of high-level language programming.

Interface Registration

Interfaces external to Procedures Language, including Database Manager and Query Manager, must be identified to the Procedures Language command processor before they can be used. This in known as registering an interface. Each interface can be registered and dropped in each Procedures Language program or can be accomplished via the STARTUP.CMD at boot time. DLL modules associated with these interfaces require system memory even when not in use. Under some circumstances, it may be desirable to drop the interfaces when they are not needed.

The systems designer must decide to register functions consistently either at boot time with STARTUP.CMD or in each program requiring them. If performed in each individual program, then the designer must ensure that these necessary functions are not dropped when they may be in use by another user. Therefore, it may be advisable to register all functions at boot time only.

The examples presented in this article all register their interfaces in STARTUP.CMD. Note that the STARTUP.CMD also executes a STARTDBM (start Database Man-

ager). Starting and stopping Database Manager is similar to registration because another application may be concurrently using Database Manager.

Figure 2 illustrates an example of a STARTUP.CMD showing the Procedures Language commands used to register the interfaces required by the sample programs.

Database Manager Interface

The Database Manager interface uses a communication area known as the SQLCA to pass information contained within variables to and from the Database Manager. Several SQLCA and other variables are provided for the user by the interface (Figure 3).

Figure 4 shows use of some of the variables provided in the communications area to handle exceptions.

API Calls Interface

The SQLDBS interface processes Database environment and utility routines. It operates on a call/return basis, whereby control is passed to Database Manager to perform a function. When Database Manager finishes, control is passed back to the program with return-code information.

API call syntax is:

CALL SQLDBS 'character-string'

'Character-string' can be any one of a number of supported API calls. The character string is delineated by quotes and can span multiple lines. A multiple line command string will be discussed in a later example.

```
/* Name     : STARTUP.cmd                    */
/* Purpose  : boot-time startup functions    */
/* Platform : OS/2 Extended Edition 1.2      */
/*                                           */

say   'STARTUP Processing'

say   'Register Procedures Language SQL functions'




rcy = RxFuncadd('SQLEXEC', 'SQLAR', 'SQLEXEC')


rcy = RxFuncadd('SQLDBS' , 'SQLAR' , 'SQLDBS')

say   'Register QMCI function'
DSQRGSTR

say   'Start Database Manager'

STARTDBM


say   ' '
say   '>>> STARTUP command complete <<<'
say   ' '
ADDRESS CMD 'EXIT'
```

The SAY command displays a message to the screen. Because the operating system starts a window for STARTUP.CMD to run in, messages are displayed in that window as the Procedures Language program executes. This keeps the user informed of a program's progress.

The syntax for registering the Procedures Language SQL interface.

The syntax for registering the Procedures Language Database Manager API interface.

The command that registers the Query Manager Callable Interface. A command is provided for this, although the other method is also available.

This step starts the Database Manager. This is handled as in OS/2 Extended Edition Version 1.1.

This step displays a message that the program has completed, again for the convenience of the user.

This step closes the window. Because EXIT is both a Procedures Language and an OS/2 command, ADDRESS CMD is used to pass an EXIT command specifically to OS/2 to end the program and close the window. Otherwise, the window would be left open and would need to be closed manually.

Figure 2. STARTUP.CMD

Figure 5 contains an example of the SQLDBS interface. As in OS/2 Extended Edition Version 1.1, certain situations require a START USING DATABASE to be performed. STOP USING also needs to be present in this program, but is not shown in this partial example.

SQL Calls Interface

The SQLEXEC interface is used to process SQL requests. It operates on a call/return basis also.

The syntax of the API call is as follows:

CALL SQLEXEC 'character-string'

'Character-string' can be any one of a number of supported SQL calls. The character string is delineated by quotes and can span multiple lines.

Figure 6 shows an example of the SQLEXEC interface.

SQLCA.SQLCODE	SQL return code
SQLCA.SQLERRML	Length of data in the following variable
SQLCA.SQLERRMC	Error message tokens, if an error occured
SQLCA.SQLERRP	Eight-byte diagnostics
SQLCA.SQLERRD.1 thru 6	Six-element array of diagnostics
SQLCA.WARN.0 thru 7	Eight-element array of warning flags
SQLISL	Isolation level in use
SQLMSG	If the SQLCA.SQLCODE variable is set to a value other than zero, this contains text describing the error

Figure 3. Partial List of SQLCA Variables

The command string passed to the interface is composed of any of the following types of elements:

- SQL keywords
- Host variables
- Pre-defined identifiers

There is support for all SQL keywords. SELECT, UPDATE, and DELETE must be PREPAREd prior to execution, because all SQL is executed dynamically with the Procedures Language interface.

Each host variable is preceded in the command string by a colon.

```
.
if SQLCA.SQLCODE \= 0 then signal SQLERR
```

The syntax of the call to the Database Manager interface will be shown in a following example. The return code should be checked immediately following any call to the Database Manager. After the call is made, check the return code stored in SQLCA.SQLCODE. If it is not equal to zero, pass control to another part of the program to handle the error situation. The SIGNAL command is used to pass control to a Procedures Language label called SQLERR.

```
.
.
.
```

If the SQLCODE was not set, continue processing in the program.

```
SQLERR:
```

A Procedures Language label is signified by the name and the colon character. Control resumes here. This program assumes all non-zero SQL codes indicate an error.

```
   say   'SQL Fatal error:'
   say   '   SQLCODE = ' SQLCA.SQLCODE
   say   '   SQLMSG  = ' SQLMSG
```

Display the error code and corresponding message. Note that unlike other languages, the Procedures Language SQL interface automatically provides an interpreted error message for your use.

```
   exit
```

End of program.

Figure 4. SQLCODE Checking

Three types of host variables are used in the SQLEXEC string:

- Statement
- Data
- SQLDA

Statement host variables are a string that contain an SQL statement. As an example, Figure 7 illustrates defining a Procedures Language variable to use as a host variable.

Data host variables are used for transferring data to the Database Manager or for receiving data from an SQL SELECT statement for use by the Procedures Language program. Figure 8 shows the SQL types supported. As stated before, Procedures Language regards them as typeless.

SQLDA host variables are used in PREPARE, DESCRIBE, OPEN, EXECUTE, and FETCH SQL statements.

A complex (array) variable is used to represent the SQLDA. The structure is shown in Figure 9.

Further information on the uses of the SQLDA can be found in *OS/2 Extended Edition Version 1.2 Database Manager Programming Guide and Reference*.

Predefined Identifiers

Cursor and statement names are predefined by the interface. Cursor names, used in DECLARE, OPEN, FETCH, and CLOSE statements, range from C1 to C100. Statement names, used in DECLARE, DESCRIBE, PREPARE, and EXECUTE statements, range from S1 to S100. The interface requires the use of these names only. Other languages allows the program to declare their name.

Sample Programs

Now that the Procedures Language basics pertaining to the Database Manager interface have been discussed, samples of these basics are presented. All the following examples run in OS/2 windows without a presentation interface.

Procedures Language, like other supported languages, requires a START and STOP USING DATABASE.

```
call SQLDBS    'START USING DATABASE DEMO IN',
               'SHARED MODE'

if SQLCA.SQLCODE \= 0 then signal SQLERR
.
.
.
SQLERR:
.
.
.
```

Routine calls are very simple to code in Procedures Language. Simply enclose the command strings in single quotes. Variables outside the quotes can be resolved as shown in a later example.

Check the SQLCODE after each call.

Code an appropriate error-handling routine as shown in the previous example.

Figure 5. Use of SQLDBS

```
call SQLEXEC 'FETCH C1 INTO :acctno :acctname'

if SQLCA.SQLCODE \= 0 then signal SQLERR
.
.
.
SQLERR:
.
.
.
```

A precompiler is not used by the Procedures Language interface. SQL statements are simply enclosed in parentheses. The use of host variables is discussed later.

Check the SQLCODE after each call.

Code an appropriate error-handling routine.

Figure 6. Use of SQLEXEC

```
creattab = 'create table neworg (',
```

The variable CREATTAB will contain the CREATE statement. A quote and comma is used to continue definition of the Procedures Language variable to the next line. The variable will be seen as a single string to the interface.

```
        'deptnum      smallint not null,',
        'deptname     varchar(14),',
```

The next line starts with a quote and continues the same way as the previous line.

```
        'location     varchar(13) )'
```

Following proper SQL syntax, the SQL statement is delineated by a closing parenthesis.

```
call SQLEXEC 'EXECUTE IMMEDIATE :creattab'
.
.
.
```

The statement is executed. SQLCODE should be checked for successful completion.

Figure 7. Statement Host Variable

Figure 10 shows an example of a program that performs this. Because the code required to perform the function is lengthy, the following program can be called from any Procedures Language program.

Figure 11 shows an application program that displays a table's contents.

Figure 12 shows how Procedures Language is used to create a Database Manager utility. A Database Administrator can use Procedures Language to create many of the tools necessary to administer Database Manager. This utility opens a scan on the volume database directory and reports valuable information. The program runs in an OS/2 window. The Dialog Manager could be used by this program to create a better presentation interface for this utility.

The preceding examples illustrate the simplicity of housing application or utility code in Procedures Language programs. The next section examines how to take advantage of Query Manager functionality in Procedures Language code.

Column type	SQL data type	
	not nullable	nullable
SMALLINT	500	501
INTEGER	496	497
FLOAT	480	481
DECIMAL	484	485
CHAR	452	453
VARCHAR	448	449
LONG VARCHAR	456	457
GRAPHIC (DBCS)	468	469
VARGRAPHIC (DBCS)	464	465
LONG VARGRAPHIC (DBCS)	472	473
DATE	384	385
TIME	388	389
TIMESTAMP	392	393

Figure 8. SQL Types Supported

XXX = Any valid variable name n = The number of the SQLVAR element	
XXX.SQLN	Maximum number of variables allowed
XXX.SQLD	Current number of valid variables
XXX. - XXX.n	SQLVARS, an array occurring 1 to n times, which contains the following for each variable
XXX.n.SQLTYPE	Data type
XXX.n.SQLLEN	Maximum length
XXX.n.SQLDATA	Data
XXX.n.SQLIND	Name of indicator variable
XXX.n.SQLNAME	Name of column

Figure 9. List of SQLDA Variables

```
/*    Name     : PL001.cmd                        */
/*    Purpose  : START or STOP USING database     */
/*    Platform : OS/2 Extended Edition 1.2        */
/*                                                */

arg dbcmd, dbname
```

Two arguments are passed from the calling program – a START or STOP instruction and the database name (used on the START only).

```
if dbcmd = 'START' then signal START
if dbcmd = 'STOP' then signal STOP
```

Branch to the appropriate section depending on the command received.

```
say   'Error: PL001 invalid command:' dbcmd
return
```

Handle an invalid command.

```
START:
```

The START USING section begins at this label.

```
   call SQLDBS dbcmd,
        'USING DATABASE' dbname,
        'IN SHARED MODE'
```

The Procedures Language variables for command and database name are embedded in the command. These are resolved by Procedures Language before the call is made.

```
   if SQLCA.SQLCODE = 0 then signal ENDIT
```

If the start is successful, go to this label to end the program.

```
   if SQLCA.SQLCODE = -1015 then
      do
         call SQLDBS 'RESTART DATABASE' dbname
         if SQLCA.SQLCODE \= 0 then signal SQLERR
         call SQLDBS dbcmd,
            'USING DATABASE' dbname,
            'IN SHARED MODE'
         if SQLCA.SQLCODE = 0 then signal ENDIT
         else signal SQLERR
      end
   else
      signal SQLERR
```

If a restart is required, then RESTART the database. If successful, then perform the START USING again. If successful, go to this label to end the program.

If the database still cannot be started, a serious error has occurred. Branch to the error-handling session.

```
STOP:
```

The STOP section.

```
   call SQLDBS dbcmd 'USING DATABASE'
   if SQLCA.SQLCODE \= 0 then signal SQLERR
```

Substitute variables as before.

```
ENDIT:
   return
```

When complete, return control to the calling program.

```
SQLERR:
   say  'SQL Fatal error:'
   say '   SQLCODE = ' SQLCA.SQLCODE
   say '   SQLMSG  = ' SQLMSG
   say 'PL001:' dbcmd dbname   'unsuccessful'
   signal ENDIT
```

The SQL error-handling code. Display descriptive information so the error can be determined.

Figure 10. START and STOP USING DATABASE

```
/*Name         : PL100.cmd                          */
/*Platform     : OS/2 Extended Edition 1.2          */
/*Purpose      : display the SAMPLE ORG table       */
/*Dependency   : SAMPLE database installed          */

call PL001 START, SAMPLE


stmtbuf = 'select * from org order by deptnumb'


call SQLEXEC 'PREPARE s1 FROM :stmtbuf'
if SQLCA.SQLCODE \= 0 then signal SQLERR

call SQLEXEC 'DECLARE c1 CURSOR FOR s1'
if SQLCA.SQLCODE \= 0 then signal SQLERR
call SQLEXEC 'OPEN c1'
if SQLCA.SQLCODE \= 0 then signal SQLERR
do while SQLCA.SQLCODE = 0
   call SQLEXEC 'FETCH c1 INTO  :deptnumb,',
                               ' :deptname,',
                               ' :manager,',
                               ' :division,',
                               ' :location'

   if SQLCA.SQLCODE = 0 then
   do




      say   '*****************************'
      say   'Department Number   =' deptnumb
      say   'Department Name     =' deptname
      say   'Manager             =' manager
      say   'Division            =' division
      say   'Location            =' location
   end

end

if SQLCA.SQLCODE \= 100 then signal SQLERR



signal ENDIT


SQLERR:
   say 'SQL Error:'
   say '   SQLCODE = ' SQLCA.SQLCODE
   say '   SQLMSG  = ' SQLMSG
ENDIT:
   call SQLEXEC 'CLOSE c1'
   call PL001 STOP
   exit
```

The program must first execute a START USING DATABASE. The program shown in the previous example will be used to START USING DATABASE SAMPLE. This program uses the optional sample database delivered with OS/2 Extended Edition 1.2, which was installed using the SQLSAMPL command.

A statement host variable is used to house the SQL SELECT statement.

The statement is prepared for execution. Error checking is performed.

A cursor is declared for the SELECT statement.

The cursor is opened.

A DO loop will be performed repetitively while the SQLCODE is equal to zero. Any non-zero SQLCODE will cause control to drop out of the loop. The DO loop is delineated by an END statement. A FETCH statement selects the first or next row.

If the SQLCODE from the fetch is zero, then the row can be displayed. An IF-THEN statement conditionally executes statements 8, 9 and 10. Similar structured programming statements in other languages are known as Case Statements.

Each column of the row is displayed.

The delineator of the conditional IF-THEN statement.

The delineator of the DO loop.

A check to see if the DO loop ended from an abnormal SQL condition code. If so, pass control to the SIGNAL label to handle the error.

If an error did not occur, pass control to the ENDIT label to terminate the program.

The SQL error-handling code. Pertinent variables will be displayed to assist in problem resolution.

The ENDIT label. The open cursor is closed, and a call is made to the subroutine to STOP USING the database. An exit statement ends the program and returns control to OS/2.

Figure 11. An Application Program

```
/* Name        : PL008.cmd                    */
/* Purpose     : Database Directory Scan      */
/* Platform    : OS/2 Extended Edition 1.2    */
/*                                            */

call SQLDBS 'OPEN DATABASE DIRECTORY ON C',
            'USING :scanvar'
if SQLCA.SQLCODE \= 0 then signal SQLERR

say
say '-----------'
say '  Database Directory Scan for C Drive'
do i=1 to scanvar.2

   call SQLDBS 'GET DATABASE DIRECTORY ENTRY',
               ':scanvar.1 USING :entry'
   if SQLCA.SQLCODE \= 0 then signal SQLERR

   say   say 'Database:' entry.2 'Alias:' entry.1,
         'Drive:' entry.3
   say ' Internal name :' entry.4
   say ' Node name     :' entry.5
   say ' Dbtype        :' entry.6
   say ' Comment       :' entry.7
   say ' Codepage      :' entry.8
   say ' Enttype       :' entry.9
   say
   say
   pause
end
say '-----------'
say
call SQLDBS 'CLOSE DATABASE DIRECTORY',
            ':scanvar.1'
if SQLCA.SQLCODE \= 0 then signal SQLERR

ENDIT:
   ADDRESS CMD 'EXIT'

SQLERR:
   say   'SQL Fatal error:'
   say   '  SQLCODE  =  ' SQLCA.SQLCODE
   say   '  SQLMSG   =  ' SQLMSG
   call SQLDBS 'CLOSE DATABASE DIRECTORY :scanvar'
   pause
   signal ENDIT
```

This statement opens a directory scan for drive C. All databases located on drive C will be reported. The first element in the compound variable "scanvar" will contain a value for the scanid.

The second element of the compound variable scanvar contains the number of databases on the drive. The loop will be performed this many times. The variable "i" will be used as a counter for a loop.

The first element of the compound variable scanvar contains a unique identifier for the scan. This variable identifies to Database Manager the resources allocated to the scan.

A compound variable was returned containing the information about the database. Each element of this variable is displayed.

A pause command is used to give users time to read the information. Dialog Manager could be used to handle user interaction more eloquently.

The scan is closed using the unique identifier. Appropriate error-code checking is performed.

Figure 12. Database Directory Scan

Query Manager Callable Interface

The Query Manager Callable Interface (QMCI) contains an interface for: (1) starting the Query Manager interface, (2) passing commands to the Query Manager and acting upon the results of those commands, and (3) stopping the Query Manager interface.

Like the Database Manager interface, the QMCI works on a call/return basis and uses a communications area to pass information. The QMCI handles the interface to the Database Manager beneath the surface. The QMCI handles the interface to the Database Manager. Therefore, START and START USING statements should not be coded unless the program is also performing SQL calls in addition to the QMCI calls.

The QMCI follows the Systems Application Architecture™ (SAA™) Query CPI format. This format is consistent with the functionality of Query Management Facility™ Version 2.4 on MVS and VM systems.

The QMCI allows a program to take advantage of all Query Manager facilities and resources. Using the QMCI can save having to code equivalent functions separately. Significant development effort can be saved. Furthermore, if end users are already familiar with the Query Manager, using a familiar interface will save documentation and training time.

Four types of statements are necessary for a program to use the Query Manager Callable Interface:

- A communications area (DSQCOMM)
- START the interface
- QM session commands
- EXIT the interface

The variables used in the communications area are all predefined by the QMCI interface. These variables will be used both as return codes and as working storage areas.

Four DSQCOMM variables contain completion information:

DSQ_RETURN_CODE indicates the call's outcome. Values returned are:

- 0 : successful execution
- 4 : successful execution with a warning
- 8 : command did not execute correctly
- 16: severe error, the QMCI session is terminated.

DSQ_REASON_CODE contains a further explanation of the call's outcome.

DSQ_ERROR_ID and DSQ_MESSAGE_ID contains error information.

The interface is initiated with the START command. A partial list of the optional parameters are:

1. DSQSMODE defines how the interface will be used. The QMCI can be run in BATCH or INTERACTIVE mode. Interactive means that Query Manager screens will be displayed for the user. Batch mode indicates that screens will not and cannot be displayed.

As an example of BATCH versus INTERACTIVE, consider an application requirement that produces a customer report. If input from the user is not necessary to run the report, then BATCH would be used. The application does not display any Query Manager panels as it runs.

If input is required (perhaps the report is run for a certain customer), then INTERACTIVE would be used. When run, a Query Manager panel would be displayed asking for a customer number that would set a Query Manager variable. That variable would then be used exactly as in a stand-alone Query Manager procedure.

2. With DSQSPROF, a Query Manager profile may be specified. A profile can indicate which database to use for subsequent queries, or can indicate a Query Manager procedure to be run at initiation of the session.

3. DSQSDBNM can be used to specify which database to use.

4. DSQSRUN specifies a Query Manager procedure to run. Note that the procedural language used by Query Manager is a limited subset of Procedures Language. See the *OS/2 Extended Edition Version 1.2 User's Guide* for a complete description of the Query Manager procedural command syntax.

5. DSQSCMD is the name of an optional .CMD file to execute. This command file will be used to start Query Manager and optionally pass parms to it. If DSQSCMD is not specified, \SQLLIB\QRWEXECZ.CMD is used. This batch command, which is delivered with OS/2, is used to start Query Manager in a stand-alone mode. If your application pro-

```
/* Name        : QM001.cmd                    *
/* Purpose     : Sample QM CI - Interactive   */
/* Platform    : OS/2 Extended Edition 1.2    */
/* Dependency  : QM query 'select * from      */
/*                staff'                      */
/*                                            */
say 'QM001 - QM CI Interactive'
say

say 'Start database SAMPLE'
CALL DSQCIX ,
   'START',' DSQSMODE=INTERACTIVE',
   ' DSQSDBNM="SAMPLE"'
if DSQ_RETURN_CODE \= 0 then signal QMERR

say 'Edit table'

CALL DSQCIX 'EDIT TABLE STAFF'
if DSQ_RETURN_CODE \= 0 then signal QMERR

say 'Run query'


CALL DSQCIX 'RUN QUERY ALL_OF_STAFF'
if DSQ_RETURN_CODE \= 0 then signal QMERR

say 'Print report'
CALL DSQCIX 'PRINT REPORT'
if DSQ_RETURN_CODE \= 0 then signal QMERR

say 'Exit'
CALL DSQCIX 'EXIT'
if DSQ_RETURN_CODE \= 0 then signal QMERR

ENDIT:
   ADDRESS CMD 'EXIT'

QMERR:
   say   'The QM CI has ended with a non-zero',
         'Return Code:'
   say
   say   '  DSQ_RETURN_CODE   ' DSQ_RETURN_CODE
   say   '  DSQ_REASON_CODE   ' DSQ_REASON_CODE
   say   '  DSQ_ERROR_ID      ' DSQ_ERROR_ID
   say   '  DSQ_MESSAGE_ID    ' DSQ_MESSAGE_ID
   say
   pause
   signal ENDIT
```

Because this program is run in an OS/2 window, a message is displayed to the user.

The QMCI is started in interactive mode for the database SAMPLE. The user has not yet seen a Query Manager screen. Because the database to be used is specified, the usual first Query Manager screen that shows the list of databases will not be shown.

The user is now in edit mode for the specified table. At this point, the first Query Manager screen is displayed. This is the familiar edit screen to choose Add or Change. After the choice is made, the user is in Edit mode.

After the edit is complete, this query is run, and the Query Manager report screen is displayed. This is the last Query Manager screen the user will see.

After the report is viewed, it is routed directly to the print queue. Print options can also be specified if desired.

The interface is stopped.

The program is ended, and the window is closed.

In case of error, pertinent variables are displayed.

Figure 13. Interactive QMCI Application

gram needs to change this procedure, make a copy of it using a new name.

Figure 13 is an example of an interactive execution using QMCI. In this example, the user will edit the table STAFF. When complete, a query is run showing the entire STAFF table. After the user has reviewed the query results, the query is printed directly to the print queue and the interface is ended.

Figure 14 shows a batch QMCI program. This program displays messages in the OS/2 window as it runs, but does not display any Query Manager screens.

```
/* Name        : QM002.cmd                        */
/* Purpose     : Sample QM CI - Batch             */
/* Platform    : OS/2 Extended Edition 1.2        */
/* Dependency  : QM query 'Select * from          */
/*             staff'                             */
/*                                                */

say
say 'QM002 - QM CI Batch'
say

say 'Start in batch mode with database SAMPLE'
CALL DSQCIX,
'START',' DSQSMODE=BATCH',' DSQSDBNM="SAMPLE" '
if DSQ_RETURN_CODE \= 0 then signal QMERR

say 'Run query'
CALL DSQCIX 'RUN QUERY ALL_OF_STAFF',
     '(INTERACT=NO)'
if DSQ_RETURN_CODE \= 0 then signal QMERR

say 'Print report'
CALL DSQCIX 'PRINT REPORT'
if DSQ_RETURN_CODE \= 0 then signal QMERR


say 'Exit'
CALL DSQCIX 'EXIT'
if DSQ_RETURN_CODE \= 0 then signal QMERR

ENDIT:
   ADDRESS CMD 'EXIT'

QMERR:
   say 'The QM CI has ended with a non-zero',
       'Return Code:'
   say
   say ' DSQ_RETURN_CODE      '  DSQ_RETURN_CODE
   say ' DSQ_REASON_CODE      '  DSQ_REASON_CODE
   say ' DSQ_ERROR_ID         '  DSQ_ERROR_ID
   say ' DSQ_MESSAGE_ID       '  DSQ_MESSAGE_ID
   say
   pause
   signal ENDIT
```

The interface is started in batch mode.

A query is run that produces a report. The INTERACT parm is set to NO to tell Query Manager not to show a screen. Without this parm, the QMCI will fail by attempting to display a screen while in batch mode.

The report is sent to the print queue. No Query Manager screens have been shown.

Figure 14. Batch QMCI Application

Conclusion

This article has introduced the interfaces to Procedures Language 2/REXX shown with examples of how powerful they can be for both tool and application development. Further examples and complete discussion of the syntax involved is available in the OS/2 Extended Edition Version 1.2 manuals.

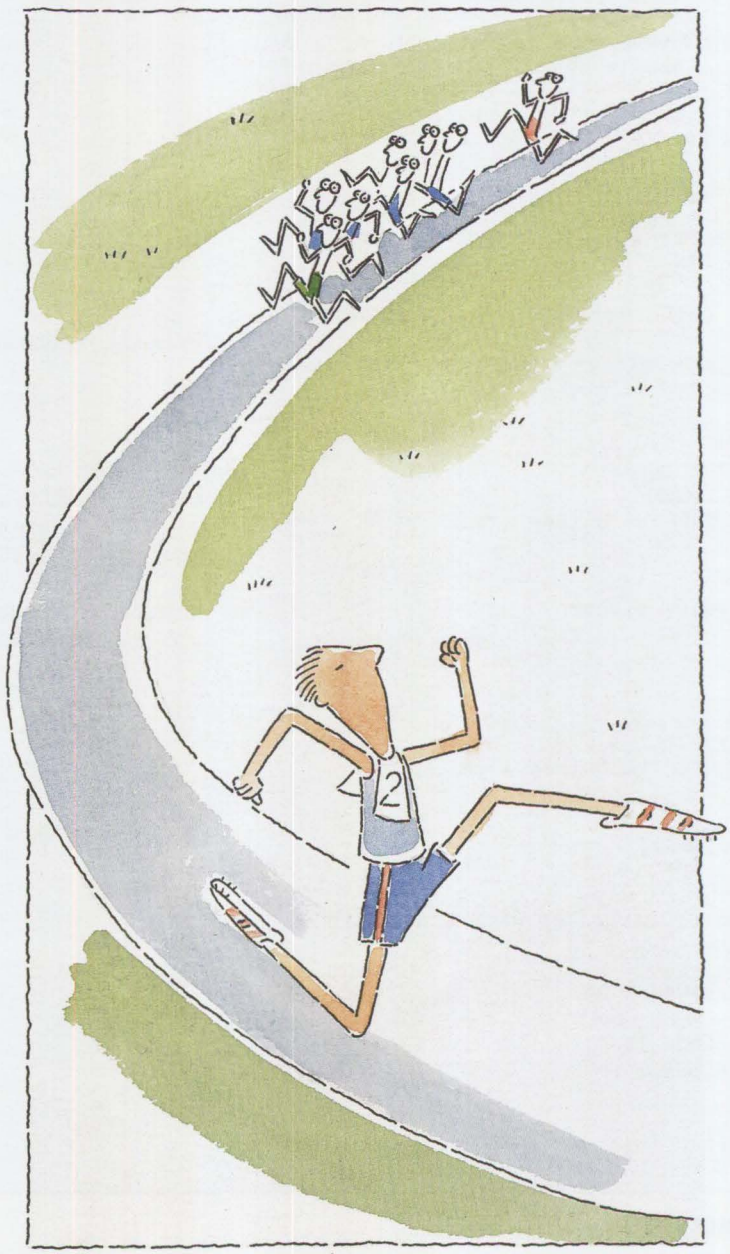
ABOUT THE AUTHOR

Jeffrey W. Fisher is a market support representative in IBM's Desktop Systems Support Center in Dallas. Jeffrey joined IBM in May 1989, and brings extensive COBOL, CICS, and DB2 application development experience to the OS/2 technical support environment. He is currently concentrating on support for the OS/2 Database Manager.

APPC Performance Tips for OS/2 EE

John Q. Walker II and
Gregory M. Callis
IBM Corporation
Research Triangle Park,
North Carolina

Three main areas for performance improvement are considered in this article: (1) configuring APPC and the Data Link Control components (DLCs) in the OS/2 Communications Manager; (2) designing and writing APPC transaction programs; and (3) providing a hospitable OS/2 environment when running programs.



Advanced Program-to-Program Communications (APPC) software is an integral part of the OS/2 Extended Edition (EE) Communications Manager. It implements the System Network Architecture (SNA) Logical Unit (LU) type 6.2. This provides an application programming interface to OS/2 programs, which allows them to exchange information with similar programs located anywhere in an SNA network.

This article suggests steps that can be taken to make APPC applications run as fast as possible in OS/2. It makes no claims to consider the performance of other concurrent OS/2 processes or the hardware of the machines. It also makes no claims as to the performance gains to be expected by making any of these changes. Actual performance gains will be dependent on the particular configuration. The steps listed here apply to OS/2 EE Version 1.2 and earlier; they may not necessarily apply to any future version of APPC in OS/2 EE.

As is often the case, there are many trade-offs that must be taken into consideration – program size, memory costs, usability, and simplicity – to gain performance. Those aspects should be considered when making any of the changes suggested here.

Some Background APPC Information

The Communications Manager portion of IBM's OS/2 EE contains a variety of software components to handle computer communications. One of these components is Advanced Program-to-Program Communications. APPC is an implementation of LU-type 6.2, which is one of the primary parts of

IBM's Systems Network Architecture (SNA). The *IBM Systems Network Architecture Technical Overview* (GC30-3073) provides a technical overview of SNA.

Programmers would normally write a pair of communications programs in such a way that when a local program starts a remote program, they would begin exchanging information – that is, they would enter into a *conversation.* These programs can request a number of services from APPC; to use these services they issue *verbs.* APPC verbs have names such as **ALLOCATE** (to set up a conversation), **SEND_DATA** (to send data to a remote program), and **RECEIVE_AND_WAIT** (to wait for data to arrive from a remote program). When APPC finishes its work on a verb, it sets a return-code field and returns to the calling program. If the verb has executed successfully, the return code indicates "OK." Otherwise, the return code shows why APPC was unable to satisfy the verb request.

A logical unit is a piece of internal APPC software that accepts and processes verbs. It acts on behalf of a program that is issuing verbs to it. To exchange data between a pair of machines, a local LU maintains one or more *sessions* with a remote LU. Each conversation uses one session, which serves to transport the data between two logical units. Generally, the local and remote LUs are located in two different machines, but with a multitasking operating system such as OS/2, both LUs may be physically located in the same machine.

The term *transaction program* (TP) has a special meaning in APPC, and it is quite different from a "program" as we know it in OS/2. A TP is not an OS/2 program; it is a section of an OS/2 program. Unfortunately, these names are a little confusing!

Figure 1 shows an OS/2 program with multiple TPs using the APPC and common services components of the OS/2 Communications Manager.

Starting a TP is simply a matter of issuing one of two verbs. When an OS/2 program successfully issues a **TP_STARTED** or **RECEIVE_ALLOCATE** verb, it identifies itself as a new entity (a TP) that APPC needs to know about. APPC internally sets aside a group of memory blocks for the TP, and devises a unique TP identifier (a **tp_id**), which it returns to the calling program.

An OS/2 program must supply that **tp_id** on all conversation verbs it issues while part of that TP. Many **tp_id**s may be active at any time in an OS/2 process, but a **tp_id** cannot be shared with other processes. If an OS/2 program issues a **TP_ENDED** verb, APPC internally clears its buffers for that TP and

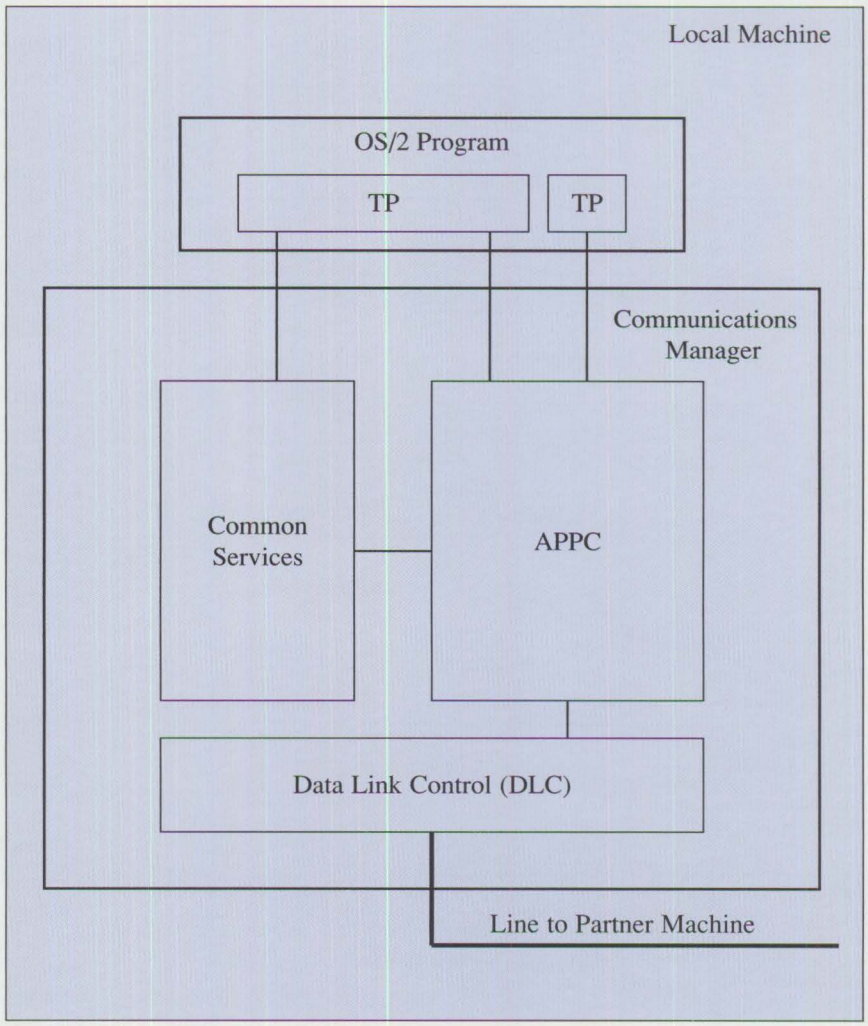


Figure 1. An OS/2 Program with Multiple Transition Programs (TPs)

marks the **tp_id** as invalid. If an OS/2 program terminates, APPC ends all the active TPs associated with that process.

Conversations similarly have their own identifiers (known as **conv_id**s), and a TP must be established before a conversation can be allocated. Many conversations may use a single **tp_id** concurrently (such as in multiple threads) or sequentially (where one conversation follows another). When a TP ends, APPC deallocates all of its active conversations. Figure 2 shows three nodes connected through an SNA network.

There are two types of conversations: basic and mapped. Basic conversation verbs require the program to put a two-byte length field at the beginning of each data block. Some of the basic conversation verbs have extra parameters that the mapped verbs do not, allowing some additional flexibility. By contrast, a mapped conversation provides the features required for information exchange in an easy-to-use, record-level manner.

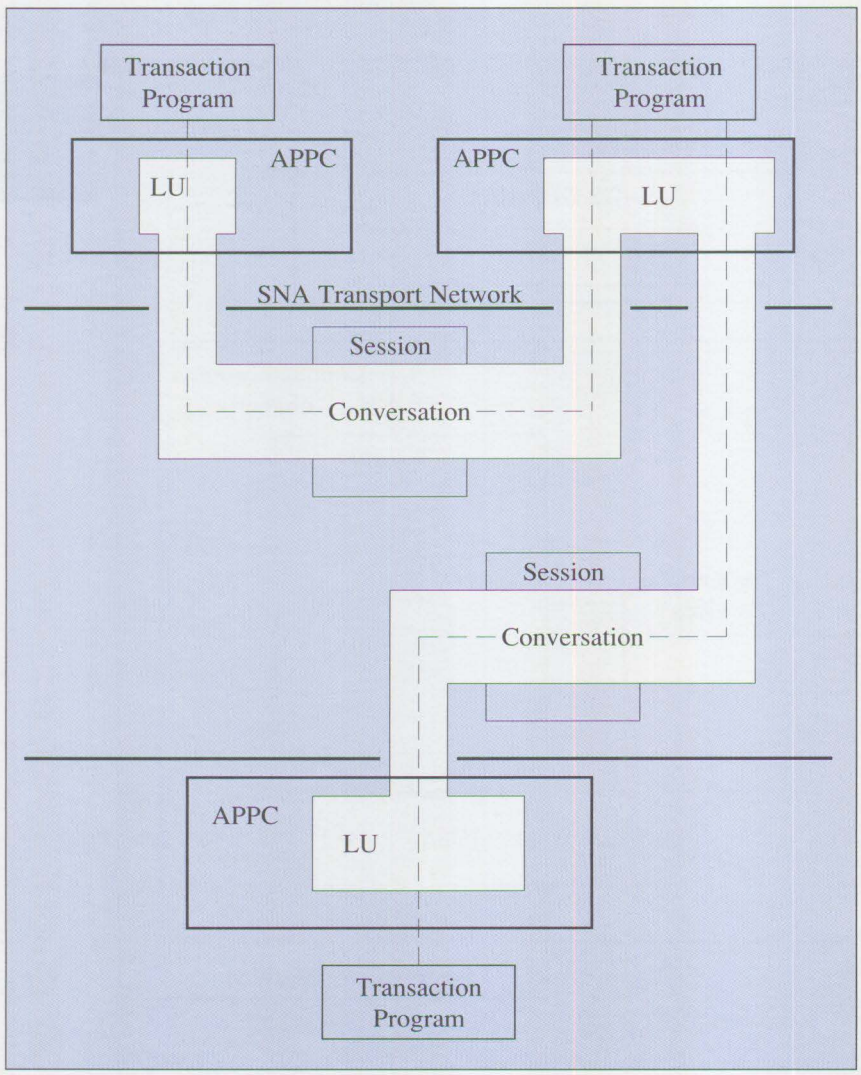


Figure 2. A Sample SNA Network Connection

In previous paragraphs, we spoke of APPC verbs named **ALLOCATE, SEND_DATA**, and so on. These are actually basic conversation verbs; their mapped conversation counterparts are named **MC_ALLOCATE** and **MC_SEND_DATA**. In the remainder of this paper, whenever either a basic or mapped form of a verb is allowed, it is signified by surrounding the "MC_" with square brackets; for example, **[MC_]SEND_DATA** signifies both **SEND_DATA** and **MC_SEND_DATA**.

There is more information on APPC concepts and operation in the *IBM Systems Network Architecture Transaction Programmer's Reference Manual for LU Type 6.2 (GC30-3084)*. The *IBM Operating System/2 Extended Edition APPC Programming Reference (S90X-7910)* specifically describes how to use APPC with OS/2 EE.

Configuring the Communications Manager

A series of menus and panels are used to set up the data link control (DLC), LUs, sessions, and TPs in OS/2. Performing this setup is known as "configuring APPC"; when configuring setup is complete, the values entered are verified to see that they properly cross-reference each other. APPC's use of memory and DLCs is tailored through configuration.

Complete details on configuring the OS/2 EE Communications Manager are found in the *IBM Operating System/2 Extended Edition System Administrator's Guide* (S90X-7908). The following steps can im-

prove APPC throughput without needing to change programs.

Step 1: Provide appropriate transmit and receive buffer sizes for the DLC.

a. Use as much as possible of the data buffering capability of the LAN adapter when configuring the "LAN Feature Profile."

   Using large transmit buffers allows large Request/Response Unit (RU) sizes to be configured for APPC. Large RUs dramatically increase throughput; we have seen performance improve by a factor of nine in our labs. The trade-off in using large transmit buffers is that more adapter and system memory is consumed, and some degradation in short transactions may be experienced. The buffer sizes are read from the configuration file when OS/2 is booted, since the configuration file is named in CONFIG.SYS. Thus, these buffer sizes are used by all the communications software using that LAN adapter, not just APPC.

   For example, set the maximum RU size to 1,500 bytes if using an Ethernet Adapter, 1,920 bytes if using an IBM Token-Ring Adapter or Token Ring Adapter/A, or as high as 15,360 bytes if using Token Ring 16/4 adapters. The Transmit Buffer Size should be set to the maximum RU size + 24. The Receive Buffer Size should be set to the minimum RU size (set in the Mode profile) + 24.

   Why is 1920 the optimal RU size for 4Mbps token-ring adapters with 2 KB buffers? The size of the RU negotiated during session setup is of the form

   $$a \times 2^b$$

   where a and b are positive integers, and a is between 8 and 15. Note that

   $$1920 = 15 \times 2^7$$

   Because 24 additional bytes are needed for frame overhead,

   $$1920 + 24 = 1944 <= 2048 \text{ (which is } 8 \times 2^8\text{)}$$

   1920 is the largest number of the proper form less than 2048, which is the hardware buffer size. With a different hardware buffer size on the adapter, this formula can be used to calculate what the largest RU size can be.

   Although the IBM Token-Ring 16/4 Adapter can support a Transmit Buffer Size of 16 KB, configuring the largest Transmit Buffer Size may result in slightly poorer performance than using a value around 15 KB; having two 16 KB transmit buffers leaves less than 32 KB for the receive buffers. If the available receive buffer space is inadequate to handle an incoming frame, the frame is rejected and must be retransmitted.

b. The minimum number of receive buffers must be between 2 and 151. For best performance, use the result of the following equation:

$$\#\,Receive\ Buffers = \frac{(Transmit\ Buffer\ Size \times 2)}{Receive\ Buffer\ Size}$$

   Because the improvement can be so dramatic, you should experiment with different sizes of transmit and receive buffers for the particular environment. One user who reported a significant improvement by using large transmit and receive buffers tried the following test with his server machine. After configuring the receive buffers as previously described, he then configured twice as many receive buffers, each half the previous size. He reported a performance improvement of another 30 percent.

c. The maximum number of link stations for the LAN adapter should be minimized, because this serves to multiply the adapter memory needed for transmit and receive buffers. For each adapter, coordinate the Maximum Number of Link Stations configuration among three places: the "Data Link Control (DLC) profile," and the IEEE 802.2 and NETBIOS profiles in the "LAN Feature profile." If NETBIOS is not being used, set its Maximum Number of Link Stations to 0 to simplify the calculation.

d. SDLC and X.25 links sometimes encounter a relatively high level of line noise, so the optimal buffer size depends on the error rate of the link. If the error rate is high, using large frames will slow performance, because they may frequently need to be retransmitted. An extended discussion of the trade-offs between buffer size and bit error rate for SDLC can be found in the articles by K.C. Traynham and R.F. Steen, "Interpreting SDLC Throughput Efficiency, Part 1 – 3 Models and Part 2 – Results," *Data Communications*, October and November 1977, pages 43-51 and 59-66.

Step 2: Match the *Maximum RU size* on the Mode to the *Maximum RU size* on the DLC.

To minimize the number of line flows, make these sizes equal to each other, and as large as possible. The mode's RU size determines the size of a data block exchanged on a session; the DLC's RU size determines the size of a data block exchanged on a link. Many sessions may concurrently use one link. The Communications Manager Verify will not allow the *Maximum RU size* on the "Transmission Service Mode profile" to exceed the *Maximum RU size* on the "Data Link Control (DLC) profile."

Step 3: Use the default DLC window sizes.

For best performance, the Data Link Control should send and receive as much data as possible before requiring an acknowledgment. DLC pacing takes place on each link between a pair of DLC adapters. It specifies the number of frames that the local DLC can send and receive before requiring an acknowledgment from the remote DLC.

Intuitively, one would think that the *Send window count* and *Receive window count* fields should be set to their largest configurable values. However, the equation is much more complex – especially for LANs – involving the topology, traffic, frame size, load on the remote machine, and these two window counts. For the token-ring LAN, it also involves the value for the T1 Acknowledgment Timer. So, although they may seem a little counter-intuitive, the default values in the DLC configuration have been demonstrated in our labs to give the best performance, considering all of these factors. We recommend experimenting with this value among the computers in individual environments.

Note: Depending on the capabilities of the remote machine, the window counts may need to be configured to match exactly those configured at the remote machine.

Step 4: Use moderately sized pacing windows for the sessions.

Session pacing is used to tell the remote LU how many RUs it can send before the local buffers might be overrun. For example, if the receive pacing window is "8," the remote LU will stop sending after the eighth RU – until the local LU gives permission to again begin sending. Best performance for a session occurs when APPC can send and receive as much data as possible before requiring a message exchange with the remote LU.

Ideally, one would configure for no pacing, that is, set the *Receive Pacing Limit* field in the "Transmission Service Mode profile" to zero (0). However, APPC can abend if too much "unpaced" data arrives and its memory allocation is exceeded. Using no pacing can also be harmful to the SNA network by overrunning the buffers of machines doing intermediate routing.

Instead, set the *Receive Pacing Limit* field for the mode to its default value, which is "8." Although it can be set to anything up to 63, the performance gain above about eight appears to be negligible, and the additional buffers that APPC would need are allocated from the available OS/2 memory. This value will be negotiated with the remote LU when the session is established, so coordinate the session pacing window values in both machines.

However, if many sessions are active with large pacing windows, OS/2 may begin swapping to disk, because of lack of memory. These concerns will need to be balanced in the environment in which the session is running.

Step 5: Avoid session contention, when configuring parallel sessions.

Determine the number of contention winners and losers for each side, and configure these appropriately on each side. Contention winners and losers are important when parallel sessions are configured between a pair of LUs. For best performance, the machine issuing an **[MC_]ALLOCATE** verb should always get a contention winner session.

For example, a *Partner LU session limit* might be configured as 10. This means 10 sessions may exist in parallel between two LUs. The contention winners and losers values determine which LU owns each of these 10 sessions. If the local LU has been configured with its *Minimum number of contention winners source* (that is, its contention winners) as five and its *Minimum number of contention winners target* (that is, its contention losers) as two, then three sessions are "up for grabs." If it is known which LU should own each of the sessions, configure the two LUs to agree. In this example, the 10 sessions might be divided among the two LUs, as follows:

Local LU		Remote LU	
winners:	6	winners:	4
losers:	4	losers:	6

Step 6: Auto-activate sessions, when possible.

Rather than waiting to bring up sessions when an **[MC_]ALLOCATE** verb is issued (which may delay the sending of data), bring up all the an-

ticipated sessions when APPC is started. To do this, specify a non-zero value for the *Number of automatically activated sessions* field (part of the "Initial Session Limit profile"). The value specified there should be less than or equal to the value configured for the contention winners.

In OS/2 EE Version 1.2, sessions can also be auto-activated using the **CNOS** management verb. The **CNOS** verb should be used sparingly: try issuing it once, to bring up the sessions for a given LU, partner LU, and mode when they are needed.

Step 7: Specify unique *Node ID* values on each machine.

Additionally, when using SDLC, configure one DLC as primary and its partner as secondary. When APPC activates a link, a pair of XID exchanges will be saved if the node IDs are unique and the link station roles are already established as primary and secondary. Carefully coordinate this configuration among all machines involved, because errors here can waste lots of time in troubleshooting. The *Node ID* is configured in the "SNA Base profile."

APPC attempts to activate a link in the following situations:

- when APPC is started,
- when a session is being established and the link it needs is not active,
- when the Subsystem Management panels are used to activate a link, or
- when the remote machine initiates the link activation. The partner is probably in one of the preceding three situations.

Step 8: Configure *Free Unused Links* as **No** when using a small number of links frequently.

The *Free Unused Links* field in the "Data Link Control (DLC) profile" specifies whether a link is automatically deactivated when no sessions are using it. If the TPs frequently bring up and take down sessions with a few partners, best performance can be obtained by leaving the link up when there are no sessions. The next time a session is started, the link is already active and is waiting to be used. This can be beneficial in a LAN environment, where the overhead to maintain a link is low.

If sessions are established with a great number of different partners, APPC must maintain many sets of control blocks for the different links – possibly degrading performance. Consider configuring *Free Unused Links* as **Yes** in this case, as well as when configuring switched SDLC profiles.

Step 9: Minimize the number of LUs, partner LUs and modes configured.

Many transaction programs (TPs) may use the same LU. This can be made particularly easy by writing the TPs to use the default LU (that is, specify the **LU_alias** as all zeros on the **TP_STARTED** verb). One partner LU configuration is needed for each remote machine, but the need for multiple partner LUs per remote machine can usually be minimized. Coordinate the mode names used with each remote machine to minimize the total number of modes required.

Designing and Writing TPs

The primary suggestion in this section is to minimize the number of calls to APPC or ACSSVC (for common services verbs). It is recommended that a transaction program be designed in such a way that it causes the least amount of code execution in APPC as possible.

Each time APPC or ACSSVC is called, the verb control block and (if present) the data buffer are checked to see that they are the proper length, have the proper segment attributes, and so on. When possible, combine functions in a single verb, and send and receive large blocks of data. Some specific ways to do this are described in the following suggestions.

1: Minimize changing conversation states between Send and Receive.

The underlying nature of LU6.2 is inherently "half-duplex;" that is, only one side of a conversation sends at a time. Here are a couple of ways to minimize the number of state changes:

a. Send as much data as possible before receiving or requesting a confirmation.

b. Programs can be designed to use two simultaneous conversations, one sending and one receiving, each on its own session. Figure 3 shows one user's design, where each thread contains a TP that is dedicated to either sending or receiving data.

Be careful that your programs do not become too complex in your efforts to avoid a state change.

2: Use "combined-receive" indicators for the **RECEIVE** verbs.

In OS/2 EE Version 1.2, APPC can return both data and status to a TP using a single **RECEIVE** verb. (In earlier versions, two **RECEIVE** verbs were required: one for the data and one for the status.) Use this capability freely on all **RECEIVE** verbs.

3: Use the **type** parameter on **[MC_]SEND_DATA** to combine operations.

In OS/2 EE Version 1.2, a TP can combine other verb operations with each **[MC_]SEND_DATA**, such as FLUSH, CONFIRM, and DEALLOCATE. This can often save an extra call to APPC.

4: Minimize use of the **[MC_]CONFIRM** verb and **sync_level**(CONFIRM).

[MC_]CONFIRM forces a transaction program to wait for an explicit acknowledgment from the partner. Don't use the confirm functions unless positive acknowledgment is really required. Instead of asking for confirmation after each Send, the paired TPs should be written to issue an **[MC_]SEND_ERROR** only when they detect some failure in the data being transmitted.

5: Minimize use of the **[MC_]SEND_ERROR** verb.

[MC_]SEND_ERROR is used to notify the transaction program at the remote location that an error has been detected.
[MC_]SEND_ERROR should not be part of the mainline Send-Receive path of the TPs. It should probably only be used when either partner detects a real error situation.

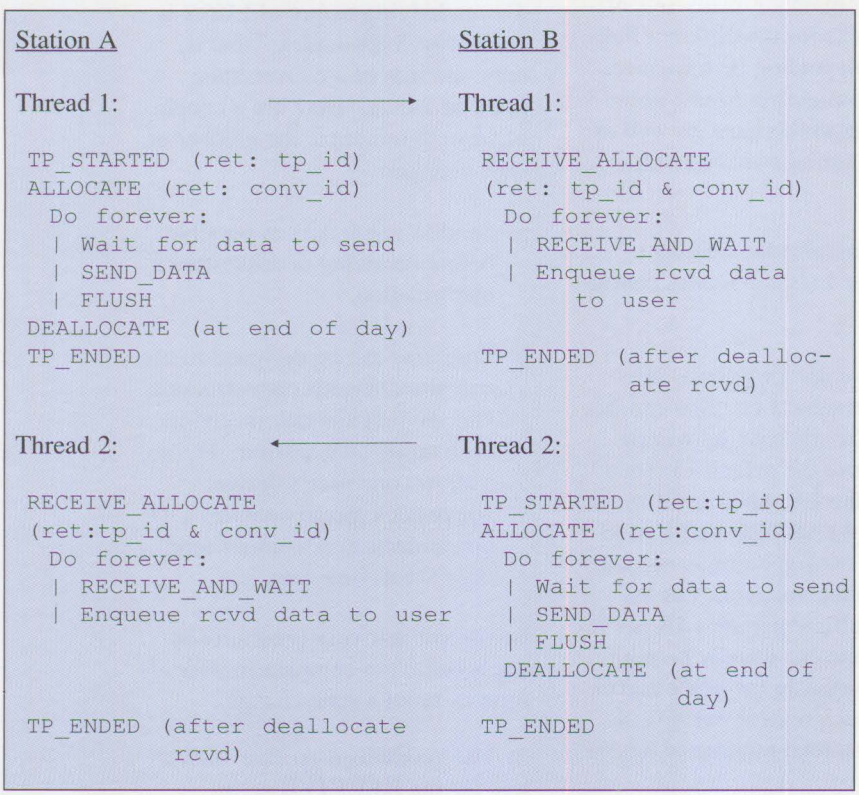


Figure 3. Two Workstations Using Two Simultaneous Conversations

In addition to causing two internal message exchanges with the partner, issuing an **[MC_]SEND_ERROR** verb causes APPC to write an entry to the Communications Manager error log.

6: Minimize use of the **[MC_]REQUEST_TO_SEND** and **[MC_]PREPARE_TO_RECEIVE** verbs.

When possible, the local and remote TPs should be written so that they are aware of the current state of the conversation, and change their Send and Receive states when appropriate. The state of the conversation is known at all times, based on which verb was issued and the return codes and **what_received** indicators that are returned. The *APPC Programming Reference* describes these states for each verb, indicator, and return code.

7: Minimize use of the **[MC_]RECEIVE_AND_POST** verb.

Every **[MC_]RECEIVE_AND_POST** issued by the program results in the creation of a thread, which is then destroyed after the receive has been satisfied. So if **[MC_]RECEIVE_AND_POST** is used to save threads or improve performance, consider creating an individual thread and issuing **[MC_]RECEIVE_AND_WAIT** verbs on that thread.

8: Minimize use of the **GET_TYPE**, **[MC_]GET_ATTRIBUTES**, and **DISPLAY** verbs.

TPs should know most of this information already. If these verbs are called, their returned parameters should be kept in local program variables for later use as required.

In OS/2 EE version 1.2, the **DISPLAY** verb returns a superset of the information returned on the **GET_TYPE** and **[MC_]GET_ATTRIBUTES** verbs. If there is a choice between using **DISPLAY** or **[MC_]GET_ATTRIBUTES** (for example, only the network name is desired), use **[MC_]GET_ATTRIBUTES**.

These verbs may prove to be helpful on error paths, so don't omit them entirely. For example, if a return code indicates that a verb was issued in the wrong state, the **DISPLAY** verb can be issued to find the current state of the conversation.

9: Use the **[MC_]FLUSH** verb judiciously.

[MC_]FLUSH forces APPC to send the data in its internal buffers, even though those buffers may not be full. Unless the partner needs the data immediately, let APPC determine when to send the data.

An example of a situation in which **[MC_]FLUSH** would not be beneficial is when multiple **[MC_]SEND_DATA** verbs are issued. APPC normally fills its buffers before sending. If the maximum frame size is 2 KB and each **[MC_]SEND_DATA** is 1 KB long, flushing the buffers would double the number of transmissions and leave the buffers only 50 percent utilized.

If the TP issues an **[MC_]SEND_DATA** verb and then plans on issuing no verbs for awhile, it should probably issue a **[MC_]FLUSH** verb to clear its buffers, so the data can be used by the remote TP.

10: Send many complete logical records with one verb.

When using basic conversations, send as much data as possible on each **SEND_DATA**. If the data is available (and less than 64 KB in total size), the program should send multiple logical records with a single **SEND_DATA** verb. However, avoid splitting logical records across verbs.

11: Receive as much data as possible on each Receive verb.

Allocate a large memory block for incoming data, and receive as much data as possible on each Receive verb: **[MC_]RECEIVE_AND_POST**, **[MC_]RECEIVE_AND_WAIT**, or **[MC_]RECEIVE_IMMEDIATE**.

When using basic conversation verbs, specify **fill**(BUFFER) on the Receive verbs, instead of **fill**(LL). With **fill**(BUFFER), the TP can receive multiple logical records with a single call to APPC.

12: Minimize the number of parameters set in each verb control block.

a. If a field in a verb control block serves only as a "returned parameter," it does not need to be set to any value before each call. The primary and secondary return codes, for example, are returned parameters that APPC does not examine on the call; it only sets them upon return.

b. The **tp_id** and **conv_id** are needed in all conversation verbs after the initial **[MC_]ALLOCATE** or **RECEIVE_ALLOCATE**. If the TP reuses the same verb control block for all verbs in a conversation, it should not need to set the **tp_id** and **conv_id** before each call to APPC.

c. Whether the verb is basic or mapped is a supplied parameter on all conversation verbs. If the verb control block is reused, a program can set the conversation type once at the beginning of the conversation, and not change it for the subsequent verbs on that conversation.

Remember to set all *reserved* fields to zeros before calling APPC.

13: Use a minimum number of segments for the data buffers in the TP.

Each different segment used for a data buffer incurs extra APPC overhead. Once APPC has been called with the address of a data buffer, APPC does not free the segment used for a data buffer until the OS/2 process APPC was called in is terminated. Reuse the same data buffer segments, when possible.

14: Convert names from ASCII to EBCDIC once.

For parameters such as **tp_name** and **mode_name**, make the calls to the **CONVERT** verb once at the beginning of the program, and save the converted names for later use.

15: Minimize the use of **log_data** and **pip_data**.

The implementation of APPC on some machines may require the use of **pip_data** to perform initialization. Similarly, some transactions may depend upon a transfer of **log_data**. But, in general, the information exchanged with these parameters (on the **DEALLOCATE**, **SEND_ERROR**, and **[MC_]ALLOCATE** verbs) can probably be handled in a more efficient manner by the two TPs.

16: Have multiple conversations serially reuse a session.

Sessions can be expensive to bring up and take down, especially since links may need to be activated. Reuse sessions, if possible.

17: Keep conversations active when using APPC frequently.

APPC sets aside internal buffers and control blocks each time a conversation is allocated, and frees these each time a conversation is deallocated. If there is no contention for a session, keep a running conversation active for as long as it is needed – then deallocate it as soon as the TP has finished with it.

However, by following this suggestion in a simple-minded manner, performance can actually be degraded. When APPC allocates the control blocks necessary for a conversation, this memory is no longer available for other use in the system. As the number of conversations increases, the available system memory can decrease, causing excessive swapping on either machine. This can especially be a problem on some remote machines with operating systems other than OS/2. If there are long pauses between a program's use of APPC, it may be better to deallocate and later reallocate a conversation.

In some APPC implementations on other machines, the attach overhead (that is, the overhead of processing an ALLOCATE issued by the partner) is so large that it is expensive to issue DEALLOCATE for anything less than a disaster. Here are a few trade-offs to consider.

- Consider the partner's attach overhead. The time to process an incoming allocation request is relatively fast for APPC/PC (for PC-DOS), APPC on OS/2, and APPC on the RT PC. This processing time is reasonable on CICS, although CICS has been optimized for attaching LU type 0 and LU type 2 transactions from financial terminals or 3270 terminals. The attach processing in the System/38 and the AS/400 appears to be somewhat slower. As with anything, performance may vary.

- Consider the number of sessions required. If the application maintains a conversation for an indefinite period, other applications will need their own sessions. If there is a chance that other applications will use an LU to communicate with applications serviced by the partner LU, they may be forced to wait for a session.

- Consider processing delays. If there is a long delay, other applications may be able to utilize the network resources. If the delay is very short, deallocating the conversation may cause excess system processing.

- Consider the server resource utilization, when using APPC in a server-requester relationship. By deallocating the conversation, the partner can allocate the program space, buffers, and control blocks to other needy requesters.

- Finally, there's the cost of the link when trying to improve performance. Long-distance telephone calls over switched SDLC links usually mean that sessions and conversations should be designed to be as short as possible.

18: Deallocate only one side of a conversation.

Either side may deallocate a conversation; this terminates the conversation for both partners. If any verb in the local TP receives one of the five DEALLOCATE primary return codes, it can safely assume the conversation is over; the local TP does not need to spend time issuing a **[MC_]DEALLOCATE** verb. These primary return codes are:

```
0005    DEALLOC_ABEND
0006    DEALLOC_ABEND_PROG
0007    DEALLOC_ABEND_SVC
0008    DEALLOC_ABEND_TIMER
0009    DEALLOC_NORMAL
```

19: Avoid using one of the ABEND types on the **[MC_]DEALLOCATE** verb.

The **type**(FLUSH) supplied parameter is faster than the other types on the **[MC_]DEALLOCATE** verb. In addition to causing two internal message exchanges with the partner, issuing a **[MC_]DEALLOCATE** verb with one of the ABEND types causes APPC to write an entry to the Communications Manager error log.

20: Take down conversations before issuing **TP_ENDED**.

Issuing a **TP_ENDED** verb causes a **DEALLOCATE** with **type**(ABEND_PROG) to be issued for each active conversation. As previously mentioned, issuing a **[MC_]DEALLOCATE** verb with one of the ABEND types causes two internal message exchanges with the partner and an error log entry to be written.

21: Minimize use of **TP_ENDED** with **type**(HARD).

In OS/2 EE Version 1.2, issuing the **TP_ENDED** verb with **type**(HARD) causes all sessions

being used by the TP to be taken down. If these sessions are to be used again, they will have to be brought back up again.
TP_ENDED with **type**(SOFT) does not affect the sessions.

22: Use the performance advantages available through the compiler and linker.

A number of the features in the OS/2 compilers and linkers are designed to speed program execution. A short summary includes:

a. Enable as many compiler and linker command-line optimizations as possible. For example, with the IBM C/2 compiler, use the /G2 compiler flag to enable the machine instruction set for the 80286, 80386, and 80486 processors. Also, consider compiling with the /Ox option, which enables all compiler optimizations. We suggest testing the program both without any optimizations and with optimizations; unfortunately, the optimizations may introduce subtle runtime errors.

b. Use as small a memory model as possible. Four-byte pointers incur more than twice the cost of two-byte pointers. If the code and data can each fit in 64 KB segments, use the small memory model.

c. Declare variables in registers. Many compilers implicitly use registers for loop counters inside of loops; consider explicitly using registers for variables – especially pointers – that are frequently used in a given procedure.

d. Compile using the Pascal function-calling convention; it is slightly faster than the standard way function arguments are pushed and popped in the C language. For example, with the IBM C/2 compiler, the /Gc option causes all functions not marked with the *cdecl* keyword to be called using the Pascal calling convention.

e. Use macros instead of procedures for small code routines that are executed frequently. This is especially effective when a routine is called from inside a loop. A good candidate for using a macro is a loop of repeated calls to a procedure to send data (that is, the called procedure sets up a verb control block, calls APPC, and examines the return code). Using macros may make the size of the resulting program larger, depending upon how often a routine is called and how big it is.

The following items, 23 through 25, do not have as much effect on performance as those previously mentioned. Programs probably should not be restructured or their usability reduced to take advantage of these.

23: Use basic conversation verbs.

Mapped conversation verbs, while easier to use, require slightly more overhead inside APPC; the best we have seen is a 3.5-percent improvement in using basic conversation verbs rather than mapped in a one-to-one comparison. But, basic conversation verbs allow sending multiple logical records on each call, reducing the calls to APPC.

24: Have multiple conversations serially reuse a TP.

APPC incurs a cost for internally managing the buffers associated with a TP. Let several conversations serially use a single TP, whenever possible.

25: Use names that differ early in the characters they use.

APPC can internally compare names faster if their characters differ early in the name. For example, use the names "ABC" and "XYZ" instead of the names "XYZ01" and "XYZ02." This applies to LU names, mode names, TP names, passwords, and user IDs.

While this does not save much in APPC performance, many staff-hours can be saved in the long term by choosing good names the first time. The following paper gives some excellent guidelines to consider when choosing names in a network:

> D. Libes, "Choosing a Name for Your Computer," *Communications of the ACM,* 32, 11, November 1989, pages 1289 - 1291.

Using OS/2 Effectively

Because every system will probably be set up a little differently, we can't make specific recommendations on how to use OS/2, but here are a few guidelines.

1: Minimize OS/2 swapping to disk, whenever possible.

Calculate the storage required for OS/2, APPC, and other active features, using the guidelines described in the *System Administrator's Guide* and the *Information and Planning Guide*. Make sure that at least this much random access memory (RAM) is installed in the machine.

2: Set aside as much DISKCACHE memory as is reasonable.

Specify the DISKCACHE size, in bytes, in the CONFIG.SYS file. The amount of memory specified should be part of the calculation of required OS/2 memory, as previously described.

3: Minimize the number of threads the program creates.

Do not create more threads than necessary to do the work needed by the OS/2 program. This avoids the cost in OS/2 of task switching among the threads.

4: Avoid starving incoming and outgoing DLC traffic.

When processing verbs, APPC runs on the thread of the calling program. When processing incoming and outgoing data, APPC runs on the thread of the DLC device driver. Hard loops of verb calls at the API can delay the task switch needed to handle incoming or outgoing data for any active TP.

For example, avoid a hard loop of unsuccessful **[MC_]RECEIVE_IMMEDIATE** verbs in the TP. There are a couple of ways to do this:

- Periodically make a call to DosSleep (with a short sleep period) to give APPC a chance to handle the data sooner.

- Replace the **[MC_]RECEIVE_IMMEDIATE** verb with a **[MC_]RECEIVE_AND_POST** verb, and then loop on DosSemWait with a non-zero timeout value specified on the DosSemWait function call. This incurs less total overhead than a loop with DosSleep and **[MC_]RECEIVE_IMMEDIATE** and the non-zero timeout still allows other threads the time to run.

5: Avoid the time-critical priority class.

Any OS/2 program can use the DosSetPrty function call to change its own priority within the system. There are three priority classes, numbered from 1 (lowest) to 3 (highest). Number 3 is the time-critical priority. Using time-critical priority disturbs the balance between handling user requests (for example, keyboard or mouse input) and incoming and outgoing data. Its use with APPC is not recommended.

Examples of using the OS/2 priorities to fine-tune the multitasking behavior of programs are given in the articles:

> M.J. Minasi, "OS/2 Notebook: Getting Your Priorities Straight," *Byte*, **14**, 12, November 1989, pages 159-162.
>
> M.J. Minasi, "OS/2 Notebook: OS/2 Multitasking Revisited," *Byte*, **14**, 13, December 1989, pages 133-136.

The second article shows how one child program can run 100 times faster than another by changing the parameters on the DosSetPrty function call.

6: Determine whether remotely started programs run in the foreground.

If a remotely started OS/2 program (that is, a program started because of an arriving allocation request from its partner) does not require interaction with a user, write the program so that it can be run in the background. Screen I/O, whether in a text-based screen group or in a Presentation Manager window, can slow down data communications. However, a program that is visible on the screen has foreground priority, and runs about 25 percent faster than the same program if it was not visible. If it does minimal I/O, its greatest cost is therefore the cost in OS/2 of creating and displaying a new window or screen group.

It is advisable to configure a program to do screen I/O (see the *Program Type* field in the "Remotely Attachable Transaction Program profile") while testing and debugging it; then later change the configuration and program so that it runs in the background with no I/O after it is debugged.

Using Traces to Estimate Timings

Because APPC performance depends on so many variables (such as the CPU speed, the amount of RAM, the OS/2 EE version, the DLC being used, and the other programs running), no one can say how long a given verb will take to execute on a system. However, the Communication Manager *Trace Services* has an easy way to get timings of each verb issued by a program.

Activate the "APPC API trace," using the **DEFINE_TRACE** common services verb or the *Trace Services* panels. Run whichever transaction programs you have that should be timed. Whenever the program calls APPC, APPC writes the verb control block to the trace data; similarly, when APPC returns to the program, it writes the returned verb control block. These trace records contain a time stamp, with an accuracy of hundredths of a second. By subtracting the time stamp of the API Request in the trace from the time stamp of the API Return, the duration of a verb can be roughly calculated. How much time the pro-

gram spends between its calls to APPC can similarly be estimated.

To find the amount of time spent exchanging data, trace the data for the DLC being used by the programs. Full directions on how to set up tracing and read the resulting trace records are in the *Programming Services and Advanced Problem Determination for Communications.*

Tracing slows the overall execution time, so when not tracing, the program will naturally run faster. The tracing speed can be increased by changing the "trace truncation" length from its default of 0 – which means writing the entire control or data block to the trace – to a small number such as 16. Also, note that the resolution of the OS/2 clock is not as granular as 1/100th of a second, which may skew the timings somewhat. Finally, all APPC calls by all active transaction programs will be in the trace together, so be aware that the trace records may be interleaved for concurrent TPs. Look at the **tp_id** or **conv_id** to tell which program a trace record is for.

Summary

- Use large RUs and buffers for the DLCs and sessions.
- Minimize flows needed for negotiation of values by careful configuration.
- Bring up links and sessions early, so they do not have to be brought up by the TP.
- Reuse sessions once they are established.
- Minimize useless calls to APPC.
- Combine functions in a single verb.
- Use large send and receive data buffers.
- Reuse data buffers and verb control blocks.
- Use the available compiler and linker optimizations.
- Minimize disk swapping and task switching in OS/2.
- Use the Communications Manager *Trace Services* to help estimate where the program is spending its time.

ACKNOWLEDGMENTS

Many thanks for the expert technical help provided by Tim Holck, Mike Kawalec, and Pat Scherer, who have worked hard to make APPC ever faster in OS/2 EE.

ABOUT THE AUTHORS

John Q. Walker II is an advisory programmer at IBM Research Triangle Park, North Carolina, responsible for Advanced Program-to-Program Communications for OS/2 EE. He joined IBM in Rochester, Minnesota, in 1978, where he was involved with the development and testing of the operating system software for System/38. In RTP, John was an architect for the IBM Token-Ring Network, serving as a co-editor of IEEE 802.5 local area network standards, 1983-1984. He received a B.S., B.A., and M.S. from Southern Illinois University and is completing a Ph.D. in computer science at the University of North Carolina at Chapel Hill.

Greg Callis is a staff programmer at IBM Research Triangle Park, North Carolina, where he is a lead developer of Advanced Program-to-Program Communications for OS/2 EE. He joined IBM at Research Triangle Park in 1984 to develop the APPC/PC product for PC-DOS. Greg received a B.S. in computer engineering from Auburn University.

EASEL OS/2 EE PROFS: Host Code Interface

Clayton Arndt
IBM Corporation
Dallas, Texas

This article presents a sample program written in EASEL for OS/2 EE. The code is used to "automate" the logon process to the PROFS application on a VM host, and to logoff from the host.

EASEL for OS/2 EE is a highly productive migration and programming tool to help customers implement programmable workstations.

EASEL Development allows users to produce graphical front ends on an OS/2 Extended Edition Programmable Workstation to replace the 3270 user interface of existing host applications. Additional application logic can also be programmed to add new workstation function while the existing host 3270 applications continue to run unchanged.

EASEL Runtime executes these new front-end applications on an OS/2 Extended Edition Programmable Workstation. The front-end applications developed with EASEL will conform to the Common User Access (CUA) guidelines of Systems Application Architecture (SAA). EASEL uses OS/2 EE Communications Manager and Presentation Manager facilities to provide both a productive programming and execution environment.

Highlights of EASEL:

- Facilities to develop and execute a graphic user interface that conforms to the CUA Graphics Model for OS/2 EE users in addition to the 3270 user interface of existing host applications for 3270 terminal users
- Development and Runtime products
  - Interactive CUA screen layout facility that uses OS/2 Presentation Manager
  - High-level language that is block structured and event driven
  - Trace and debug facilities
  - Separately licensed runtime facility
- 3270 terminal emulation in the OS/2 Communications Manager
- Facilities to add logic at the programmable workstation without impacting existing 3270 host applications or users of those applications at 3270 terminal devices
- Facilities to implement graphic user interfaces that can support higher productivity
- National Language Support (NLS)

EASEL 3270 Host Support

EASEL interfaces with 3270 host applications through the OS/2 EE Communications Manager Emulation High Level Language Application Programming Interface (EHLLAPI). For the programmer's benefit, an interface to EHLLAPI is supplied by the EASEL 3270 support module. EHLLAPI functions are not directly callable from an EASEL program; instead, the EASEL 3270 module translates 3270 action routine calls into the appropriate EHLLAPI function calls. The 3270 action routines let the programmer:

- Emulate standard and special function keystrokes
- Find the current cursor position
- Monitor the Operator Information Area (OIA) for messages
- Get the attributes and length of a field on the screen
- Get the contents of a single field or an entire line from the screen
- Place text into a field with or without padding blanks
- Read screen contents, or a portion of any screen, into windows

EASEL can connect up to five host sessions, depending upon the configuration of the host line and of Communications Manager. EASEL can also disconnect previously connected host sessions in any order.

A description of the support provided by the EASEL 3270 module may be found in Appendix C of the *EASEL OS/2 EE Developer's Guide* (SC38-7034).

EASEL Host Programming – FIELDS Utility

The EASEL Development package includes a FIELDS utility for mapping the 3270 host screens to determine the starting and stopping locations of fields on the screen and the attributes associated with each field. For example, the programmer analyzes the host screen currently displayed in the 3270 Terminal Emulation session "B" by entering the following command at the OS/2

screen prompt:

fields -sB profs.fld

The "-s" option indicates that the report is to be generated for the terminal session represented by the specified single-letter short session name (in this case, session "B"). This parameter defaults to the first session. It is recommended that this parameter always be specified when Communications Manager is configured for more than one session.

The parameter "profs.fld" is simply the name of the ASCII text file into which the programmer wants the FIELDS output to be written. Any valid OS/2 filename can be substituted. If a file name is not given, the results are sent to the workstation display.

Running the FIELDS utility against the previously mentioned host screen will produce the output shown in Figure 1.

The field report lists the session name at the top and displays a duplicate image of the host screen. This is followed by the total field count on the screen and the current cursor position. Next, the field summary describes each of the fields found on the screen:

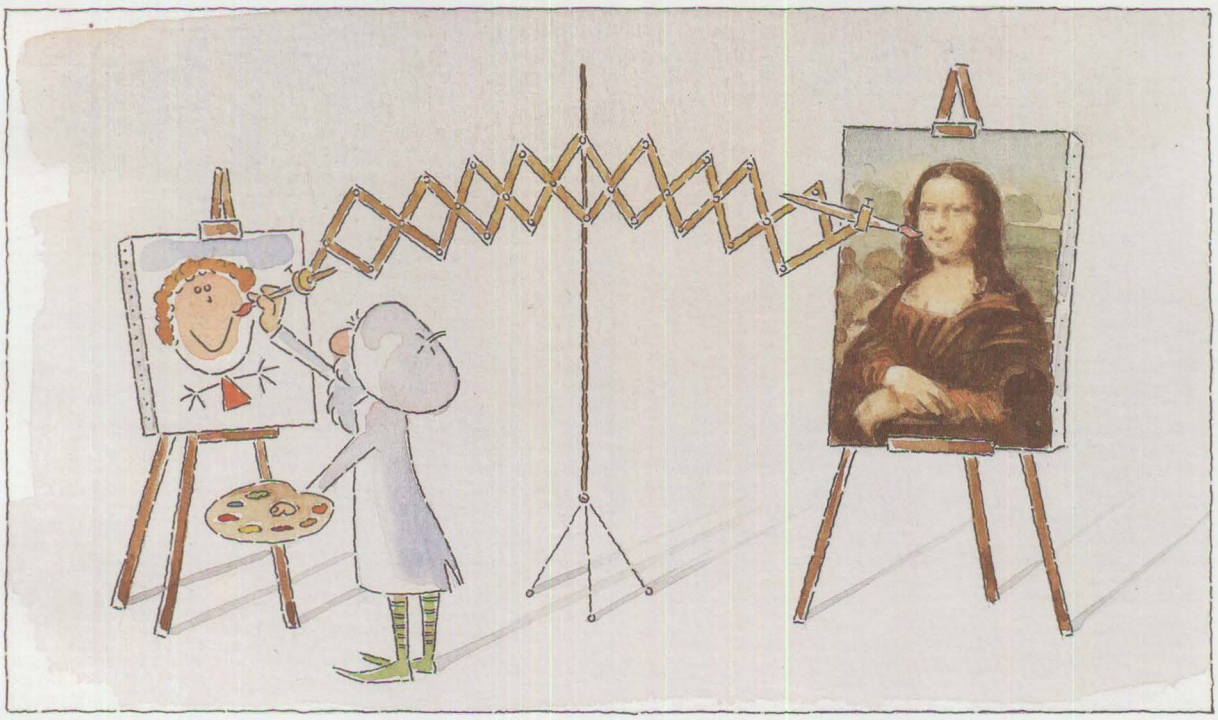
- Field number
- Starting coordinates
- Ending coordinates
- Length of field (characters)
- Field attributes: unprotected, numeric, nondisplay, modified
- First 30 characters found in the field

This summary shows the programmer which fields are fixed and which can be filled in with information. In addition, each field can be referenced in the EASEL program by the field number listed in the report, or the coordinates of the field may be used as the basis for interaction. The EASEL program presented later in this article shows how the contents of a field can be used to determine which host screen is currently displayed.

Developing the EASEL Host Interface

The programmer developing the EASEL interface to the host application should have a thorough knowledge of the structure and flow. It will be necessary to supply logic in the EASEL code to handle all anticipated screens. The FIELDS utility can be used to make reports of the structure and contents of each host screen.

Host screens that are dynamically generated (that is, the contents are not statically fixed) present a different problem. Here the FIELDS utility would not be useful because the number, location, and content of the fields can change as the host application progresses. For dynamic screens, EASEL can read the entire screen into a textual region and can then scan through the region, searching for keyword(s) that indicate the desired information.



```
FIELDS utility - Version H1.20  (HLLAPI)
Copyright (C) 1987,1988,1989 Interactive Images, Inc.
All Rights Reserved.

Session short name = B

000000000111111111122222222223333333333444444444455555555556666666666777777777778
123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
================================================================================
 VIRTUAL MACHINE/SYSTEM PRODUCT


                      W E L C O M E  T O  T H E
                   ========= ==========  ====        ====
                   ========= =========== =====       =====
                      ===     ====   ====  ======   ======
                      ===     ==========   ======= =======
                      ===     ==========   === ======= ===
                      ===     ====   ====  ===  =====  ===
                   ========= =========== ====   ===   ====
                   ========= ==========  ====    =    ====
                  M & D SE REGION I/S SUPPORT SERVICES NETWORK
                  This system is restricted to IBM management
                  approved business purposes only.
                  ENTER: VMEXIT to return to the initial
                  selection screen.
 Fill in your USERID and PASSWORD and press ENTER
 (Your password will not appear when you type it)
 USERID   ===>
 PASSWORD ===>

 COMMAND  ===>
                                                          RUNNING   XXXXXXXX
================================================================================
Screen contains 30 fields.
Cursor is at [20,16].


Field summary:    Unp=Unprotected, Num=Numeric, Nds=Nondisplay, Mod=Modified

Fld#  Start     End        Len   Unp Num Nds Mod        First 30 Characters
-------------------------------------------------------------------------------
  1 | [1,2]    [1,47]      46  |   |   |   |   | [VIRTUAL MACHINE/SYSTEM PRODUCT
  2 | [1,49]   [2,17]      49  |   |   |   |   | [
  3 | [2,19]   [3,17]      79  |   | X |   |   | [
  4 | [3,19]   [4,17]      79  |   | X |   |   | [
  5 | [4,19]   [5,17]      79  |   | X |   |   | [        W E L C O M E  T O  T
  6 | [5,19]   [6,17]      79  |   | X |   |   | [ ========= ==========  ====
  7 | [6,19]   [7,17]      79  |   | X |   |   | [ ========= ============ =====
  8 | [7,19]   [8,17]      79  |   | X |   |   | [    ===     ====   ====  =====
  9 | [8,19]   [9,17]      79  |   | X |   |   | [    ===     ==========   =====
 10 | [9,19]   [10,17]     79  |   | X |   |   | [    ===     ==========   === =
 11 | [10,19]  [11,17]     79  |   | X |   |   | [    ===     ====   ====  ===
 12 | [11,19]  [12,17]     79  |   | X |   |   | [ ========= ============ ====
 13 | [12,19]  [13,17]     79  |   | X |   |   | [ ========= ==========  ====
 14 | [13,19]  [14,17]     79  |   | X |   |   | [M & D SE REGION I/S SUPPORT SE
 15 | [14,19]  [15,17]     79  |   | X |   |   | [This system is restricted to I
 16 | [15,19]  [16,17]     79  |   | X |   |   | [approved business purposes onl
 17 | [16,19]  [16,31]     13  |   | X |   |   | [ENTER: VMEXIT]
 18 | [16,33]  [17,17]     65  |   | X |   |   | [to return to the initial
 19 | [17,19]  [17,80]     62  |   | X |   |   | [selection screen.
 20 | [18,2]   [18,80]     79  |   | X |   |   | [Fill in your USERID and PASSWO
 21 | [19,2]   [19,80]     79  |   | X |   |   | [(Your password will not appear
 22 | [20,2]   [20,14]     13  |   | X |   |   | [USERID   ===>]
 23 | [20,16]  [20,23]      8  | X |   |   | X | [        ]
 24 | [20,25]  [20,80]     56  |   | X |   |   | [
 25 | [21,2]   [21,14]     13  |   | X |   |   | [PASSWORD ===>]
 26 | [21,16]  [21,55]     40  | X |   | X | X | [
 27 | [21,57]  [22,80]    104  |   | X |   |   | [
 28 | [23,2]   [23,14]     13  |   | X |   |   | [COMMAND  ===>]
 29 | [23,16]  [24,59]    124  | X |   |   | X | [
 30 | [24,61]  [24,79]     19  |   |   |   |   | [RUNNING   XXXXXXXX ]
```

Figure 1. PROFS Field Utility

Remember that the EASEL interface is built within and runs within the OS/2 interface in the programmable workstation. No changes to the host application are required.

An Example of an EASEL 3270 Interface

Figures 2a through 2f contain an EASEL program that provides a graphical CUA interface to the PROFS system running on a VM host. Its application requirements are simple:

- Show the PROFS screens in an OS/2 Presentation Manager window
- Use Communications Manager session B (defined as a Model 2 screen size – 24 x 80)
- Provide a means to:
  - Log on to the host – enter user ID and password
  - Log off from the host

Direct entry of data to PROFS is not required; this interface simply provides a means of logging on and off from PROFS.

To use a different session, copy the short name to the CMSession constant. Be sure that this session is at the initial PROFS logon screen. After compiling and running, the current screen image from the specified session is read and displayed.

To logon to the host, select the Start Host option from the action bar. A dialog box requesting the user ID will appear. Clicking on CANCEL will cancel the logon process. Typing in the ID and clicking on ENTER will cause the logon to be processed. The ID entered will be echoed on the host screen image for a second.

Next, a dialog box requesting the user password will appear. This box has a "protected" entry field so the password will not show. Type the password and select ENTER. A message will appear stating that the host connection is being established.

While waiting for the connection, PROFS can take a number of seconds to perform the initial DASD linking, message displaying, and so forth, before actually reaching the main PROFS menu. The program will allow 30 seconds for this activity to be performed. If the time limit expires, a "no-connection" message will be displayed. This sample code does not carry the error-checking any further; you could hit the refresh button to bring up the menu, or end the program, logoff, and try again.

To logoff, select the Logoff Host pulldown in the action bar when the PROFS main menu is displayed. The string "logoff" will be displayed and the PROFS logon screen should reappear in a few seconds.

The refresh button simply rereads the current PROFS screen into the textual region. The exit button in the upper right of the action bar ends the program.

The figure containing the program shows the code in the left column: the explanation of the code is in the right column. In a screen display of such a program, comments would be preceded by a pound sign (#). In this example, the static text in the dialog boxes may be shown as wrapped, but in an actual program should be entered as a single line.

Conclusion

The code in this EASEL PROFS interface is simple and is limited in function. There are many enhancements that could be added, for example:

- Pulldowns to send PF keys to the host
- Pulldowns to send other attention keys (for example, Enter, Clear, Reset)
- The ability the type directly into the textual region and send the data to the host
- Single functions that navigate through several host screens without additional user interaction (for example, retrieving mail or generating a list of all files in personal storage).

EASEL is a great tool for generating easy-to-use, interactive interfaces to optimize your host applications.

ABOUT THE AUTHOR

Clayton Arndt is an associate technical support representative in IBM's Application Solutions Product Support Operations supporting EASEL for OS/2 EE. He holds a B.S. in computer science engineering from the University of Texas at Arlington. Clayton was employed by IBM as a cooperative education student from 1986 to 1989 supporting the 3270 Information Display System. He received full employment upon graduation in May 1989.

```
screen size device units
module M3270
```

Specification for the EASEL 3270 module

```
string constant LogoffString     is "logoff"
string constant WaitingString    is "Waiting for
      host connection ..."
string constant GiveUpString     is  "Sorry ...
      host connection failed."
string constant CMSession        is  "B"
```

Declaration and initialization of string constants used in the EASEL interface. (WaitingString and GiveUpString are shown here as wrapped but should be entered as a single line.)

```
string variable UserID
string variable PasswordString
```

Declaration of variables used in the interface

```
action bar ProfsBar is
   pulldown StartHost  text "~Start Host"
      choice StartPROFS text "Start PROFS"
   end pulldown
   disabled pulldown LogoffHost text "~Logoff Host"
      choice LogoffPROFS text "Logoff PROFS"
   end pulldown
   button Refresh text "~Refresh"
   button Exit text "~Exit"
end action bar
```

Define the action bar to appear at the top of the interface window. Two pulldowns will appear:

pulldowns	choices
Start Host	Start PROFS
Logoff Host	Logoff PROFS

In addition, the action bar contains two buttons: Refresh (to re-read the current host screen into the text window) and Exit (to end the program).

```
primary textual region Profs
   size 580 280
   at position 30 50
   yellow foreground
   size border
   title bar "PROFS"
   system menu
   horizontal scroll bar
   vertical scroll bar
   action bar ProfsBar
   minimize button
   maximize button
   font "asc59"
```

Primary Region Definition:
The primary window is defined as a textual region with the title "PROFS". The window has a yellow foreground (the color in which the text will be displayed), and a black background (by default). The window also sports an OS/2 PM sizeable border, a system menu, and PM horizontal and vertical scroll bars. The previously defined action bar is included, as are minimize (default icon when minimized) and maximize buttons. The font "asc59" is an EASEL fixed-cell font with a total size of 7 x 11. This is the font used when displaying text within the window.

```
enabled invisible modal dialog box IDBox
   size 120 102
   at position 128 35
   dialog border
   title bar "Host User ID"
enabled visible static text IDText
   size 102 41
   at position 9 54
   in IDBox
   left align
   top align
   word wrap
   text "Please enter your host user ID.  Press
      ENTER to complete, or press CANCEL to quit."
disabled visible entry field IDField
   size 80 12
   at position 20 31
   in IDBox
   text size 8 columns
   left align
```

Dialog Box Object Definition(s):
The first dialog box is called IDBox. It has a dialog border and a title bar which displays "Host User ID". A static text field inside the dialog box provides instructions to the user to enter the ID into the entry field. (The static text in the dialog boxes may be shown here as wrapped, but should be entered as a single line.) An entry field called IDField is provided to accept the user's input; this field is limited to eight characters maximum. Two push buttons are also provided: one labeled OK (the default push button taken when the ENTER key is pressed) and one labeled Cancel (the cancel push button taken when the ESC key is pressed).

Figure 2a. Sample EASEL Program

```
enabled visible default push button OK
   size 38 12
   at position 11 3
   in IDBox
   text "~Enter"
enabled visible cancel push button Cancel
   size 38 12
   at position 60 3
   in IDBox
   text "~Cancel"

enabled invisible modal dialog box PasswordBox
   size 120 102
   at position 128 35
   dialog border
   title bar "Host Password"
enabled visible static text PasswordText
   size 102 41
   at position 9 54
   in PasswordBox
   left align
   top align
   word wrap
   text "Please enter your host password.  The
      entry field is protected so that the password
      will not show."
enabled invisible entry field PasswordField
   size 50 12
   at position 29 31
   in PasswordBox
   text size 8 columns
   left align
disabled visible entry field DummyField
   size 50 12
   at position 29 31
   in PasswordBox
   text size 30 columns
   left align
disabled visible group box ThinLine
   size 1 12
   at position 32 33
   in PasswordBox
enabled visible default push button OK
   size 38 12
   at position 11 3
   in PasswordBox
   text "~Enter"

action ShowProfsCursor is
   action GetCursorPosition
   add to Profs
      move to column Column line Row
   make Profs cursor visible



action ErrorMessage is
   clear Profs
   add to Profs
      move to column 20 line 10
      insert at cursor GiveUpString
```

The second dialog box is called PasswordBox. It also has a dialog border and a title bar which displays "Host Password". A static text field within the box instructs the user to enter the host password and reassures the user that the password will not be displayed. Two entry fields are defined in the dialog box: the real (active) entry field is called PasswordField. The other entry field is called DummyField and is positioned directly over the PasswordField. This has the effect of covering up PasswordField so that the password will not be displayed. To make it appear to the user that the top entry field is active, a very thin group box is positioned within DummyField to simulate the appearance of the text cursor. Finally, a default push button labeled OK is provided to allow the user to complete the password entry.

Action/Subroutine Definitions:

action ShowProfsCursor – Executes the 3270 action GetCursorPosition, which returns the current position of the host cursor in the global variables Column and Row. Within the text window Profs, the textual cursor is moved to these coordinates and is made visible.

action ErrorMessage – This action is called in response to a timeout on a host action. The textual region Profs is cleared and the message in GiveUpString is inserted into the window.

Figure 2b. Sample EASEL Program

```
subroutine GetUserID(string:String) is
   make IDBox visible
   activate IDField
   clear IDField
   begin guarded
      response to OK in IDBox
         copy (text of IDField) to String
         make IDBox invisible
         leave block
      response to Cancel in IDBox
         copy "" to String
         make IDBox invisible
         leave block
   end
```

subroutine GetUserID – This subroutine is called when the interface needs to request the user's ID for the host. The IDBox dialog box is made visible and the text cursor is moved to the IDField entry field. In addition, the IDField is cleared to remove any text remaining from a previous entry. A guarded block is set up to allow a response to be taken to the buttons in the dialog box. If the user clicks on the OK push button, the text typed into the IDField is copied to the subroutine parameter String (which is coincidentally a string variable), the IDBox is made invisible, and the guarded block is left. If the user clicks on the Cancel push button, a null string "" is copied to the parameter String, the IDBox is made invisible, and the guarded block is left.

```
subroutine GetPassword(string:String) is
   make PasswordBox visible
   activate PasswordField
   clear PasswordField
   begin guarded
      response to OK in PasswordBox
         copy (text of PasswordField) to String
         make PasswordBox invisible
         leave block
   end
```

subroutine GetPassword – This subroutine is called when the host is ready to receive the user's password. The PasswordBox dialog box is made visible on the screen, and the PasswordField entry field is activated and is cleared of any residual text. Remember from the definition of the controls in PasswordBox that the DummyField entry field overlays the PasswordField, so the user will not see the password as it is being entered. A guarded block is set up to allow a response to be taken to the push button in the dialog box. When the user has finished entering the password, clicking on the OK push button causes the text in PasswordField to be copied to the subroutine parameter String. The PasswordBox is then made invisible and the block is left.

```
response to start
   copy CMSession to SessionName
   action Init3270
   copy Profs to TextRegionName
   action ReadScreen
```

Response Definitions:
response to start – This is the initial response taken when the EASEL interface is first run. The short session name contained in the constant CMSession is copied to the global variable SessionName. The 3270 module is then started by calling Init3270. Profs is established to be the textual region into which the screens will be read, and the initial host screen is copied in. By default, the 3270 module is set up to use a Model 2 (24 x 80) host screen.

```
response to item StartPROFS in Profs
```

response to item StartPROFS in Profs – This rather lengthy response contains the logic for logging on to the host. Comments are embedded within the response to assist in explaining the logic of the code.

```
   disable item StartHost in Profs
   enable item LogoffHost in Profs
```

Change availability of items in the action bar.

```
   action ScanScreen
   action ShowProfsCursor
```

Read current host screen into textual region and show cursor.

```
   call GetUserID(UserID)
```

Call the subroutine GetUserID to query the user for the host ID. The result is returned in the string parameter UserID.

Figure 2c. Sample EASEL Program

`if (UserID != "") then`	If the UserID is not a null string (i.e. the user entered an ID and selected the OK push button), begin the logon.

`add to Profs`
`  insert at cursor UserID`
`wait 1`
	Display user ID in textual region and wait one second (so that the user can see the ID being entered).

`copy UserID to Keystrokes`	Copy the ID to the global variable Keystrokes in preparation for sending the ID to the host.

`action DefineWatch`
`copy 24 to WatchRow`
`copy 61 to WatchCol`
`copy "C" to WatchChar`
`action WatchForChar`
`copy 20 to SettleTime`
`action WatchForNoX`
`action EnterString`
`copy 3000 to GiveUpTime`
`action WatchAndWait`
	Set up a watch to indicate when we are at the next screen (this screen requests the user's password). The FIELDS utility was run previously against this next screen, and it was noticed that the string "CP READ" appeared in the field located at coordinates (24,61). The appearance of the character "C" at this location will be detected by the action WatchForChar and will serve to verify that the desired host screen is available. The watch will also wait for the "X" in the OIA to disappear. A characteristic of the host system is that the "X" may flicker while the screen is being updated; to prevent this flickering from giving the false indication that the host screen is ready, the value 20 (0.2 sec) is copied to the global variable SettleTime. The entire watch is limited to the time specified in GiveUpTime (in this case, 3000 = 30 sec). If the watch conditions are not satisfied before the GiveUpTime expires, the watch will be considered unsuccessful.

`if (WatchGaveUp) then`	Test to see if the watch was successful.
`  action ErrorMessage` `else`	Watch unsuccessful - display error message to user.

`  copy Profs to TextRegionName`
`  clear Profs`
`  action ScanScreen`
	Watch successful - clear the Profs textual region and scan the new host screen for its fields.

`  call GetPassword(PasswordString)`	Call the subroutine GetPassword to query the user for the host password. The password is returned in the subroutine parameter PasswordString.
`  copy PasswordString to Keystrokes`	Copy PasswordString to the global variable Keystrokes in preparation for sending the string to the host.

`  add to Profs`
`  move to column 20 line 10`
`  insert at cursor WaitingString`
	Display a message in the Profs textual region indicating that the interface is waiting on the host.

`  action DefineWatch`
`  copy 1 to WatchRow`
`  copy 78 to WatchCol`
`  copy "A" to WatchChar`
`  action WatchForChar`
`  action WatchForNoX`
`  action EnterString`
`  action WatchAndWait`
	Define a watch (similar to the previous watch) to wait while the host processes the password. By running the FIELDS utility against the next anticipated screen (the PROFS main menu), the character "A" located in the field at coordinates (1,78) was selected as the indication that the next host screen is ready. The SettleTime and GiveUpTime from the previous watch are still in effect.

Figure 2d. Sample EASEL Program

```
        if (WatchGaveUp) then
```

Test to see if watch was successful.

```
           action ErrorMessage
        else
```

Watch unsuccessful - display error message to user.

```
           copy Profs to TextRegionName
           clear Profs
           action ReadScreen
           action ScanScreen
           action ShowProfsCursor
        end if
     end if

  else
     enable item StartHost in Profs
     disable item LogoffHost in Profs
  end if
```

Watch successful - clear the Profs textual region and read in the current host screen. In addition, scan the host screen for its fields and show the text cursor in the textual region at the current cursor position.

This "else" is associated with the test to determine whether the user entered a user ID or cancelled the operation (which returned a null string). The availability of the pulldowns in the action bar is changed so that only the Start Host pulldown is active.

```
response to item LogoffPROFS in Profs
```

response to item LogoffPROFS in Profs – This response is taken when the user selects the Logoff Host pulldown from the action bar. It is assumed that the user is at the PROFS main menu (where the logoff command would normally be issued).

```
  action ScanScreen
  action ShowProfsCursor
```

Scan the host screen for its fields and activate the text cursor in the Profs textual region.

```
  disable item LogoffHost in Profs
  enable item StartHost in Profs
```

Change the availability of the pulldowns in the action bar.

```
  add to Profs
     insert at cursor LogoffString
  wait 1
```

For the benefit of the user, insert the contents of the string constant LogoffString into the textual region. Wait one (1) second so that the user can see the string.

```
  copy 50 to FieldNumber
  copy LogoffString to FieldText
  copy false to FillFlag
```

By running the FIELDS utility against the PROFS main menu, it was noted that the input field at the bottom of the screen is designated by the number 50. This is the field into which the string "logoff" (contained in the string constant LogoffString) must be typed. The FillFlag is set to false because it is not necessary to pad the rest of the field with blanks.

```
  action WriteField
```

The contents of the desired field are sent to the host.

```
  action DefineWatch
  copy 13 to WatchRow
  copy 19 to WatchCol
  copy "M"to WatchChar
  action WatchForChar
  action WatchForNoX
```

A host watch is defined to determine when the host has finished processing the logoff command. By running the FIELDS utility against the intitial host logon screen (see Figure 1), a certain characteristic of that screen could be chosen for the watch. Notice that field 14 at coordinates (13,19) begins with the characters "M & D". The character "M" at this location was chosen to indicate that this screen has been reached. The SettleTime and GiveUpTime from the previous watches are still in effect.

Figure 2e. Sample EASEL Program

```
   action PressENTER
   wait 2
   action PressENTER
   action WatchAndWait



   if (WatchGaveUp) then

      action ErrorMessage
   else

      copy Profs to TextRegionName
      clear Profs
      action ReadScreen
      action ScanScreen
      action ShowProfsCursor
   end if

response to item Refresh in Profs
   clear Profs
   copy Profs to TextRegionName
   action ReadScreen
   action ShowProfsCursor

response to item Exit in Profs
   copy CMSession to SessionName
   action Disconnect
   action Stop3270
   exit
```

One of the characteristics of this host system is that the ENTER key must be pressed again before the initial logon screen is displayed. Therefore, after entering the "logoff" string, this code will wait a couple of seconds and then press the ENTER key again before starting the watch for the next screen.

Test to see if watch was successful.

Watch unsuccessful - display error message to user.

Watch successful - copy the new host screen into the Profs textual region, and show the cursor at the current location. In addition, scan the new host screen to determine its fields for the next interaction.

response to item Refresh in Profs – This response is taken when the user clicks on the Refresh button in the action bar. It simply clears the Profs textual region and rereads the current host screen; nothing is sent to the host.

response to item Exit in Profs – This response to taken to the selection of the Exit button from the action bar. The 3270 module is stopped by calling the action Stop3270 and the interface is ended. Note that if the user selects Exit while the program is in the middle of the PROFS application, the EASEL interface will stop but PROFS will not. The session will be left alone.

Figure 2f. Sample EASEL Program

PS/2 RPG II Application Platform and Toolkit

Jim Weir
IBM Corporation
Boca Raton, Florida

The RPG II Platform supports multiple users on a PS/2 running the OS/2 operating system. Up to eight ASCII workstations are supported by the Platform. A ninth user is supported on the VGA-attached display on the PS/2 host system. The RPG II Platform also supports a total of 16 LAN-connected workstations. A combination of ASCII and LAN workstations are supported – up to a total of eight ASCII workstations and a system total of 16 workstations.

The RPG II Platform is a prerequisite for the IBM PS/2 RPG II Toolkit. The Toolkit is a separate product that allows RPG II application owners to migrate their System/36 RPG II applications to the PS/2 RPG II Platform. The RPG II Platform and Toolkit together make available on the PS/2 the thousands of System/36 RPG II applications available in the marketplace today.

Figure 1 shows the RPG Platform and the RPG II applications executing on the PS/2 host. The Toolkit consists of:

- RPG Compiler
- Screen Design Aid
- Screen Format Generator
- RPG II Auto Report
- Data File Utility List Creation
- Development Support Utility

OS/2 is the required operating system on the PS/2 host; OS/2 1.1 and 1.2 are supported by Release 1.1 of the Platform. Both OS/2 Standard Edition and OS/2 Extended Edition (EE) are supported. The RPG II Platform includes a terminal manager that provides the multiuser support. The terminal manager operates in conjunction with OS/2 multitasking capabilities and adds a record locking capability to ensure data integrity when files are shared among multiple users. The Platform uses standard OS/2 EE LAN software to support LAN workstations.

The RPG II execution environment supports a subset of System/36 OCL and SSP commands and includes several key System/36 utilities. All System/36 file types are supported, including indexed, alternate index, and direct files. Background processing is supported through a job queue facility.

In addition to the workstation support, two printers are supported (*note:* printers do *not* count as workstations) – one connected through

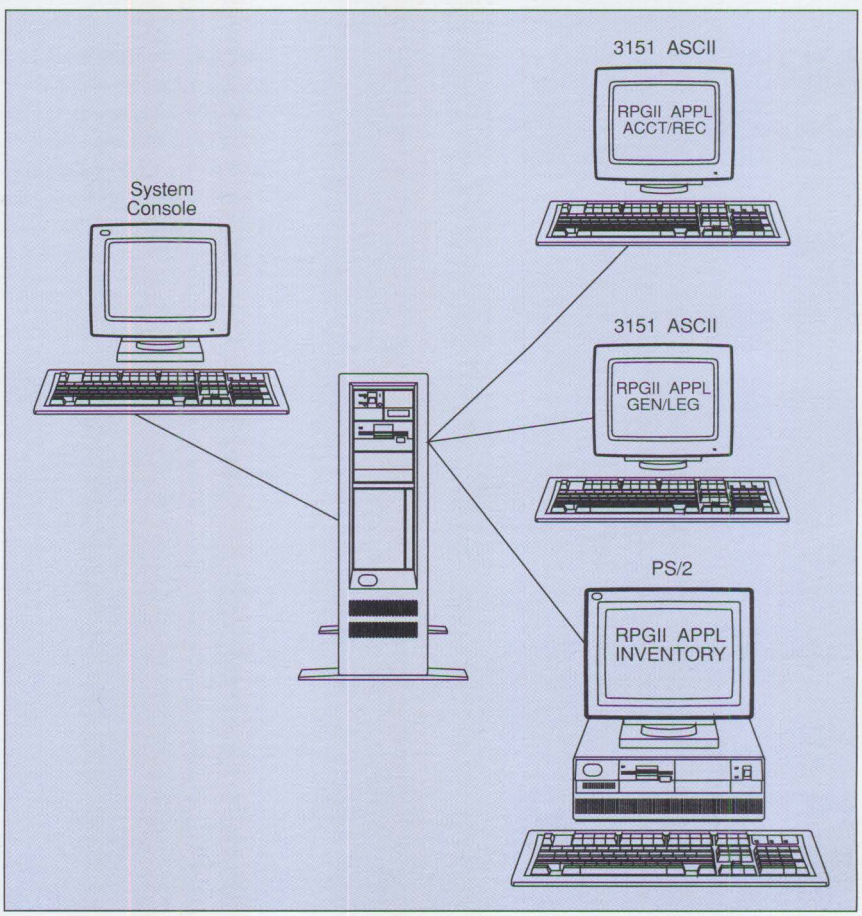


Figure 1. PS/2 RPG II Application Platform

the parallel port and one connected through the serial port. The printers supported are the Proprinter™, the Quickwriter®, and the 4224 model 3X X. The ASCII display stations supported are the IBM 3151 ASCII display, any PS/2, the PC AT 339 and the PC XT/286. All devices require the IBM Enhanced Keyboard, and any mix of these devices is supported (up to a total of eight).

In the LAN configuration, the server must use OS/2 EE, either 1.1 or 1.2. DOS and OS/2 workstations are supported on the LAN. These workstations can be supported with either the IBM PC Network or the IBM Token Ring.

Figure 2 shows how the RPG II Platform conceptually functions with OS/2. Standard LAN hardware and software are required to connect the LAN workstations to the OS/2 LAN Server. The RPG II Platform uses this standard LAN support to download RPG programs across the LAN and provide file server support on the OS/2 EE Server. The RPG programs execute on the LAN workstation. In the ASCII configuration, the terminal manager manages the attached workstations and provides the multiuser support on top of OS/2. Attached PS/2 or PC workstations operate in 3151 emulation mode provided by the Independent Workstation (IW) Program that is included with the RPG II Platform product. The 3151 Displays, Model 3X or 4X, require a feature called the Cartridge for Multiuser System Attachment. The 3151 Model 5X and 6X do not require this feature.

The execution environment and the terminal manager together require one OS/2 session. Each attached display station runs as a thread within the RPG terminal manager session. Figure 2 shows a total of four active RPG tasks running different RPG II applications. Other OS/2 sessions may also be active, such as a 3270 communications session (with OS/2 EE) as well as other personal computer applications. Figure 2 shows a PC Application running under the OS/2 operating system independent of the RPG II Platform.

The RPG II Platform is designed to utilize standard OS/2 interfaces and functions, such as print spooling, wherever possible. This means that compatibility with future releases of OS/2 is maintained. Attachments of the various devices supported by the Platform are shown in Figure 3. The Proprinter and Quickwriter can be attached to the parallel or the se-

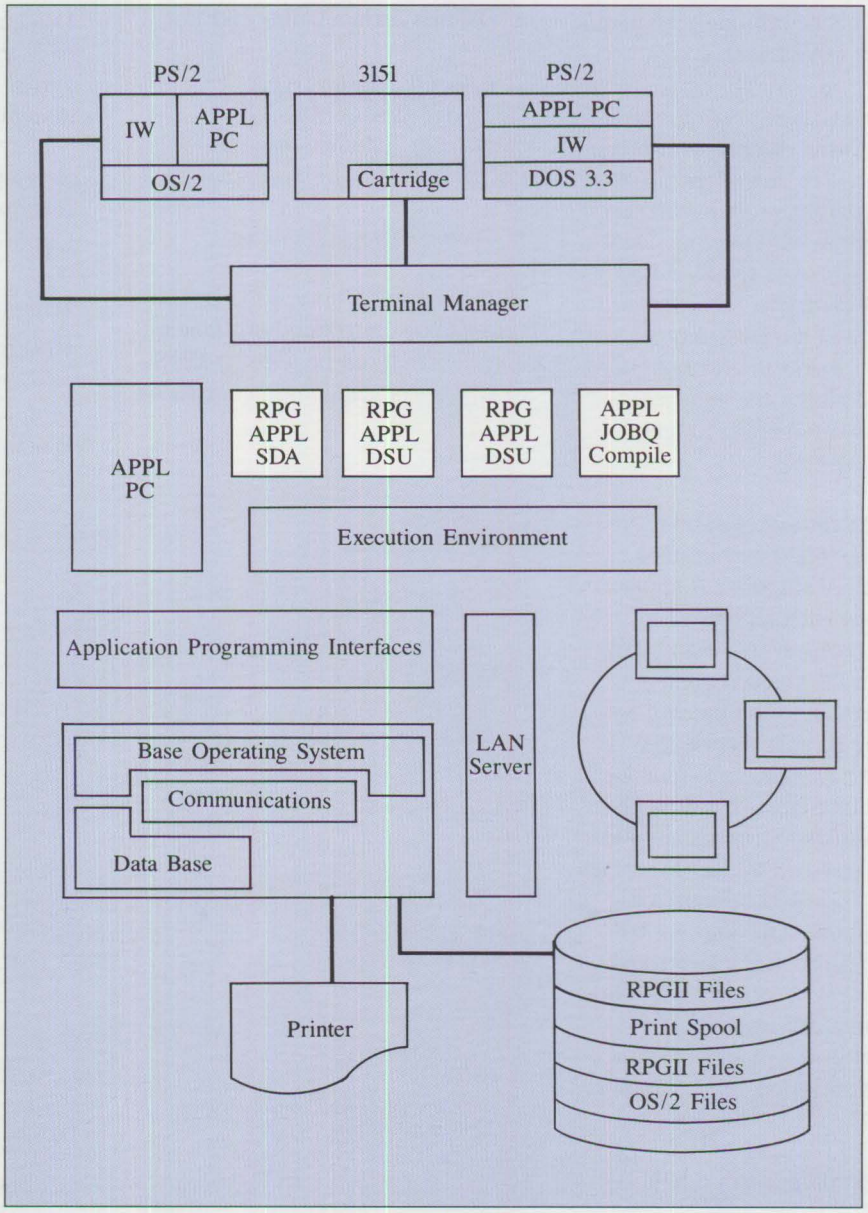


Figure 2. RPG II Platform and Toolkit with OS/2 EE

rial port. The 4224 printer is supported only through the PS/2 Serial Port. This standard serial port can be used to attach either a printer or one of the ASCII display stations.

A two-workstation configuration, therefore, can be supported without any additional adapters by using the system display as one workstation and the standard serial port for the second workstation. The Proprinter would be attached through the parallel port and serve as the system printer.

The Dual Async Adapter/A can support up to two of any of the display stations, providing a four-workstation configuration. Only one Dual Async Adapter/A is supported by the RPG II Platform. The standard Serial Port and the Dual Async Adapter/A support the RS232 interface, which allows the attached device to be located up to 50 feet from the host PS/2.

If more than four workstations are required, the Realtime Interface Coprocessor Multiport/2 Adapter must be used. It supports up to eight workstations. All eight workstations can be attached via the RS232 interface, or optionally, up to four may be attached via the RS232 interface and up to four via the RS422 interface. The RS422 attachment allows the workstation to be located up to 4,000 feet from the host PS/2. The RS232 interface adapter supports a distance of 40 feet. The Multiport/2 adapter and the Dual Async Adapter/A are mutually exclusive; for example, the RPG II Platform will support either one of these adapters but not both simultaneously.

The Multiport/2 adapter should be considered even in a two- to three-workstation configuration where growth is anticipated. It is required if the distance between the host PS/2 and the attached display station must be greater than 50 feet.

Utilities

RPG II Platform utilities are:

- Data File Conversion (DFC) Utility
- Data Exchange Utility (DEU)
- Data File Utility (DFU)
- Sort Utility
- Source Entry Utility (SEU)

With the DFC Utility, RPG II data files can be converted into standard text files that can be used by DOS or OS/2 personal computer applications, such as spreadsheets or word processors.

The DEU converts files from the EBCDIC file format of the System/36 to the ASCII file format of the PS/2, and vice versa. With this utility, files that have been downloaded from a System/36 can be converted and used by the PS/2 RPG II applications, and vice versa.

DFU, as on the System/36, is an easy-to-use tool for processing files and generating reports without doing any programming.

Sort and SEU provide the same functions as they do on the System/36.

Independent Workstation Program

Also included with the RPG II Platform package is the Independent Workstation Program, which pro-

Attachment Options	Devices Supported
PS/2 Parallel Port	Proprinter Quickwriter
PS/2 Serial Port RS232 Up to 50 feet	Printer 3151 ASCII Display PS/2 PC AT 339 PC/XT 286
Dual Async Adapter/A 2 Devices/1 Adapter Not Supported w/Multiport/2 RS232 Up to 50 feet	 3151 ASCII Display PS/2 PC AT 339 PC/XT 286
Realtime Interface Coprocessor Multiport/2 Adapter Up to 8 via RS232 Up to 4 each RS232 (50 feet) and RS 422	 3151 ASCII Display PC AT 339 PC/XT 286

Figure 3. RPG II Platform ASCII Attachment

vides the 3151 emulation capability and must be loaded on each of the attached PS/2 or PC workstations. In addition, the Independent Workstation Program enables "hotkey" capability to a stand-alone DOS or OS/2 application. This means that a user can access the RPG II applications on the host PS/2 and "hotkey" to a DOS or OS/2 application executing on the Independent Workstation; for example, running DisplayWrite for letter writing or other text work.

Because the RPG II applications are upwardly compatible to the S/36, a PS/2 system with the Platform and Toolkit software can serve as a stand-alone programmer development system for System/36 applications as well as for the PS/2. Programmers using the attached workstations can modify or create programs using the SEU or the DSU. RPG II programs can be entered, compiled and tested using the RPG II Platform and Toolkit. Once testing is satisfactory, RPG II programs can be converted from ASCII to EBCDIC using the DEU, then taken to a S/36 for final compilation and testing.

Host System	
Memory (+ OS/2 Requirements)	
Base System	400 KB
JOBQ	200 KB
System Console	200 KB
Attached Workstation	200 KB ea.
Maximum for 8 attached users =	2.5 MB
Fixed Disk	
RPG II Platform	6 MB
RPG II Toolkit	2 MB
RPG II Applications	?
Independent Workstation (IW)	
Memory	
IW Program	75 KB
Fixed Disk	
IW Program	50 KB

Figure 4. RPG II Platform/Toolkit Resource Requirements

Figure 2 shows a PS/2 used as a development system for RPG II applications. In this example, one workstation is running the Screen Design Aid (SDA), and two are using the DSU to create or modify RPG II source code. An RPG II compile has been initiated from the JOBQ, which executes as a separate OS/2 session. As mentioned previously, other OS/2 sessions may also be initiated for other personal computer applications.

Resource Requirements

Figure 4 lists the minimum memory required to run the RPG II Platform. The Platform requires a minimum of 1 MB of memory in addition to the memory required by the specific OS/2 edition installed. This 1 MB of memory supports the VGA-attached display and one attached display station. Because each attached display station requires 200 KB, 2.5 MB (in addition to the OS/2 requirement) is the maximum required to support eight workstations. For performance reasons these memory requirements allow the RPG II Platform and associated application code to be resident in memory so that no swapping is required.

In LAN configurations, the OS/2 EE Server should have a minimum of 8 MB of memory. OS/2 LAN requesters (which must also use OS/2 EE for the requester support) should have a minimum of 5 MB of memory. DOS workstations on the LAN should have a minimum of 640 KB.

The fixed-disk requirement for the RPG II Platform and Toolkit programs is approximately 6 MB. The total fixed-disk requirements will be determined by the specific applications and size of the customer's data files. The RPG II programs and data files take up approximately the same amount of space on the PS/2 as on the System/36.

The Independent Workstation (IW) Program, running on the attached PS/2 or PC workstations, requires 75 KB of memory and 50 KB of fixed-disk storage. For PS/2 diskette models, the IW program would be loaded at the beginning of the day, and then the diskette drive can be used for other PC applications as required.

Summary

The RPG II Platform adds an exciting capability for the Personal System/2. By expanding the application base to include many successful and proven RPG II business applications, IBM can offer a greater variety of PS/2 solutions.

By addressing the needs of small businesses, or small departments within larger businesses, the RPG II Platform and RPG II Toolkit offer new ways to meet customer needs.

ABOUT THE AUTHOR

Jim Weir is a senior systems analyst in IBM's Entry Systems Division laboratory in Boca Raton, Florida. He joined IBM in 1967 at the IBM Datacenter in New York City, then worked as a systems engineer in the Long Island branch office for 14 years. This is his third year working on the development of PS/2 RPG II Platform. He is currently pursuing a master's degree in computer science at Florida Atlantic University.

The IBM Independence Series Products

This is a description of the IBM products developed for people with disabilities.

In 1986, IBM Entry Systems Division in Boca Raton formed a new development group, Special Needs Systems. This group was chartered to provide not only products for people with disabilities, but also to provide direction to other IBM product development groups. This direction helps ensure that people with disabilities have equal access to IBM computers. The group's mission is "to enhance the employability of and the quality of life for people with disabilities through the use of IBM technology and products."

Special Needs Systems introduced Screen Reader Version 1.0 in January 1988. This was the group's first product and the beginning of the Independence Series of products.

Since then, two additional products were announced, as well as enhancements to Screen Reader. These products comprise the Independence Series today:

- Screen Reader™ – used by the blind and severely visually impaired to read screen information.
- SpeechViewer™ – used as a tool by speech therapists for the treatment of speech disorders.
- PhoneCommunicator™ – used by the deaf, severely hearing- and/or speech-impaired to communicate from an IBM Personal Computer to a Telecommunications Device for the Deaf (TDD), a Touch Tone™ telephone or another personal computer.

Screen Reader

IBM announced Screen Reader Version 1.1 in October 1989. This product uses a synthesized voice to read the text on the computer screen.

Using the separate 18-key keypad, the user sends commands to the Screen Reader so that characters, words, lines, sentences, or paragraphs are spoken. The entire screen can be read, or Screen Reader can automatically read as you browse through documents. Additionally, keystrokes can be read as they are typed.

Screen Reader contains the following features:

- Automatic monitoring and speaking of screen changes
- Adjustable reading boundaries on the screen
- Monitoring and speaking of all SAA menu selections
- Special profiles that customize Screen Reader for the following software applications:
  - IBM Assistant Series®
  - dBASE III Plus™
  - DisplayWrite 4™

SpeechViewer

  - Lotus 1-2-3™
  - WordPerfect™
  - 3270 and 3250 emulation programs
- Profile Access Language (PAL) for creation of custom profiles by customers
- A full-screen utility program for changing the preset synthesizer values and messages spoken.

Screen Reader Development: Screen Reader represents a significant technology transfer from researcher to end-user product. The original concept and software development for Screen Reader was done by research scientists at IBM's Thomas J. Watson Research Center. The technology and usability were enhanced by Special Needs Systems in conjunction with research and many IBM users who are blind.

During Screen Reader development, some work was done in conjunction with several vendors to provide a complete workstation for blind or visually impaired people. For the first time on a single IBM computer system, a user can scan in printed documents and read them with voice or screen enlargement.

Two vendors, Arkenstone and Kurzweil, have products that can scan hardcopy material, such as letters, book pages, and newsprint into the computer. Screen Reader can then be used to read the scanned information.

Vista™ and LP-DOS™ are two programs that can be used to enlarge screen text. Users with partial sight then may be able to read the screen information. Characters can be enlarged in synchronization with the reading of screen information using Screen Reader.

Documentation: The documentation for Screen Reader is available in four media:

- Audiocassette
- Online
- Print
- Braille

The self-paced, online tutorial makes learning the basics easy. Reference material is presented in a clear and easy-to-read form.

Operating Requirements: IBM Screen Reader works on the PC XT, PC XT 286, Personal Computer AT, and all PS/2 models.

Additionally, the program requires IBM DOS version 3.30 or later, 128 KB system memory, and sufficient storage for applications and DOS.

Compatibility: Screen Reader supports most commercially available text-to-speech synthesizers, including the following:

- Accent™
- Apollo™
- Audapter™
- Calltext 5050 5010™
- DECtalk™
- Echo PC™
- Personal Speech System™
- Prose 2020™
- TYPE 'N TALK™
- VoxBox™

Support for internal synthesizers may be available through some synthesizer manufacturers.

Screen Reader can use expanded memory if it is available on the computer.

Ordering Information: The IBM Screen Reader option for the PS/2 computers (order number 6450616) includes:

- 18-key keypad
- Keypad cable
- Program diskettes
- Online documentation
- Audiocassettes
- Printed book

The IBM Screen Reader option for the IBM Personal Computers (6450617) includes all of the preceding plus an adapter for IBM personal computers.

The IBM Screen Reader Braille Book (6024937) includes Braille documentation of the User's Guide and a reference book.

SpeechViewer

Special Needs Systems announced SpeechViewer, the second member of the Independence Series, in November 1988. SpeechViewer is a clinical tool for speech pathologists, teachers, and other professionals trained in the treatment of speech, language, and hearing disorders.

Using SpeechViewer, speech professionals can help their clients monitor and gain control over such speech attributes as voicing, pitch, loudness, vowel pronunciation accuracy, and speech timing.

SpeechViewer hardware and software addresses a key element in speech therapy: the feedback process. SpeechViewer complements traditional speech therapy procedures by providing visual and auditory feedback on selected speech attributes. SpeechViewer uses a microphone to accept a user's speech sounds. The sounds are digitized, stored, and instantly analyzed while providing immediate feedback to the user.

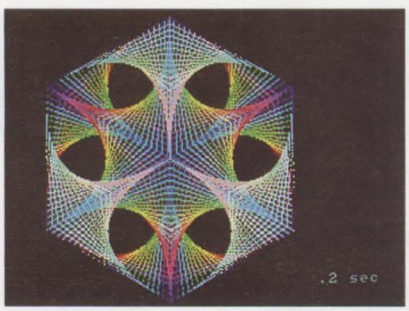
SpeechViewer software consists of 13 modules accessible from a main menu. The diversity of the modules makes SpeechViewer appropriate for all age ranges and for many communication disorders. SpeechViewer is easy to use, yet flexible. The speech professional can customize the modules to meet individual client needs.

The SpeechViewer modules are classified by their clinical objectives. The three classifications are:

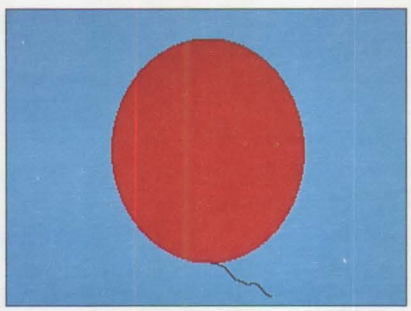
- Awareness – The Awareness modules use simple displays and employ a cause-and-effect methodology to focus attention on a discrete speech attribute. These modules can hold a client's attention and provide positive feedback about selected attributes of speech. For example, with the Pitch Awareness module, clients can see the mercury in a thermometer rise and fall as their pitch rises and falls.

- Skill Building – The Skill Building modules use game-like displays and employ a goal-oriented methodology to help the client gain control over speech attributes such as pitch, voicing, and vowel pronunciation accuracy. For example, a client can guide a mobile through a maze by accurately pronouncing four different vowel sounds.

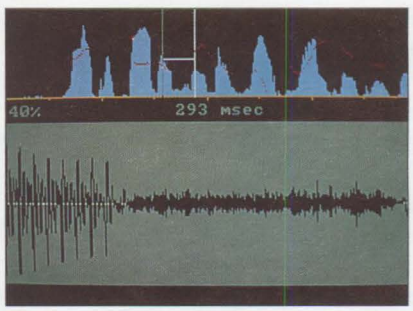
- Patterning – The Patterning modules use technical displays for creating and matching visual representations of speech attri-



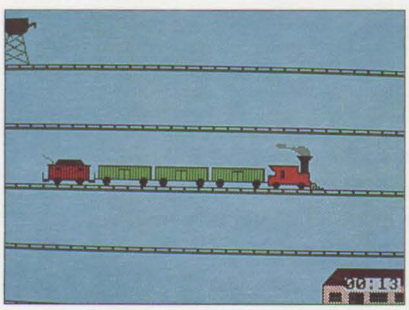
Sound



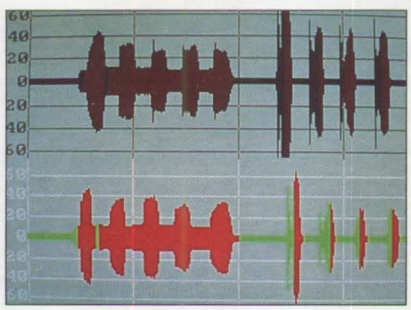
Loudness



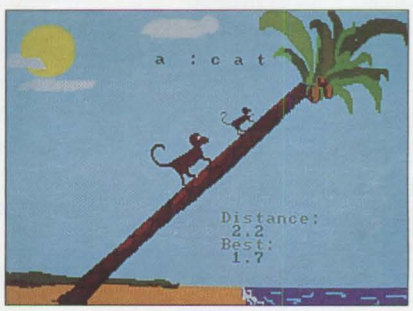
Waveform



Voicing Onset



Pitch and Loudness



Vowel Accuracy

butes. These modules contain technical, quantifiable information for critical analysis.

SpeechViewer Development: SpeechViewer is based on more than 10 years of research and clinical experience. In 1977, the IBM Paris Scientific Center organized a research team to create a speech training tool for deaf children. The first prototype, which used digital signal processing technology to analyze a child's speech, was installed in a school for the deaf in Paris in 1979. Eventually, prototypes were installed in more than 150 locations in 29 countries. As technology advanced and input was received from teachers and clinicians using the prototype, the researchers at the Paris Scientific Center continued to make improvements to the speech training tool.

In 1986, an effort was begun to gather data on the clinical effectiveness of the speech training prototype, not only with deaf children, but with people of all ages and with many types of communication disorders. Testing in hospitals, schools, universities, and schools for the deaf demonstrated that the programs were highly motivating, provided critical visual feedback to therapists and clients, and could be integrated into established and accepted clinical practices. The positive results provided the impetus for Special Needs Systems to transfer the technology from the research stage into a product.

Operating Requirements: SpeechViewer is designed to run on an IBM Personal System/2 Model 25, 30, and 30-286 (8525 or 8530) with the following:

- IBM DOS 4.00
- 512KB of system memory
- One available full-size expansion slot for the adapter
- Graphics printer (optional)

Compatibility: The SpeechViewer program requires exclusive use of the system hardware while running.

Ordering Information: The IBM Personal System/2 SpeechViewer Convenience Kit (6450610) includes the SpeechViewer Hardware Option and SpeechViewer Application Software.

The IBM Personal System/2 SpeechViewer Hardware Option (6450611) includes:

- SpeechViewer adapter
- One 3.5-inch diagnostic diskette
- Hardware Option Installation Guide
- OEM microphone and cable
- OEM speaker and cable
- Test plug

The IBM Personal System/2 SpeechViewer Software Option (6280310) includes:

- User's Guide
- Quick Reference
- Two 3.5-inch application software diskettes

PhoneCommunicator

The PhoneCommunicator program is the newest member of the IBM Independence Series. This program represents a major step forward in providing communication assistance to hearing- or speech-impaired people. These people now have a product to aid them in communicating with the hearing world through the use of an IBM computer and a standard touch-tone telephone. It also communicates with TDDs, providing the user a full-screen view of the dialogue of both parties. Other computer applications may be running while PhoneCommunicator is active.

PhoneCommunicator Development: This product had its genesis from the Augmented Phone Services program, which was introduced in 1985. Although based on the 1985 software, the PhoneCommunicator program has undergone major enhancements. Feedback from IBM internal users inspired two of the original developers to continue enhancing the product to satisfy user requirements. Their efforts resulted in an IBM internally released version of the product.

In 1988, an effort was started by the Special Needs Systems group in Boca Raton to understand the effectiveness of the IBM internal product and assess whether it could be enhanced to meet the needs of any hearing- and/or speech-impaired person. As the development team worked, testing began inside IBM with hearing- and speech-impaired employees, and outside IBM at universities, deaf service centers, and government agencies. The development team worked diligently with all the test groups to understand and incorporate many of their requests into the final product. IBM announced PhoneCommunicator in December 1989.

Product Description: The PhoneCommunicator adapter, which uses the personal computer bus ar-

chitecture, provides a synthesized voice output to the telephone. The program can also provide a modem connection to the telephone line for communication with a TDD (Baudot or ACSII), another computer using the PhoneCommunicator program, or an ASCII bulletin board service.

When the user receives a telephone call, the screen flashes, visually indicating an incoming call. The user can then press a sequence of keys to suspend the current application and start the PhoneCommunicator program.

The hearing- and/or speech-impaired person can read conversational messages on the screen. Those messages can be saved and printed.

PhoneCommunicator contains the necessary facilities to access and communicate with bulletin board services in conversational mode. The modem on the PhoneCommunicator Adapter can be used to transfer ASCII data files to and from a host computer. This requires Crosstalk™ XVI, Version 3.61A, a separately purchased software package.

The program can also be like a telephone answering machine. This allows the user to have incoming calls answered by the computer. Telephone numbers and messages can be left by callers.

A base vocabulary of 8,500 words is standard. These words are used to decode the numbers sent from a touch-tone telephone into words that can be viewed by the user on the computer screen. The user can append words to the vocabulary.

Operating Requirements: The PhoneCommunicator Hardware Adapter is a personal computer bus adapter and uses a full-length slot.

PhoneCommunicator runs on the following IBM computers:

- Personal Computer
- Personal Computer XT, except XT/370
- Personal Computer XT-286
- Personal Computer AT, except AT/370
- Personal System/2 Models 25, 30, and 30-286

The computers must have IBM DOS version 3.30 or 4.00, 512 KB of system memory and one fixed-disk drive or two diskette drives.

Compatibility: With 640 KB of memory, programs up to 256 KB may be run while PhoneCommunicator is active.

With the separate purchase of SoftwareCarousel™ version 3.0, applications up to 512 KB can co-reside with PhoneCommunicator in a 640 KB system. SoftwareCarousel allows programs to be temporarily suspended while a telephone call is being made or received. No data is lost from the program you are working in when you suspend the program to make or answer a telephone call.

Ordering Information: The IBM PhoneCommunicator Convenience Kit (6450618) includes the PhoneCommunicator Hardware and Software Options.

The IBM PhoneCommunicator Hardware Option (6450619) includes:

- PhoneCommunicator Adapter
- Hardware installation instructions
- A 5.25-inch and a 3.5-inch diagnostic diskette

The IBM PhoneCommunicator Application Software (57F1954) includes:

- PhoneCommunicator Application diskettes in both 5.25- and 3.5-inch media
- PhoneCommunicator User's Guide
- Telephone reference cards (10 per pack)

Product Support

The IBM National Support Center for Persons with Disabilities (NSCPD) provides technical and marketing support for these products. The NSCPD also acts as a clearing house for information on other products and agencies that assist people with disabilities. The NSCPD can be contacted at 1 800 426-2133; or if using a TDD, at 1 800 284-9482.

The Independence Series products are available only through IBM TeleMarketing Operations. Product orders can be placed by calling 1 800 426-3388 (voice calls) or 1 800 426-3383 (TDD calls). If calling from Canada, dial 1 800 465-1234.

New Products

Hardware

IBM PS/2 Model 70 486 (8570-B61 and 8570-B21)

The Personal System/2 Model 70 486 (8570-B61 and 8570-B21), with a 25 MHz i486™ 32-bit microprocessor, enhanced the PS/2 family of systems by offering a new level of advanced performance in a desktop unit. The Model 70 486 includes a 25 Mhz i486 32-bit microprocessor that features an internal memory cache controller, an internal 8 KB memory cache, and an internal floating-point processing unit to perform the functions of external 80387 Math Co-Processor.

The Model 70 386 (8570-A61 and 8570-A21) can be upgraded to the Model 70 486 by the installation of the 486/25 Power Platform.

The Model 70 486 is appropriate for advanced users who perform numeric-intensive applications requiring floating-point operations, or large memory resident applications that will benefit from the advanced processing capabilities of the 25 Mhz i486 32-bit microprocessor.

The Model 70 486 is compatible with most existing software products for IBM personal computer and Personal System/2 systems. Due to the higher speed of the i486 processor in the Model 70 486, some timing sensitive software may require modification.

The Model 70 486 is supported by Operating System/2 (OS/2) Standard Edition 1.1 and 1.2, OS/2 Extended Edition 1.1 and 1.2, and the Disk Operating System (DOS) Version 3.3 and 4.0. Corrective Service Diskette UR 25066 is required to support Expanded Memory Specification (EMS) with DOS 4.0.

Highlights:

- 25 MHz i486 32-bit microprocessor
- Internal memory cache controller and 8 KB internal memory cache
- Internal floating-point unit replaces need for external 80387
- System board memory expandable to 8 MB
- 60 MB or 120 MB fixed disk with integrated controller
- Micro Channel™ Architecture with one 16-bit and two 32-bit slots
- Well-suited for applications such as engineering/scientific, image processing, financial modeling, software development, and other numeric-intensive applications
- Software compatibility with 80386 systems
- Upgrade capability from Model 70 386 (8570-A61/A21).

IBM Token-Ring Network PS/2 Model P70 386 Adapter/A

The IBM Token-Ring Network PS/2 Model P70 386 Adapter/A (feature code #1598) permits attachment of the IBM PS/2 Model P70 386 (8573-061 and 8573-121) to the IBM Token-Ring Network. This adapter is half-length and designed to fit in the short slot. It transmits and receives at 4 million bits per second using protocols conforming to IEEE 802.5 and ECMA 89 standards. The adapter provides logical link control functions conforming with the IEEE 802.2 standard and is compatible with International Standard ISO – 8802/5. It provides 16 KB of RAM and all of the function of the IBM Token-Ring Network Adapter/A (feature code #4790). To facilitate attachment to the IBM PS/2 P70 386, a unique L angle connector with a permanently attached short cable is provided.

Highlights:

- Provides 16 KB of RAM
- Designed for use in half-length slot
- Compatible with International Standard ISO-8802/5
- Unique L angle connector with permanently attached short cable

M-Motion Video Adapter/A For The IBM Personal System/2

The M-Motion Video Adapter/A is an Personal System/2 Micro Channel adapter card that allows Micro Channel PS/2s to display full motion, interactive color video on standard color PS/2 monitors. This adapter card also provides full-line audio and limited digital audio capabilities. The M-Motion Video Adapter/A receives analog signals from an audio-video source, processes the signals, and sends them to a PS/2 monitor and speaker. The M-Motion Video Adapter/A is designed to support all existing InfoWindow™ applications.

Highlights:

- Produces interactive full motion color video and audio on PS/2 Micro Channel systems
- Produces windowed video on color PS/2 monitors
- Allows two high-quality stereo inputs as well as three NTSC/PAL video inputs or two Super VHS video inputs
- Allows recording and playback of narrative quality audio to and from a DASD device
- Supports all VGA modes (640 x 480 pixels)
- Includes multiple connector cable allowing input from audio and video devices provided
- Allows computer control of video and audio functions (for example, source selection, volume, tone, windowing, color, brightness, hue, saturation contrast, etc.)

IBM Store Controller Kit for IBM PS/2 Model 55 SX

The IBM Store Controller Kit allows the IBM PS/2 Model 55 SX (8555-031 or 8555-061) with the IBM 4680 Operating System Version 2 (5601-192) to become a store controller for the IBM 4680 Store System. The Second Store Loop Adapter/A option monitors another store loop's activity and, with the 4680 Operation System, takes control of the monitored loop if it detects a lack of store controller activity.

Highlights:

- Lower entry price store controller
- Internal speed and memory capacities of PS/2
- User can upgrade installed PS/2 machines
- Token-Ring Networking to service multicontroller – PS/2 and PC AT® store controllers with the same Network Master
- Support of selected PS/2 options

Software

NetView/PC Version Available with Additional Function

NetView ™/PC Version 1.2.1 (English version) provides additional function and will be available on March 30, 1990, to coincide with Operating System/2 Extended Edition (OS/2) EE Version 1.2 availability.

NetView/PC is an extension of the Communication Network Management (CNM) services provided by NetView. It is enhanced to installing Network/PC Gateway function and the option of selectively installing Network/PC the NetView/PC Gateway function, or both. The Remote Console Facility may be also be installed with the functions.

With the NetView/PC Gateway function, those customers who require a screen-less NetView/PC may install only the NetView/PC application programming interface/communication system (API/CS) function. It allows vendor applications to provide a single user interface, and continues to provide base gateway services to NetView for IBM and vendor applications managing non-SNA equipment.

Highlights:

- Screenless NetView/PC Gateway:
  - — Provides an API/CS for writing applications
  - — Allows vendor applications to provide a single user interface
  - — Allows operation in an unattended environment
  - — Provides optional alert logging capability
  - — Requires fewer system resources
- Installation Options — The user may install NetView/PC, the NetView /PC Gateway function, or both, with or without Remote Console Facility.

IBM DataInterchange/2 Availability

IBM DataInterchange/2 provides translation and business document management facilities for Electronic Data Interchange (EDI) in Systems Application Architecture™ (SAA ™) operating system environments. The translation facility includes a utility for converting application data formats into or from U.S. EDI standards (ANSI X12) and international EDI standards (EDIFACT and UN/TDI). Standard updates are provided and maintained as a part of the product.

DataInternational/2 can be tailored for different application data formats, standard transactions, trading partners, and network parameters via online transactions, trading partners, and network parameters via online customization. It can be integrated with existing or new application or can be operated as a stand-alone application.

DataInterchange/2 provides the capability to read application data from a file or write to a file, that can be uploaded or downloaded to other systems. The product also includes facilities for logging, error recovery and security.

M-Control Program

The IBM M-Control Program includes program interfaces for DOS and an OS/2 Presentation Manager toolkit. The M-Control Program provides functional support for application developers who want to use PS/2 model configurations containing the following multimedia features:

- Models 50, 50Z, 55SX, 60, 70, and 80
  - — M-Video Adapter/A for the IBM PS/2
- Supported models of videodisc players as audio and/or video storage devices

The DOS program interfaces support the InfoWindow™ Touch Display Control Program function that applies to the M-Motion Video Adapter/A for the PS/2.

The OS/2 Toolkit Dynamic Link Libraries (DDL) provide OS/2 Presentation Manager objects and messaging interfaces for video control. The toolkit also includes a sample multimedia application, with source code, demonstrating use of toolkit functions.

Highlights:

- Allows programmers the ability to control multimedia devices such as video and audio
- Allows programmers the ability to create pointer device drivers that meet PS/2 pointer interface requirements such as mouse, touchscreen or other

- Supports the M-Motion Video Adapter/A for the PS/2 that provides scalable full-motion video with graphic overlay
- Supports multiple RS-232C serial interfaced videodisc players
- Supports both NTSC and PAL video standards
- Supports the part of the InfoWindow Touch Display Control Program function that applies to the M-Motion Video Adapter/A for the PS/2

IBM LinkWay Multimedia Tools

IBM LinkWay Multimedia Tools provide LinkWay application developers functional support for the following multimedia features:

- Personal System/2 Audio Capture/Playback Adapter/A
- PS/2 Audio Capture/Playback Adapter
- M-Motion Video Adapter/A for the IBM Personal System/2
- Supported Videodisc Player models for video display

The LinkWay Multimedia Tools must be used with IBM LinkWay.

Highlights:

- Allows developers the ability to control multimedia devices such as video and audio
- Expands the ability of developers to create multimedia applications using IBM LinkWay
- Supports the M-Motion Video Adapter/A for the PS/2 and the PS/2 Audio Capture/Playback Adapter/A that provide high-quality audio and scalable full-motion video with graphic overlay
- Multiple RS-232C serial interfaced videodisc players supported

IBM Storyboard Plus Version 2.00 LAN Pack

The Storyboard™ Plus Version 2.00 LAN Pack Licensed program is a multimedia application offering for qualified educational institutions. It combines drawing, painting, text, still audiovisual presentations on an IBM Personal Computer, certified IBM compatible computer, or Personal System/2 (PS/2) system.

The Storyboard Plus Version 2.00 LAN Pack may be used under control of a local area network program, concurrently on up to 50 machines including one server machine that controls the network.

Highlights:

- Functions to enhance picture and story editing capabilities
- Support for a number of video capture interface cards
- Support for additional printers, including additional color printers
- Support for the IBM 8514/A Video Adapter in 640 x 480 – 256 color mode
- Ability to mix display resolutions within a presentation

Software Carousel Version 3.01 Lotus 1-2-3 Release 2.2

Software Carousel® 3.01 provides memory relief to permit large DOS applications to run in the DOS task area when operating with Workstation Connectivity Memory Management Enhancement program offering 83X9133. The functions provided by Software Carousel and Workstation Connectivity Memory Management Enhancement allow a user to free up to approximately 400-500 K of memory for application use. It is recommended that the programming announcement for Workstation Connectivity Memory Management Enhancement be reviewed to ensure that it is appropriate for your environment.

Software Carousel 3.01 used in conjunction with the IBM Workstation Connectivity Memory Management Enhancement program supports the following IBM Program/Product Announcement offerings:

- IBM Personal Communications/3270
- PC/HOST File Transfer and Terminal Emulator Program Version 2.1
- IBM Personal System/2 Expanded Memory Options
- Application System/400™ (AS/400™) PC Support Release 2

Software Carousel allows you to load up to 12 programs, and work with all 12 at once. In a local area network (LAN) environment, a workstation user can have a host connectivity program, a spreadsheet program, a word processing program, a data base program and others in different partitions, only one of which will be active at a given time. Programs are stored in expanded or extended Random Access Memory (RAM), or in a disk file while inactive. All of lower RAM is available to each program you run. A menu-driven, fill-in-the-blanks, configuration program makes automating your system quick and easy.

Highlights:

- Compatible with virtually every PC programs.
- Solves conflicts with using many RAM-resident programs.
- Makes full use of expanded and extended memory for lightning-fast switching between programs.
- Completely user-configurable so command keys never conflict with other programs like WordPerfect or Sidekick.

Lotus 1-2-3 Release 2.2 (3.5-inch and 5.25-inch)

- The add-in Manager has been built into Release 2.2 as a main-menu item making it easy to use existing 1-2-3 add-ins. The user can now hook or un-hook add-in functionally as needed.

- The Lotus Learn add-in has been built into Release 2.2 proving a method of recording keystrokes for easy macro creation.

- The Macro Library Manager add-in allows for safe, off-sheet storage of macros. The user can run macros from the Macro Library Manager in the current file. The Library can contain macros, formulas, ranges or entire applications.

- Enhanced STEP mode displays macro commands at the bottom of the screen as the user "steps" through the execution of a macro. Allows the user to test automated routines and quickly debug macros.

- Undo is an error correction mechanism that allows users to reverse changes made to the worksheet since 1-2-3 was last in ready mode.

- Search and Replace allows users to find and replace any string that is located within a formula and/or label entry.

- Spreadsheet publishing with Allways has been incorporated into Release 2.2. Allways is an add-ins that allows users to prepare typeset quality output with mixed text and graphics directly from within the spreadsheet.

MERVA/2 Version 2

The IBM Message Entry and Routing with Interfaces to Various Applications (MERVA)/2 licensed program replaces MERVA/2 Version 1 including the Multi MERVA workstation feature and the associated product Direct S.W.I.F.T. Network Link (DSNL)/2 Version 1. MERVA/2 Version 2 integrates all the functions necessary to support connections to the SWIFT I and SWIFT II networks [the worldwide private networks defined and maintained by the Society for Worldwide Interbank Financial Telecommunication (SWIFT)], as well as to Telex networks and other MERVA installations (on PS/2 or System/370). MERVA/2 Version 2 operates under OS/2 Extended Edition 1.2.

To allow consideration of test results with the new SWIFT II network, general availability of MERVA/2 Version 2 will be announced in April 1990.

Highlights:

- Integrated solution covering the following functions:
  - — SWIFT I and SWIFT II network support
  - — Telex network support via the PS/2
  - — MERVA to MERVA connection between PS/2 and PS/2 as well as between PS/2 and System/370 MVS-CICS and VSE-CICS

- Support of new and changed message types, to be implemented by S.W.I.F.T. until general availability of MERVA/2 Version 2

IBM Plant Floor Series Plantworks: Application Automation Edition

The IBM Plant Floor Series™ PlantWorks™: Application Automation Edition™ consists of the Execution Services/2, Build Services/2, Definition Services/2, and Interfaces Services/2 licensed programs. PlantWorks provides a distributed, plant-floor supervisory facilities for monitoring and controlling plant floor equipment. It includes an application creation capability for the non-programmer and a set of application execution functions. PlantWorks executes on specified IBM Operating System/2 (OS/2) Extended Edition and Plant Floor Series Distributed Automation Edition.

Plant Floor Series Distributed Automation Edition™ Entry Communications System/2

Plant Floor Series Distributed Automation Edition is a set of licensed programs that provides a standard application interface to plant floor devices and other communication facilities.

Entry Communications System/2, which addresses workstation requirements, is added to the Distributed Automation Edition. It allows the user to develop manufacturing or process control applications on IBM industrial computers, IBM personal computers, or IBM Personal System/2, using the Operating System/2 (OS/2 Extended Edition (EE)). Entry Communications System/2 may be upgraded to Communications System/2.

Highlights:

- Communications between programs anywhere in a network of IBM industrial computers, IBM personal computers, Personal System/2 computers, and ES/9370

- Communication and management capabilities to support plant floor devices, such a robots, bar code readers, programmable controllers, and plant floor data collection terminals, using the IBM Real Time Interface Co-processor resident in a remote Communications System/2 node

- Routing and management of plant floor data and programs

- Resource management through logical views rather than through complex physical characteristics

- Application program interfaces (API) for use of high-level program languages

- System services for print spooling, logging, network/node initialization, and shutdown

- Interoperability support between VM and OS/2 EE platforms
- Future Manufacturing Automation Protocol (MAP)(1) and OSI Manufacturing Messaging Services (MMS) support
- Relationship to the IBM Computer Integrated Manufacturing (CIM) Architecture and Systems Application Architecture™SAA™

(1) MAP is an acronym of the General Motors Corporation to identify its Manufacturing Automation Protocol.

SolutionPac Writing to Read Center V2

The SolutionPac™ Writing to Read® Center Version 2 allows more flexibility for customers to customize their labs based on the number of students, from as few as eight students to as many as 40 students per class, in increments of eight-student blocks. The new configuration options of the SolutionPac Writing to Read Center Version 2 make it possible to reach schools and students not previously accommodated by the initially announced configuration.

Highlights:

- A Combination of the Personal System/2 (PS/2) Model 25 computers, Proprinter® II, Listening Library books, cassette players and headsets, and Writing to Read and other software
- Additional copies of Primary Editor Plus (PEP) program diskettes and formatted story diskettes are included
- On request, Installation/Training Option Services are available at no charge from an IBM Authorized Education Specialist (for the PS/2 equipment only)
- Customer support for the application software is provided through a toll-free number.

Exploring Measurement, Time, & Money - Level II Version 1.00

Exploring Measurement, Time, and Money – Level II Version 1.00 is designed for students working at the third- and fourth-grade level. This program provides exploratory, structured, and construction activities for the content areas of measurement, time, and money.

Exploring Measurement, Time, & Money - Level III Version 1.00

Exploring Measurement, Time, and Money – Level III Version 1.00 is designed for students working at the fifth- and sixth-grade level. This program provides exploratory, structured, and construction activities for the content areas of measurement, time, and money.

IBM Interleaf™ Publisher Version 1 Release 1

IBM Interleaf Publisher Version 1 Release 1 contains the following additional features not contained in the previous version:

- Novell Netware™ Version 2.12 support
- Math Equation formatting and editing
- Continuous tone image editing (gray scale)
- Reduced disk storage requirements
- Enhanced import capabilities (TIFF, RFT/DCA, HPGL, EPS)
- IBM Personal Page Printer II Sheetfeed option support

IBM Standard Generalized Markup TextWrite Operating System/2 Edition Limited Availability

Standard Generalized Markup Language (SGML) TextWrite Operating System/2 (OS/2) Edition will be available February 23, 1990, for OS/2 Standard Edition 1.2 users, on a limited basis.

General availability for SGML TextWrite on both OS/2 Standard Edition Version 1.2 and OS/2 Extended Edition Version 1.2 and OS/2 Extended Edition Version 1.2 will be April 27, 1990.

The following is more specific information about the validation function provided in SGML TextWrite OS/2 Edition:

- SGML TextWrite highlights any tags that are entered out of a valid context, as specified by the Document Type Definition (DTD).
- The user can also invoke a background process in the editor to validate the entire file for full compliance with the DTD according to International Standard (ISO) 8879.

IBM AIX PS/2 Personal graPHIGS Programming Interface Version 2

IBM announces the availability of AIX Personal System/2 (PS/2) Personal graPHIGS Programming Interface Version 2. In addition to previously announced capabilities, an installable option and supporting documentation is provided that allows applications written to the Graphical Kernel System (GKS) International Standard (ISO 7942) to run on the product.

Highlights:

- Graphical Kernel System-Compatibility Option and associated documentation
- The option to run in X-Windows will be supported until AIX PS/2 Operating System Version 1.2 becomes available

"**In addition to large memory addressing, productivity gains within the OS/2 environment come from multitasking and multithreading.** (page 5)

"**A major enhancement is the new dual boot capability.** (page 10)

"**OS/2's architecture lays the groundwork for much tighter coordination among different tasks on a user's desktop.** (page 17)

"**It is possible to develop fully object-oriented applications with C under the control of OS/2 Presentation Manager.** (page 22)

"**RDS is designed for multiuser access to a database.** (page 36)

"**The precompiler maintains a list of all host variables.** (page 51)

"**Procedures Language 2/REXX has a free format syntax.** (page 67)

"**This article suggests steps that can be taken to make APPC applications run as fast as possible in OS/2.** (page 80)

"**The RPG II Platform adds an exciting capability for the Personal System/2.** (page 105)

G325-5006-00