Exchange

January/February 1986

Software

1 PCWATCH

9 Handling the BOUND Range Exceeded Exception

12 Compiled BASIC Compatibility with SNA 3270

17 Windows in the Professional Debug Facility

28 Jump Tables in Assembly Language

29 Review of Application Display Management System

Getting Started

31 Memory in the IBM Personal Computer

33 PC Memory Organization

35 Use Care With DOS ASSIGN Command

36 Disabling Call-Waiting

Random Data

37 Memory Intensive Programs on the PC*jr*

39 Inside DOS BACKUP

42 The Last Word

Departments

43 New Products

Exchange of IBM PC Information

IBM

Exchange of IBM PC Information is a monthly publication of the National Distribution Division, International Business Machines Corporation, Boca Raton, Florida, USA.

Managing Editor	Michael Engelberg
Technical Editor	Bernard Penney
Associate Editor/ Design Director	Karen Porterfield
Writer/Automation Consultant	John Warnock
User Group Editor	Steve Mahlum
Editorial Assistant	Wayne Taylor
Automation Consultants	Greg Klipstein
	Sherry Reardon
Illustrator	Jeff Jamison
User Group Support Manager	Gene Barlow

Exchange of IBM PC Information is distributed at no charge to registered PC user groups. To register with us, please write to:

IBM PC User Group Support
IBM Corporation (2900)
P.O. Box 3022
Boca Raton, FL 33431-0922

To correspond with *Exchange*, please write to:

Editor, *Exchange*
IBM Corporation (2900)
P.O. Box 3022
Boca Raton, FL 33431-0922

POSTMASTER: send address changes to *Exchange of IBM PC Information*, IBM Corporation (2900), P.O. Box 3022, Boca Raton FL 33431-0922

PCWATCH

James H. Gilliam, Jr.
Larry K. Raper
IBM Corporation

Editor's note: The authors of this article are the authors of the PCWATCH program. If you have LOADRAM.EXE (a program available through IBM's Directory of Personally Developed Software*), you can download and run a demo of PCWATCH from our Electronic Bulletin Board System, (305) 998-EBBS. You must download the following programs from the <F>iles section of the bulletin board: PCWATCH.RAM, PCWATCH.DOC, and ST.EXE.*

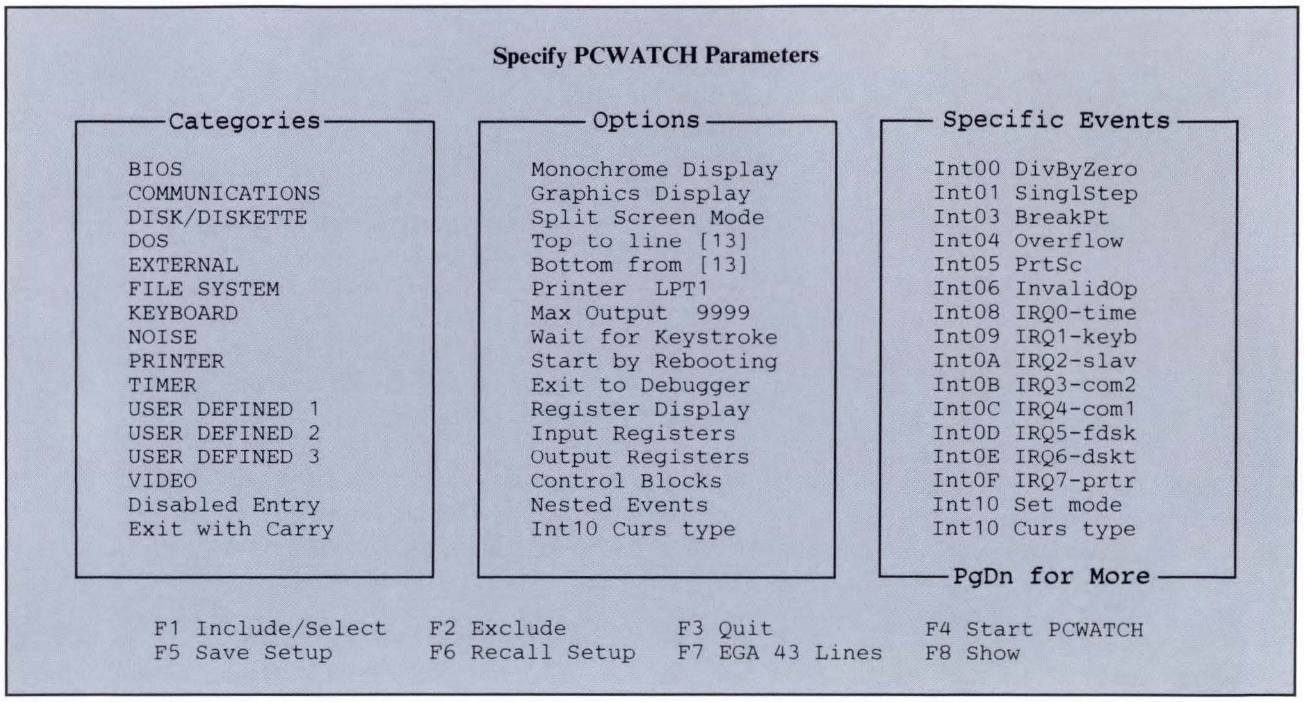
PCWATCH offers a new approach to determining software problems: it monitors your computer's activity while your computer is running—it lets you "see" what is happening. By displaying the inner workings of DOS and application programs, PCWATCH provides an excellent way for the novice to learn how DOS and BIOS functions are used. This approach also gives the serious software developer a highly flexible tool for resolving many complex programming problems. This article discusses PCWATCH, explains some aspects of its design, and gives examples of how it is used.

PCWATCH differs from trace facilities available on other systems in three major ways:

1. It is interactive. You choose program options and monitoring specifications by moving a block cursor and making selections with function keys. Monitoring requests may be generic or specific, and narrowed by further refinements. The keyboard is also used to activate and deactivate monitoring while the computer is running.

2. All results are shown in real time (while the monitored processes are active), making it easy to match internal activity with the visible results of a program.

3. Many aspects of its operation are user-definable, without the need to add user-supplied programming.



```
                        Specify PCWATCH Parameters

 ----Categories-----      ------ Options ------      --- Specific Events ---

   BIOS                   Monochrome Display          Int00 DivByZero
   COMMUNICATIONS         Graphics Display            Int01 SinglStep
   DISK/DISKETTE          Split Screen Mode           Int03 BreakPt
   DOS                    Top to line [13]            Int04 Overflow
   EXTERNAL               Bottom from [13]            Int05 PrtSc
   FILE SYSTEM            Printer  LPT1               Int06 InvalidOp
   KEYBOARD               Max Output  9999            Int08 IRQ0-time
   NOISE                  Wait for Keystroke          Int09 IRQ1-keyb
   PRINTER                Start by Rebooting          Int0A IRQ2-slav
   TIMER                  Exit to Debugger            Int0B IRQ3-com2
   USER DEFINED 1         Register Display            Int0C IRQ4-com1
   USER DEFINED 2         Input Registers             Int0D IRQ5-fdsk
   USER DEFINED 3         Output Registers            Int0E IRQ6-dskt
   VIDEO                  Control Blocks              Int0F IRQ7-prtr
   Disabled Entry         Nested Events               Int10 Set mode
   Exit with Carry        Int10 Curs type             Int10 Curs type

                                                    ----PgDn for More----

     F1 Include/Select   F2 Exclude       F3 Quit          F4 Start PCWATCH
     F5 Save Setup       F6 Recall Setup  F7 EGA 43 Lines  F8 Show
```

Figure 1. The PCWATCH Interface Showing Program Options and Event Selections

Terminology

Two terms are used with PCWATCH for describing interrupts: *service* and *event*. Both are roughly synonymous with *interrupt*, but are used in different ways.

Service Service is similar to the term "software interrupt" in that it covers the functional system interfaces that are supplied on the IBM PC as interrupts. But service is a more general term than interrupt, because it also describes interrupt subfunctions (which may be given a service name with PCWATCH). For example, INT 21 is where DOS services are located. There are over a hundred subfunctions (or services) at this interrupt—Select disk, Open a file, Rename a file, etc.—and each is specified by particular values in the AH or AL registers: AH=0E, AH=3D, AH=56 respectively.

Event Event can be equated with "hardware interrupt," but also includes the specific occurrence of a particular service. This term describes interrupts or programming exception conditions that take place outside the central processor.

Overview of PCWATCH

By "watching" the use of system services, PCWATCH monitors activity at the system interface level (rather than at the machine language instruction level), exposing the same functional abstractions that programs use when interacting with DOS or BIOS. It allows the user to see the invisible internal codes that produce the visible external results.

For its level of function, the program is unusual in not needing (and not providing) a help screen; the PCWATCH interface is intuitively easy to use. A colorful, interactive interface, reproduced in Figure 1, lets you specify all program options and event selections. You can set complex monitoring instructions using a few keystrokes, and you can review and revise these instructions before activating PCWATCH. Although it may appear oriented toward the technical user, novices easily understand the interface as well.

Categories

The *Categories* menu on the interface provides a filtering specification that lets you select the services you want monitored. You can include (using the F1 key) or exclude (F2 key) monitoring services based on categories, individual service names (found in the *Specific Events* menu), or arbitrary combinations of both.

You may monitor a specific service even if you previously excluded it or excluded the category to which it belongs. For example, INT 16, AH=2, "test key" is a part of the NOISE category, yet even after you excluded the entire NOISE category, you could choose to monitor INT 16.

Furthermore, since many functional categories overlap, you may want to begin monitoring broad categories and then exclude lesser categories that are of no interest. A particular service/event may be included in multiple categories; for example, interrupt 21, register AH=2C (the DOS "Get time" service), is assigned to the categories DOS, TIMER, and NOISE.

The NOISE category is a special one that describes all periodic events and continuously used services that occur while the computer is idle. Because these interrupts occur constantly, even when nothing significant is going on, they are seldom of interest in determining problems. NOISE services are interesting only in special circumstances and are usually excluded.

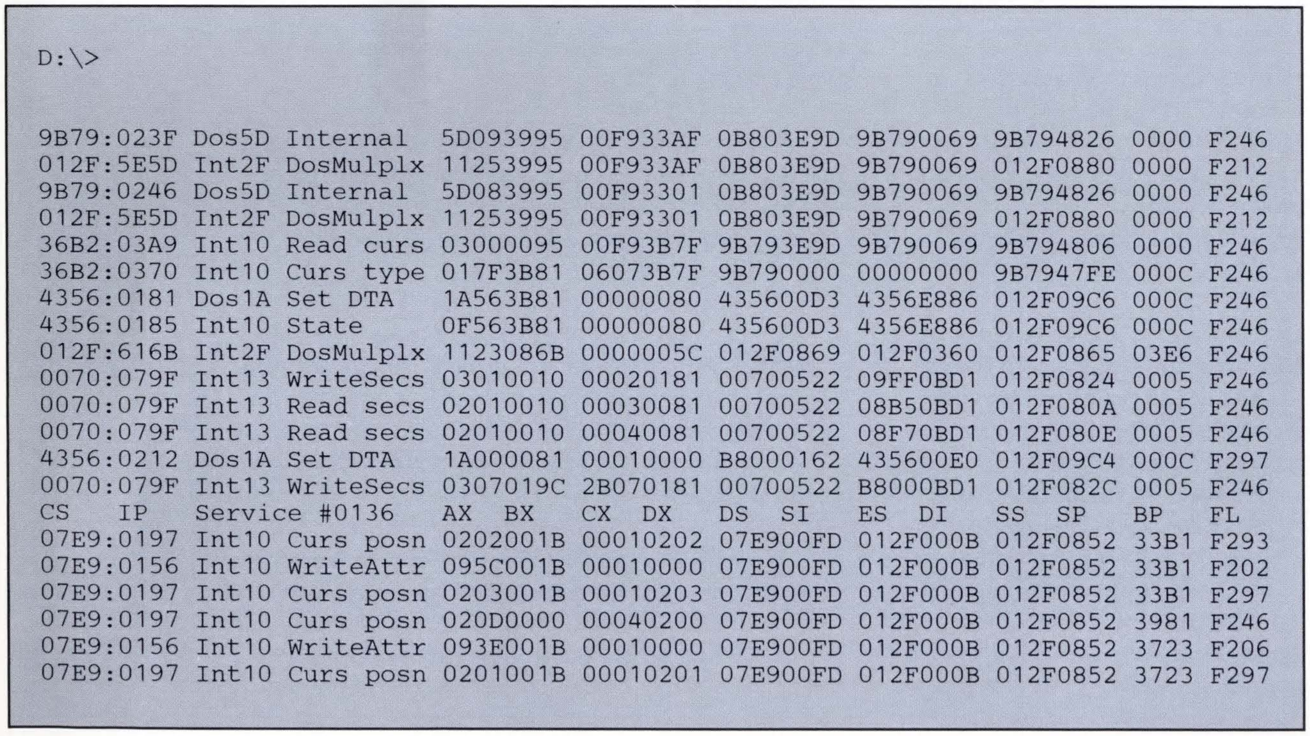
Options

There are several *Options* that can be specified using the PCWATCH interface.

Monochrome Display, Graphics Display, Split Screen Mode, Printer: On a two-display system, PCWATCH can use one display while DOS or application programs use the other. On a single-display system, PCWATCH splits the screen into two partitions, one for its own use and one for DOS's use. (You choose the position and size of the PCWATCH partition.) Figure 2 provides an example of a PCWATCH screen with DOS using three lines and PCWATCH using the rest of the screen. Optional output is also available to a parallel printer.

With an IBM Enhanced Graphics Adapter (EGA) and appropriate display, PCWATCH can operate with 43 lines of output on a single screen. This allows a full 25-line DOS partition and 18 additional lines for PCWATCH on the same display. PCWATCH also supports custom-made EGA fonts that permit even greater numbers of lines on the screen.

Max Output: This option allows for automatic deactivation of PCWATCH after a specified number of events, or at the press of a user-selected key. Similarly, you can temporarily suspend PCWATCH output (and later resume it) via a keystroke, allowing you to selectively monitor services in different portions of one or more programs.



```
D:\>


9B79:023F Dos5D Internal  5D093995 00F933AF 0B803E9D 9B790069 9B794826 0000 F246
012F:5E5D Int2F DosMulplx 11253995 00F933AF 0B803E9D 9B790069 012F0880 0000 F212
9B79:0246 Dos5D Internal  5D083995 00F93301 0B803E9D 9B790069 9B794826 0000 F246
012F:5E5D Int2F DosMulplx 11253995 00F93301 0B803E9D 9B790069 012F0880 0000 F212
36B2:03A9 Int10 Read curs 03000095 00F93B7F 9B793E9D 9B790069 9B794806 0000 F246
36B2:0370 Int10 Curs type 017F3B81 06073B7F 9B790000 00000000 9B7947FE 000C F246
4356:0181 Dos1A Set DTA   1A563B81 00000080 435600D3 4356E886 012F09C6 000C F246
4356:0185 Int10 State     0F563B81 00000080 435600D3 4356E886 012F09C6 000C F246
012F:616B Int2F DosMulplx 1123086B 0000005C 012F0869 012F0360 012F0865 03E6 F246
0070:079F Int13 WriteSecs 03010010 00020181 00700522 09FF0BD1 012F0824 0005 F246
0070:079F Int13 Read secs 02010010 00030081 00700522 08B50BD1 012F080A 0005 F246
0070:079F Int13 Read secs 02010010 00040081 00700522 08F70BD1 012F080E 0005 F246
4356:0212 Dos1A Set DTA   1A000081 00010000 B8000162 435600E0 012F09C4 000C F297
0070:079F Int13 WriteSecs 0307019C 2B070181 00700522 B8000BD1 012F082C 0005 F246
CS   IP   Service #0136   AX   BX   CX   DX   DS   SI   ES   DI   SS   SP   BP   FL
07E9:0197 Int10 Curs posn 0202001B 00010202 07E900FD 012F000B 012F0852 33B1 F293
07E9:0156 Int10 WriteAttr 095C001B 00010000 07E900FD 012F000B 012F0852 33B1 F202
07E9:0197 Int10 Curs posn 0203001B 00010203 07E900FD 012F000B 012F0852 33B1 F297
07E9:0197 Int10 Curs posn 020D0000 00040200 07E900FD 012F000B 012F0852 3981 F246
07E9:0156 Int10 WriteAttr 093E001B 00010000 07E900FD 012F000B 012F0852 3723 F206
07E9:0197 Int10 Curs posn 0201001B 00010201 07E900FD 012F000B 012F0852 3723 F297
```

Figure 2. An Example of a Split Screen in PCWATCH

Wait for Keystroke: By default, all services appear as they occur in real-time, which is the most convenient way to watch for specific low-volume events. But when you monitor several interrupts, you must pause the processor to allow adequate time to review the output. The "slow motion" mode fills the display with the events you have selected and then waits for you to press a key before continuing.

Start by Rebooting: PCWATCH has the ability to simulate a boot sequence before activating a watch specification. It lets you follow the computer's actions as DOS is booted, see the loading of device drivers, and observe the execution of the COMMAND.COM shell. This means that you can monitor not only DOS initialization sequences but also many stand-alone programs.

Exit to Debugger: This option provides the capability to trap a particular occurrence of a designated event and invoke a resident debugging program. A "resident debugger" is already loaded in memory, lying dormant, but ready to respond to some keyboard sequence, pushbutton, breakpoint interrupt, or similar mechanism. After you load a resident debugger, you continue to see the DOS prompt (not a debug prompt) and can operate the system normally. An example of a resident debugger is the Resident Debug Tool, part of the IBM *Professional Debug Facility.*

Also, some interactive debugging programs, such as the Debug program in DOS 3.00 and 3.10, and SYMDEB (included with MASM 3.00 and IBM's new C Compiler 1.00) can be made resident by loading a second-level command processor (COMMAND.COM) from within the debugging program.

This open-ended capability lets you switch to a full-function debugger at a precisely designated moment without requiring you to preset breakpoints in your code. This technique works where breakpoints are not possible (such as stopping in the middle of certain ROM sequences), or are not convenient (such as in the middle of a program that has not been preloaded, but will be fetched dynamically during execution).

Entry to the resident debugging program appears as though an actual breakpoint had occurred. All registers and flags are preserved, and the CS:IP value is set to permit the interrupted program to resume. In addition, PCWATCH is automatically deactivated so that it will not interfere with the debugging program.

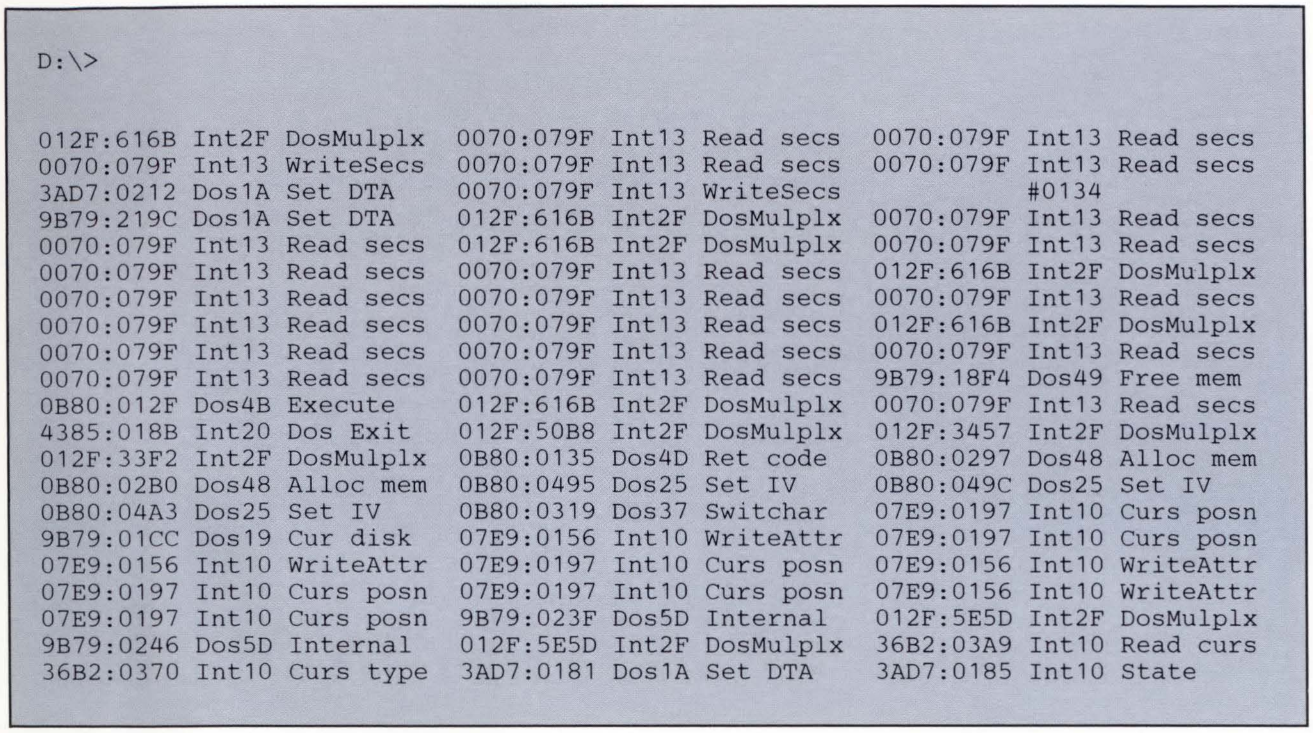
Register Display: An optional register display shows the contents of all registers and flags. You can request input registers, output registers, or both. Whenever output registers are displayed, any registers or flags that were changed by the service are shown with the changes clearly highlighted. This capability ensures that correct parameters have been supplied to a particular service and that the results are what you expected. Unwanted side effects in registers or flags are easy to observe.

If you don't need a register display, PCWATCH produces an event display (shown in Figure 3 on page 5) that lists each occurring service by name and return address. This lets you follow several events with little output and is especially useful if the sequence of events is more important than the contents of the registers.

Also, a formatted display of common control blocks can be used as arguments to many system services. You can display File Control Blocks (FCBs), ASCIIZ strings, diskette parameters, disk status bytes, Network Control Blocks (NCBs), and portions of I/O buffers, etc.

Other Options: The PCWATCH program provides the following functions:

- A "how does it work?" mode that waits for a particular service to occur and then monitors all of that service's internal functions. (Normal exclusion rules let you easily suppress unwanted NOISE events.)
- Recall of previously set specifications. Because repetition usually determines how important some services are in addressing a particular problem, complex sets of PCWATCH specifications can be saved and recalled later for further modification or reuse.
- Compatibility with TopView or PC Network environments. Certain reservations apply when running under TopView or other multitasking and context-switching environments; for instance, you can trace only input registers (not output registers). In addition, PCWATCH must be loaded before TopView.



```
D:\>

012F:616B Int2F DosMulplx  0070:079F Int13 Read secs  0070:079F Int13 Read secs
0070:079F Int13 WriteSecs  0070:079F Int13 Read secs  0070:079F Int13 Read secs
3AD7:0212 Dos1A Set DTA    0070:079F Int13 WriteSecs            #0134
9B79:219C Dos1A Set DTA    012F:616B Int2F DosMulplx  0070:079F Int13 Read secs
0070:079F Int13 Read secs  012F:616B Int2F DosMulplx  0070:079F Int13 Read secs
0070:079F Int13 Read secs  0070:079F Int13 Read secs  012F:616B Int2F DosMulplx
0070:079F Int13 Read secs  0070:079F Int13 Read secs  0070:079F Int13 Read secs
0070:079F Int13 Read secs  0070:079F Int13 Read secs  012F:616B Int2F DosMulplx
0070:079F Int13 Read secs  0070:079F Int13 Read secs  0070:079F Int13 Read secs
0070:079F Int13 Read secs  0070:079F Int13 Read secs  9B79:18F4 Dos49 Free mem
0B80:012F Dos4B Execute    012F:616B Int2F DosMulplx  0070:079F Int13 Read secs
4385:018B Int20 Dos Exit   012F:50B8 Int2F DosMulplx  012F:3457 Int2F DosMulplx
012F:33F2 Int2F DosMulplx  0B80:0135 Dos4D Ret code   0B80:0297 Dos48 Alloc mem
0B80:02B0 Dos48 Alloc mem  0B80:0495 Dos25 Set IV     0B80:049C Dos25 Set IV
0B80:04A3 Dos25 Set IV     0B80:0319 Dos37 Switchar   07E9:0197 Int10 Curs posn
9B79:01CC Dos19 Cur disk   07E9:0156 Int10 WriteAttr  07E9:0197 Int10 Curs posn
07E9:0156 Int10 WriteAttr  07E9:0197 Int10 Curs posn  07E9:0156 Int10 WriteAttr
07E9:0197 Int10 Curs posn  07E9:0197 Int10 Curs posn  07E9:0156 Int10 WriteAttr
07E9:0197 Int10 Curs posn  9B79:023F Dos5D Internal   012F:5E5D Int2F DosMulplx
9B79:0246 Dos5D Internal   012F:5E5D Int2F DosMulplx  36B2:03A9 Int10 Read curs
36B2:0370 Int10 Curs type  3AD7:0181 Dos1A Set DTA    3AD7:0185 Int10 State
```

Figure 3. PCWATCH Event Display

Customizations

To avoid specific BIOS or DOS dependencies, PCWATCH does not attach any semantics to a particular interrupt. All information about what an interrupt represents (including its subfunctions by register codes) is supplied in a file called PCWATCH.TBL.

PCWATCH.TBL is an ASCIIZ file containing definitions of all system services that PCWATCH can intercept. It contains a comprehensive set of definitions for all current BIOS and DOS services (including DOS 3.10), but you can easily modify it to encompass future extensions or additions using any text editor.

PCWATCH is supplied with a companion program PCWTBLB, the PCWATCH Table Builder. PCWTBLB lets you customize PCWATCH's operation by reading the PCWATCH.TBL file and incorporating its definitions directly into the PCWATCH program.

You can customize the following:

- The interrupts (and their subfunctions) that PCWATCH monitors.
- The names and definitions of system services.
- The assignment of system services to functional categories, including three user-defined categories.
- The types of control block formatting applied to data passed as input to a service or returned as output, or both. (Different formatting may be requested for input and output data.)

  The control block formatting options that may be specified are:

  - The data at DS:[DX] as a DOS ASCIIZ string.
  - The data at DS:[DX] as a DOS FCB.
  - The current BIOS diskette parameters.
  - Up to 13 words from the stack.
  - The status bytes from the disk or diskette controller.
  - The data at ES:[BX] as a PC Network Control Block.
  - 32 bytes at DS:[BX]
  - 32 bytes at DS:[SI]
  - 32 bytes at DS:[DX]
  - 32 bytes at ES:[BX]
  - 32 bytes at ES:[DI]

- You can designate certain services as "ONEWAY". A ONEWAY service does not return to its invoker under all circumstances. This information affects how PCWATCH sets and handles traps.

  Some examples of ONEWAY services are:

  - Interrupt 00 - divide by zero program exception
  - Interrupt 20 - normal program termination
  - Interrupt 21 AH=4C - exit with a return code
  - Interrupt 23 - DOS Control-break handler
  - Interrupt 24 - DOS Critical error handler
  - Interrupt 27 - terminate and stay resident

- The keys PCWATCH uses to suspend, resume, or terminate its operations.

- The interrupt used to interface with a resident debugging program. Typical choices would be:

  Interrupt 01 - single step
  Interrupt 02 - nonmaskable
  Interrupt 03 - breakpoint

  but any interrupt may be specified if appropriate.

Because 40% of the current size of PCWATCH is information from the PCWATCH.TBL file, you can use the PCWATCH Table Builder program to generate customized versions of PCWATCH that are optimized for particular situations or application mixes.

Practical Applications of PCWATCH

There are three main applications for PCWATCH, and each has a potentially different user community.

Application	User Community
1. Program debugging	Software developers
2. Problem Determination	Departmental PC Administrators PC Network Administrators Software Evaluators Home Users
3. Learning how programs work	Novices Educators Software Product Reviewers

Program Debugging

The following example, taken from a real-life situation, illustrates several of PCWATCH's functions.

A programmer, after reading the *DOS Technical Reference* manual, decides to write his own special purpose diskette copy routine. Because of the specific nature of his program, he decides to use interrupt 25, the DOS absolute disk read service, for his input. From the description in the DOS manual, he knows that starting with logical sector number X, this service obtains Y 512-byte sectors and places them in a passed buffer area. His program simply provides the values of X and Y and a buffer location.

The programmer concludes that in order to copy one double-sided 360KB diskette, it is sufficient to set X to zero, Y to 720, and pass the address of a 360KB buffer. He writes a small program to confirm this. When tested, the service appears to work (no error indications are returned), but on closer examination, he discovers much of the buffer is unused, and most of the expected data is missing. Furthermore, some program storage areas preceding the buffer are completely destroyed. What went wrong?

Conventional debugging approaches, such as using breakpoints, tracing procedure calls, or selective display of program variables will simply verify that all of the inputs to the DOS service have been properly specified, and that no error flags have been returned.

In desperation, a programmer might proceed by trial-and-error techniques to learn more about why the DOS service is not functioning as described in the manual.

PCWATCH addresses this problem directly. Since the internal functioning of DOS interrupt 25 is suspect, it is a natural candidate for PCWATCH's "How does it work?" mode. To find out, the programmer runs PCWATCH, includes the Int25 DosAbsRd function, excludes NOISE and EXTERNAL categories (seldom of interest when trying to understand the in-line actions of a function), and selects the "Nested Events" option ("Nested Events" is the program option that triggers the "How does it work" mode of operation). Then he starts PCWATCH and reruns the faulty program.

When the INT 25 service is finally invoked (and not until then), PCWATCH displays all of the internally used DOS and BIOS services that are not in the excluded categories. When the INT 25 service concludes and returns to the invoking program, PCWATCH automatically ceases monitoring.

PCWATCH clearly shows that the key BIOS service employed by INT 25 is INT 13 Read Sector. By "watching" the parameters generated by DOS and passed in registers when using this service, the programmer observes the following:

- DOS *does*, in fact, read all 720 sectors from the diskette—9 sectors at a time, resulting in 80 calls to the BIOS INT 13 Read Sector service.
- The offset portion of the buffer address, originally supplied by the user program, is incremented by 1200 hex (4608 decimal, equivalent to 9 x 512) on each use of the BIOS Read Sector service.

However, the offset portion of the address overflows when it exceeds 64KB and wraps around to zero again, with no corresponding adjustment of the segment portion of the address.

This explains what goes wrong. Although not mentioned in its description, the DOS Absolute Disk Read service is clearly not prepared to deliver any more than 64KB of data with one invocation. Also the full 64KB of data is obtainable only if the initial value of the offset portion of the buffer address is set to zero.

DOS fails to detect an error and returns an indication of successful completion.

This example illustrates how a rather subtle aspect of DOS's behavior can be readily brought to the surface with a simple application of PCWATCH.

Problem Determination

Software developers frequently debug, but problem determination is a more universal activity.

One of the strongest virtues of the IBM PC is its open architecture. An unfortunate aspect of open architecture is that the integration of separately obtained hardware or software parts is frequently left to the user. Generally, when used or installed correctly, all of the pieces work together smoothly, but occasionally the correct method is not clear, and hardware or software incompatibilities result. Determining how or why a product is not functioning properly, or understanding its system requirements, is a common exercise in addressing a problem.

Large organizations sometimes cope with this by establishing focal points for problem resolution. Departmental PC administrators, network administrators, software evaluators, and others ensure that individual PC users need not waste time wrestling with such problems. However, the home user usually has only limited resources for problem solving.

Consider a user confronting a new program that refuses to run. Perhaps the program appears to do nothing, hangs the system, or produces an insightful message like "Required file missing" or "Unable to create profile." At this point that most users will decide to read the documentation. But when the documentation does not lead to any apparent remedy, what other answer is available?

PCWATCH can expose a host of potential problems by showing precisely what a program is trying to do.

A typical solution for this type of problem might consist of the following steps:

- Retry the program. Maybe the problem will go away.
- Call the dealer, supplier, or possibly a technical support number.
- Find another user and ask for help.
- Make random alterations in the hardware or software configuration to see what effect they might have.

Program developers or technical people may even tackle the program with a debugger by tracing through the machine code trying to understand what the program needs.

PCWATCH provides you with a simple alternative that even non-technical users can apply to such problems. When confronting a problem caused by something unknown, you can monitor broad categories of system activity, and then narrow the focus of interest selectively until you isolate sources of conflict or point directly to the problem.

If the problem is that the program appears to do nothing, or causes the machine to hang, then include the PCWATCH categories BIOS and DOS and exclude NOISE. This configuration displays a fairly comprehensive overview of system activity that may lead to discovering the cause of the failure. If the program terminates with messages that fail to isolate the problem sufficiently, just include the category DOS (and exclude NOISE). This can expose a host of potential problems, many with obvious procedural remedies, by "showing" precisely what the program is trying to do. (A software product reviewer using PCWATCH could speak with authority when commenting on a product's efficient use of DOS services.)

If a program is trying to open a file that doesn't exist, PCWATCH displays the file name. If a leading or trailing backslash was needed but omitted from a command argument, sometimes it can be readily spotted by examining path names constructed by the program. Whatever the cause, you can observe first-hand all directory and file references leading up to the error.

If you have ever wondered why the diskette light has come on, you can immediately see what is going on at that exact point in the program. If DOS says "Abort, retry, or ignore" while reading a hard disk or a diskette, you can easily determine the precise sector that has gone bad. The list of other problems you can pinpoint goes on and on.

Limitations

For debugging or general problem determination, PCWATCH works best when the problem can be repeated. Being able to repeat (i.e., re-create) the problem will let you narrow a PCWATCH specification as you learn more about the problem. A problem that cannot be recreated must be caught in the act, and for this it may be necessary to operate the computer continuously with PCWATCH active.

Conclusions

In some sense PCWATCH contributes to the illusion that it is possible to see what the computer is actually doing. For beginners, it is a giant leap from the conceptual model of the computer to one of discrete concepts like programs and files. For people interested in understanding the relationship between programs, the DOS operating system, and the more primitive BIOS support, PCWATCH is a natural vehicle for filling the gaps in their knowledge. Furthermore, for experienced programmers, it offers a new approach to the perplexing problems that software developers encounter.

Handling the BOUND Range Exceeded Exception

Greg Gruse
IBM Corporation

The IBM Personal Computer Macro Assembler version 2.00 includes the BOUND instruction, which requires the Intel 80286 main processor, available only in the IBM Personal Computer AT.

The BOUND instruction tests to ensure that a given value does not fall outside specified bounds. If the value is outside the bounds, the BOUND instruction creates a Bound Range Exceeded exception. However, instead of handling the BOUND Range Exceeded exception, the Personal Computer AT will do something altogether unrelated—it will print the screen repeatedly, and you will have to re-boot the system.

The reason this happens, and a program for circumventing the situation, are given in the following text.

The 80286 Processor, Interrupt 5, and the BIOS

The 80286 Processor uses interrupt 5 as the Bounds Exceeded exception interrupt. When the result of a BOUND instruction is a Bounds Exceeded exception, the 80286 generates an interrupt 5 and sends the interrupt to the Personal Computer AT's BIOS.

However, to ensure compatibility with the BIOS in all other members of the IBM Personal Computer family, the interrupt 5 vector in the Personal Computer AT's BIOS points to the Print Screen routine. Therefore, the interrupt 5 generated by the Bounds Exceeded exception causes the screen to print.

Meanwhile, the 80286 is programmed to anticipate that the exception condition has been handled and corrected. To verify this, the 80286 returns control to the BOUND instruction that generated the interrupt 5, and executes the BOUND instruction again. But the original exception condition was not handled and corrected. Therefore, the second execution of the BOUND instruction causes another screen to print. This situation will continue indefinitely until you re-boot your system.

Circumventions and Solutions

The simplest way to prevent this situation from occurring is to avoid using the BOUND instruction. However, you can use it if you install the assembly language program shown in Figure 1, arbitrarily named INT5TEST.

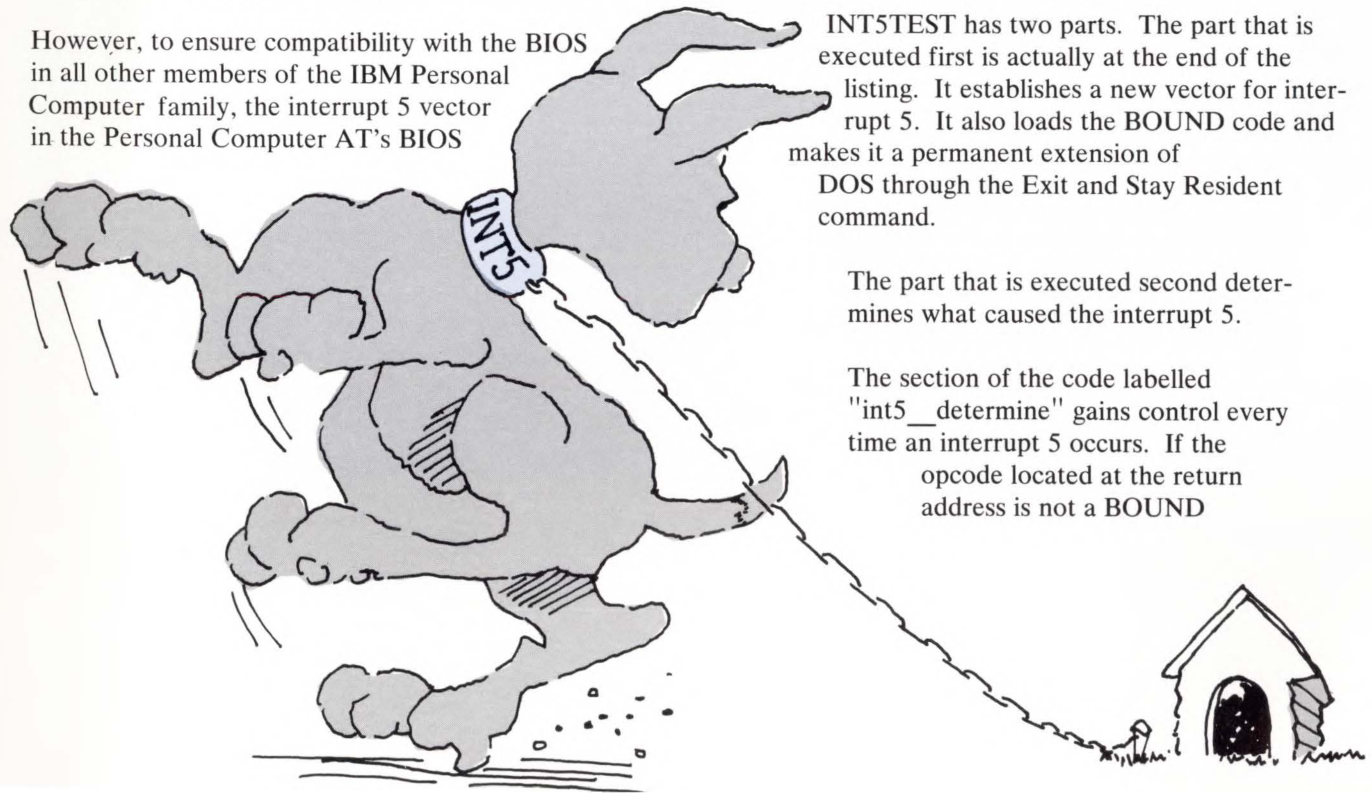
INT5TEST is intended as a resident extension of DOS. It must be brought up and made resident every time you turn on your Personal Computer AT. Also, INT5TEST must be loaded as the last routine affecting interrupt 5 (the print screen interrupt).

For example, if you first load INT5TEST and later load the DOS GRAPHICS routines that also change interrupt 5, you will get incorrect results. The GRAPHICS code will intercept both BOUND and PRINT SCREEN functions. But GRAPHICS knows nothing about BOUND, so (if you're currently in graphics mode) it will proceed to print the screen.

INT5TEST has two parts. The part that is executed first is actually at the end of the listing. It establishes a new vector for interrupt 5. It also loads the BOUND code and makes it a permanent extension of DOS through the Exit and Stay Resident command.

The part that is executed second determines what caused the interrupt 5.

The section of the code labelled "int5_determine" gains control every time an interrupt 5 occurs. If the opcode located at the return address is not a BOUND



instruction, control passes to the normal interrupt 5. However, if a BOUND instruction caused the interrupt 5, INT5TEST takes two actions. First, it advances the saved CS:IP to point to the instruction following the BOUND instruction. The BOUND instruction can have several different lengths, so INT5TEST must determine how far to move the IP. Second, INT5TEST writes a message to the screen, indicating that a bound exception occurred.

In INT5TEST, the code that begins at the label "bound_exception" is given strictly as an example. Depending on your particular application, you may want to take a different action when the bound exception occurs. (If you want to ignore the exception, you should instead avoid using the BOUND instruction.)

Editor's note: IBM has written and tested the programs that follow. However, IBM does not guarantee that they contain no errors. The compiled version of this program, INT5TEST.COM, is available for downloading from the <F>iles section of IBM's Electronic Bulletin Board System, (305) 998-EBBS.

```
        page    ,132
        title   Determine if bounds error has occurred

cseg    segment para
        org     100h
        assume  cs:cseg

entpt:  jmp     start               ; go to initialization code
int5_vector     dd      ?           ; save location for interrupt 5
int5_determine  proc    far
        push    bp
        mov     bp,sp               ; addressing into stack
        push    ds
        push    bx                  ; save registers
        mov     bx,[bp+2]           ; get IP
        mov     ds,[bp+4]           ; get CS
        cmp     byte ptr [bx],062h  ; look for bound op-code
        je      bound_exception

; Here if print screen

        pop     bx
        pop     ds                  ; recover the registers
        pop     bp
        jmp     cs:[int5_vector]    ; go to the original

; Here if bound exception
; This routine looks at the mod-r/m byte to determine the length of the
; bound instruction.  The routine increments the IP value on the stack to
; point to the instruction following the bound.

bound_exception:
        push    ax
        mov     al,[bx+1]       ; get mod-r/m byte
        and     al,11000111b    ; isolate mod and r/m bits
        cmp     al,00000110b    ; look for the special case
        je      add_4
        and     al,11000000b    ; leave only the mod bits
        cmp     al,00000000b    ; look for mod = 00
        je      add_2
        cmp     al,01000000b    ; look for mod = 01
        je      add_3
```

Figure 1. INT5TEST Program Code

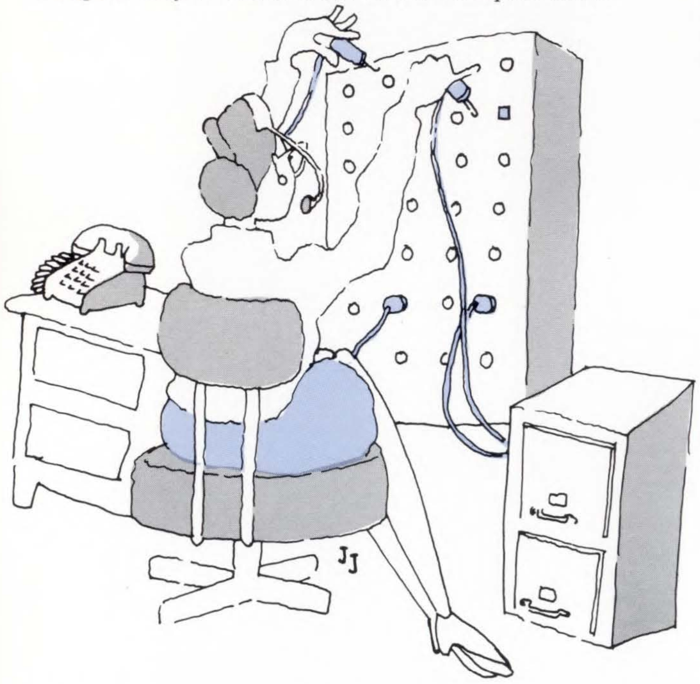
```
; Skip checking for mod = 11, since that is a register designation, and
; has no meaning for a doubleword operand as required by BOUND.
; Therefore, the only one left is mod = 10, which has a two-byte
; extension.
add_4:  add     word ptr [bp+2],4   ; add 4 to pass opcode, mod-r/m and
                                    ; two- byte displacement
        jmp     add_exit
add_3:  add     word ptr [bp+2],3   ; add the value to IP on stack
        jmp     add_exit
add_2:  add     word ptr [bp+2],2

add_exit:
        call    show_something      ; this routine is just for show
        pop     ax
        pop     bx
        pop     ds
        pop     bp
        iret                        ; return to the bound problem

; This routine is here just to display something when a bound exception
; occurs.  This routine and the call are unnecessary.

show_something  proc    near
        push    dx
        mov     dx,cs               ; simple routine to say
        mov     ds,dx               ;    "Bound exception"
        lea     dx,bound_message
        mov     ah,9
        int     21h                 ; print a message
        pop     dx
        ret
show_something  endp

bound_message   db      'Bound exception',10,13,'$'

int5_determine  endp
code_end        label   near

start   proc    near
        assume cs:cseg,ds:cseg,ss:cseg,es:cseg

; Reset interrupt 5 to point to bound checking code

        cli                     ; no interrupts during this time
        mov     ax,3505h
        int     21h             ; get the current interrupt 5
        mov     word ptr int5_vector, bx
        mov     word ptr int5_vector+2, es
        mov     dx,offset int5_determine
        mov     ax,2505h
        int     21h             ; set interrupt 5 interrupts back on
        sti
        lea     dx,code_end
        int     27h             ; exit and stay resident
start   endp
cseg    ends
        end     entpt
```

Figure 1. INT5TEST Program Code (cont.)

Compiled BASIC Compatibility with Network SNA 3270

Charles Lovell
IBM Corporation

Editor's note: The information in this article applies only to BASIC Compiler 1.00. PC Network compatability is included in BASIC Compiler 2.00.



The IBM PC Network SNA 3270 Emulation Program (hereafter called PC Network 3270) lets you load and execute programs as "alternate tasks" while the emulation program is operating in a stand-alone, gateway, or network station configuration.

In alternate task mode, you can run almost any program you wish. However, BASIC programs normally cannot run as alternate tasks because they are incompatible with PC Network 3270. This article explains how to modify BASIC programs compiled with BASIC Compiler 1.00 so that you can execute them as alternate tasks in the PC Network 3270 environment.

Selecting Alternate Task Mode

PC Network 3270's Setup menu contains an entry that asks if you want alternate tasks. Select "Yes" to activate alternate tasks. Once activated, this mode will remain active until you select "No" from the Setup menu.

After you activate alternate tasks, reload PC Network 3270 to handle swapping between PC Network 3270 and the alternate task. To switch between PC Network 3270 and the alternate task, press Alt + Esc.

Alternate Task Requirements

To run as an alternate task, a BASIC program must abide by several rules:

- It must run under DOS 2.10 or later.
- It must be relocatable.
- It must interface with DOS and BIOS using the standard DOS and BIOS interrupt and function call interfaces (i.e., it must not CALL to DOS or BIOS interrupts and functions).
- It must not alter any of the following hardware or software interrupts used by PC Network 3270:

INT	Name
8	Timer Interrupt Service Routine (ISR)
9	Keyboard ISR
A	Local Area Network ISR
B	Communications ISR
C	Communications ISR
10	Video BIOS
16	Keyboard BIOS
21	DOS Function Call

- It should obtain the addresses of these and any other interrupt handling routines *only* with the DOS function GET VECTOR (35H), which will ensure that the program receives the proper addresses.
- It should alter interrupts other than those listed above *only* by using the DOS function SET INTERRUPT VECTOR (25H), and it must restore those that it alters (using SET INTERRUPT VECTOR) before it terminates.

BASIC Programs

To run as a PC Network 3270 alternate task, a BASIC program must be compiled (not interpreted). However, a compiled BASIC program is not compatible with PC Network 3270 immediately after it is compiled and linked. To make your compiled BASIC program compatible, you must modify it to change certain things done by the BASIC Compiler. The modifications are relative simple.

Your program must have been compiled with the native BASIC Compiler rather than with a version of the compiler that has been modified for timer interrupt (1CH) chaining. If you used any version other than the native BASIC Compiler, recompile and re-link your program with the native version first.

If, after you apply the modifications given here, your program still does not work as a PC Network 3270 alternate task, one of the following reasons may apply:

1. Your program was compiled using a modified BASIC Compiler; you must recompile it with the native BASIC Compiler.
2. Your program depends on the timer interrupt chaining of the modified BASIC Compiler; you must change your program.
3. Your program loads and executes the BASIC Interpreter (either BASIC.COM or BASICA.COM); the BASIC Interpreter is not compatible with PC Network 3270.
4. Your program does not satisfy all of the requirements listed above under "Alternate Task Requirements".

Before you begin applying the modifications given below, please be sure you understand the following assumptions and cautions:

- Modifications will be made to your program, *not* to the BASIC Compiler.
- You will make backup copies of your compiler and program diskettes *before* you make the modifications to your program.
- The modifications are limited in scope to the codes generated by the BASIC Compiler itself and cannot address unique situations in your program, such as those mentioned above, that may cause it to be incompatible.
- This article assumes you have some knowledge of the IBM PC Macro Assembler and that you understand the byte reversal in the Intel processors.

All compiled BASIC programs have an .EXE or .COM file that you call by name to begin executing the program. That file is the one you will modify.

One way to make modifications is to use the DOS Debug program, DEBUG.COM, documented in your *DOS Reference* manual. Another way is to use the Resident Debug Tool program in the IBM *Professional Debug Facility*. This article assumes you will use DOS Debug.

Since Debug cannot write a file in .EXE format, you must first change the name of the .EXE file you are going to modify. Assuming your program's name is XYZ, rename your .EXE file using the following command:

```
A>RENAME XYZ.EXE XYZ.TMP<Enter>
```

where <Enter> indicates you should press the Enter key. To begin editing your file with Debug, type:

```
A>Debug XYZ.TMP<Enter>
```

Debug will respond with a dash, which is its prompt:

Modification #1: Resetting BASIC Interrupt Offsets

To find the first group of code to be modified, type:

```
S CS:0 L FFFF C7 06 00 00<Enter>
```

Debug will respond with a location in the format

```
-ssss:tttt
```

where ssss is the segment address (usually the value in CS) and tttt is the offset address.

Next, disassemble the code. Substitute the actual values that Debug gave you for ssss and tttt, and type the Debug command:

```
U ssss:tttt<Enter>
```

Debug will then display the disassembled listing shown below. In this listing, there are 20 bytes of code in the leftmost column (6 bytes on the first line, 4 bytes on the second, 6 bytes on the third, and 4 bytes on the fourth line). The middle and right columns contain the disassembled commands and operands. Note the byte reversal; for example, on the fourth line, the last two bytes in the left column are 12 00, but in the right column they have been reversed to 00 12. As another example, aabb is disassembled into bbaa.

Debug will display:

```
C7060000aabb  MOV WORD PTR [0000],bbaa
8C0E0200      MOV [0002],CS
C7061000ccdd  MOV WORD PTR [0010],ddcc
8C0E1200      MOV [0012],CS
```

where aabb and ccdd are offsets. The disassembled listing reverses the bytes to bbaa and ddcc. Make note of these offset values because you will use them in making this modification #1.

The code you have just displayed uses MOV instructions to establish the offsets to the (interrupts) 00H and 04H BASIC routines. However, PC Network 3270 requires applications to use DOS function call 25H to set interrupt vectors. Modification #1 will make this happen, but one small problem must be resolved first.

The code that you are going to modify occupies 20 bytes in memory, but the modification normally takes 21 bytes. Therefore, to maintain the integrity of the registers you use in the modification, some adjustments have to be made to gain the needed byte. Therefore:

1. Compare aabb and ccdd.
2. If the values for aa and cc are the same (and they probably are), use Choice #1 below.
3. If aa and cc are not the same, use Choice #2 below.

In both Choice #1 and Choice #2, you will enter 20 new bytes to replace the 20 bytes listed above. However, the 20 new bytes are different for each choice.

Now go to either Choice #1 or Choice #2 in this text.

Choice #1: Keeping the DX Register

You will enter 20 new bytes to replace the 20 bytes listed above. The new bytes you will enter are:

1E 52 0E 1F BA aa bb B8 00 25
CD 21 B6 dd B0 04 CD 21 5A 1F

(You can see these same 20 bytes listed in the left column of the next disassembled listing.)

In the 20 new bytes, substitute the actual values for aa, bb and dd.

To make Modification #1, type the Debug Edit command:

```
E ssss:tttt<Enter>
```

Debug responds with:

```
ssss:tttt C7.
```

Following the period, enter the first byte of the patch:

```
1E<Space>
```

where <Space> indicates you should press the Space bar.

Now continue to enter each successive byte (52, 0E, 1F, etc.), and after each byte press <Space>. After you have entered the 20th byte, press <Enter>.

To verify that you have entered all 20 new bytes correctly, you can disassemble the modification you have just made. To disassemble it, enter:

```
U ssss:tttt<Enter>
```

Debug should then list the following disassembled code:

```
1E     PUSH DS
52     PUSH DX
0E     PUSH CS
1F     POP  DS
BAaabb MOV  DX,bbaa ;offset to
                    ;  00H BASIC
                    ;  routine
B80025 MOV  AX,2500
CD21   INT  21
B6dd   MOV  DH,dd   ;low byte of
                    ;  offset
                    ;to 04H BASIC
                    ;  routine
B004   MOV  AL,04
CD21   INT  21
5A     POP  DX
1F     POP  DS
```

Of course, the disassembled listing will not contain the two comments I have added at the right for clarity.

If your modified code does not look exactly like the listing above, use Debug again to correct it.

Now skip over Choice #2 and read Modification #2.

Choice #2: Changing the DX Register

In choice #2, you will enter 20 new bytes to replace the 20 bytes listed above. This choice is less desirable because it does not preserve the integrity of the DX register.

The bytes you will enter are:

1E 0E 1F BA aa bb B8 00 25 CD
21 BA cc dd B0 04 CD 21 1F 90

(You can see these same 20 bytes listed in the left column of the next disassembled listing.)

In the 20 new bytes, substitute the actual values for aa, bb, cc and dd.

To make Modification #1, type the Debug Edit command:

```
E ssss:tttt<Enter>
```

Debug responds with:

```
ssss:tttt C7.
```

Following the period, enter the first byte of the patch:

```
1E<Space>
```

where <Space> indicates you should press the Space bar.

Now continue to enter each successive byte (0E, 1F, BA, etc.), and after each byte press <Space>. After you have entered the 21st byte, press <Enter>.

To verify that you have entered all 20 new bytes correctly, you can disassemble the modification you have just made. To disassemble it, enter:

```
U ssss:tttt<Enter>
```

Debug should then list the following disassembled code:

```
1E      PUSH DS
0E      PUSH CS
1F      POP  DS
BAaabb  MOV  DX,bbaa ;offset to 00H
                     ;BASIC routine
B80025  MOV  AX,2500
CD21    INT  21
BAccdd  MOV  DX,ddcc ;offset to 04H
                     ;BASIC routine
B004    MOV  AL,04
CD21    INT  21
1F      POP  DS
90      NOP
```

Of course, the disassembled listing will not contain the two comments I have added at the right for clarity.

If your modified code does not look exactly like the listing above, use Debug again to correct it.

Modification #2: Using Function Call 25H

The second modification is the one that permits your compiled BASIC program to load and execute with PC Network 3270.

To find the second group of code to be modified, type:

```
S CS:0 L FFFF C7 06 90 00<Enter>
```

Debug will respond with a location in the format

```
vvvv:wwww
```

where vvvv is the segment address and wwww is the offset address.

Next, disassemble the code. Substitute the actual values that Debug gave you for vvvv and wwww, and type the Debug command:

```
U vvvv:wwww<Enter>
```

Debug will then display the following disassembled listing:

```
C7069000eeff  MOV  WORD PTR [0090],ffee
8C0E9200      MOV  [0092],CS
C7066C00gghh  MOV  WORD PTR [006C],hhgg
8C0E6E00      MOV  [006E],CS
C7067000iijj  MOV  WORD PTR [0070],jjii
8C0E7200      MOV  [0072],CS
```

This code sets interrupts 24H, 1BH and 1CH respectively. Just as in Modification #1, you must change this code to use DOS function call 25H.

The above code occupies 30 bytes, but Modification #2 requires only 28 bytes, so the modification contains two extra NOP instructions at the end.

The 30 new bytes you will enter are shown in the leftmost column of the next disassembled listing. Be sure to substitute the actual values for ee, ff, gg, hh, ii and jj.

To make Modification #2, type the Debug Edit command:

```
E vvvv:wwww<Enter>
```

Debug responds with:

```
vvvv:wwww C7.
```

Following the period, begin entering the 30 new bytes, each one followed by <Space>, and the last one followed by <Enter>.

To verify that you have entered all 30 new bytes correctly, you can disassemble the modification you have just made. To disassemble it, enter:

```
U vvvv:wwww<Enter>
```

Debug should then list the following disassembled code:

```
1E      PUSH DS
52      PUSH DX
0E      PUSH CS
1F      POP  DS
BAeeff  MOV  DX,ffee ;offset to 24H
                     ;BASIC routine
B82425  MOV  AX,2524
CD21    INT  21
BAgghh  MOV  DX,hhgg ;offset to 1BH
                     ;BASIC routine
B01B    MOV  AL,1B
CD21    INT  21
BAiijj  MOV  DX,jjii ;offset to 1CH
                     ;BASIC routine
B01C    MOV  AL,1C
CD21    INT  21
5A      POP  DX
1F      POP  DS
90      NOP
90      NOP
```

As before, verify that your modification looks exactly like the above disassembled listing. If it does not, use Debug again to correct it.

Modification #3: Leaving the 8259 Processor Alone

To find the third group of code to be modified, type:

```
S CS:0 L FFFF E4 21 0C 18<Enter>
```

Debug will respond with a location in the format

```
xxxx:yyyy
```

where xxxx is the segment address and yyyy is the offset address.

Next, disassemble the code. Substitute the actual values that Debug gave you for xxxx and yyyy, and type the Debug command:

```
U xxxx:yyyy<Enter>
```

Debug will then display the following disassembled listing:

```
E421  IN   AL,21
0C18  OR   AL,18
E621  OUT  21,AL
```

This code resets the two communication interrupts on the 8259 chip. When this happens, communication ceases. Obviously you don't want this to happen while PC Network 3270 is communicating. Modification #3 ensures that the communications link will remain active when your compiled BASIC program terminates.

The six new bytes you will enter are shown in the leftmost column of the next disassembled listing.

To make Modification #3, type the Debug Edit command:

```
E xxxx:yyyy<Enter>
```

Debug responds with:

```
xxxx:yyyy E4.
```

Following the period, begin entering the six new bytes, each one followed by <Space>, and the last one followed by <Enter>.

To verify that you have entered all six new bytes correctly, you can disassemble the modification you have just made. To disassemble it, enter:

```
U xxxx:yyyy<Enter>
```

Debug should then list the following disassembled code:

```
E421  IN    AL,21
90    NOP
90    NOP
E621  OUT   21,AL
```

Again, verify that your modification looks exactly like the above disassembled listing. If not, use Debug again to correct it.

Saving Your Modifications and Concluding

After you have made and verified all three modifications, save them by typing:

```
W<Enter>
```

This command tells Debug to write the modifications onto your file XYZ.TMP.

After Debug has written the modified file XYZ.TMP, terminate Debug by entering:

```
Q<Enter>
```

Finally, rename XYZ.TMP back to XYZ.EXE:

A>RENAME XYZ.TMP XYZ.EXE<Enter>

Unless your compiled BASIC program has its own inherent problems such as those listed at the beginning of this article, it now should be compatible with the PC Network SNA 3270 Emulation Program. You can now load and execute it as an alternate task under PC Network 3270.

Windows in the Professional Debug Facility

John Warnock
IBM Corporation

Resident Debug Tool Windowing

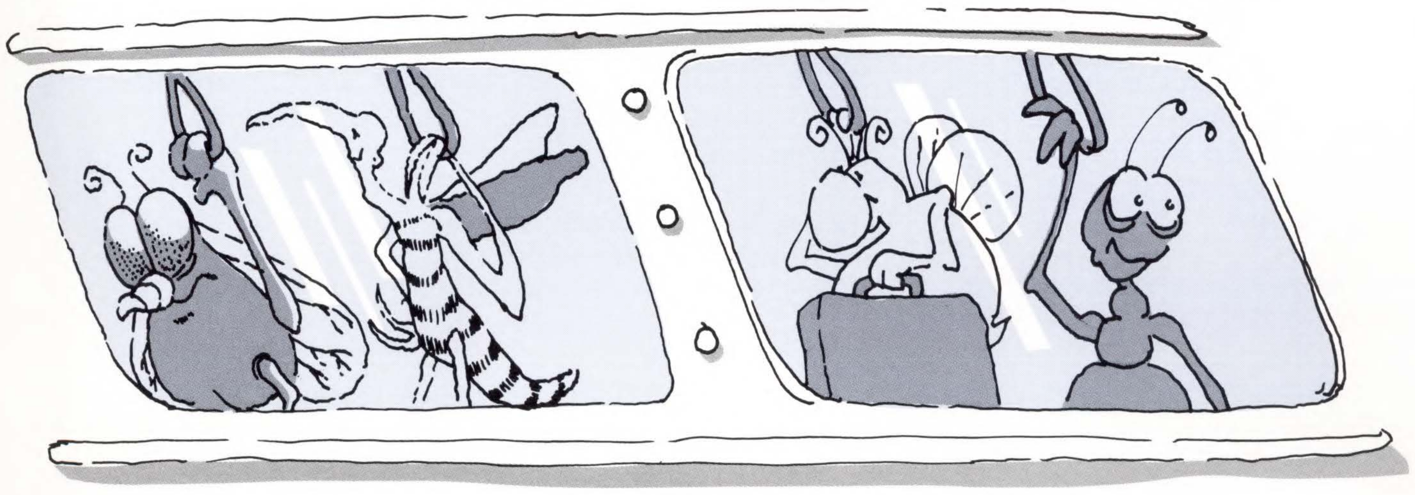
The Professional Debug Facility's Resident Debug Tool (RDT) features extensive windowing capabilities. You can use the window area (lines 12 through 25) of the RDT screen to:

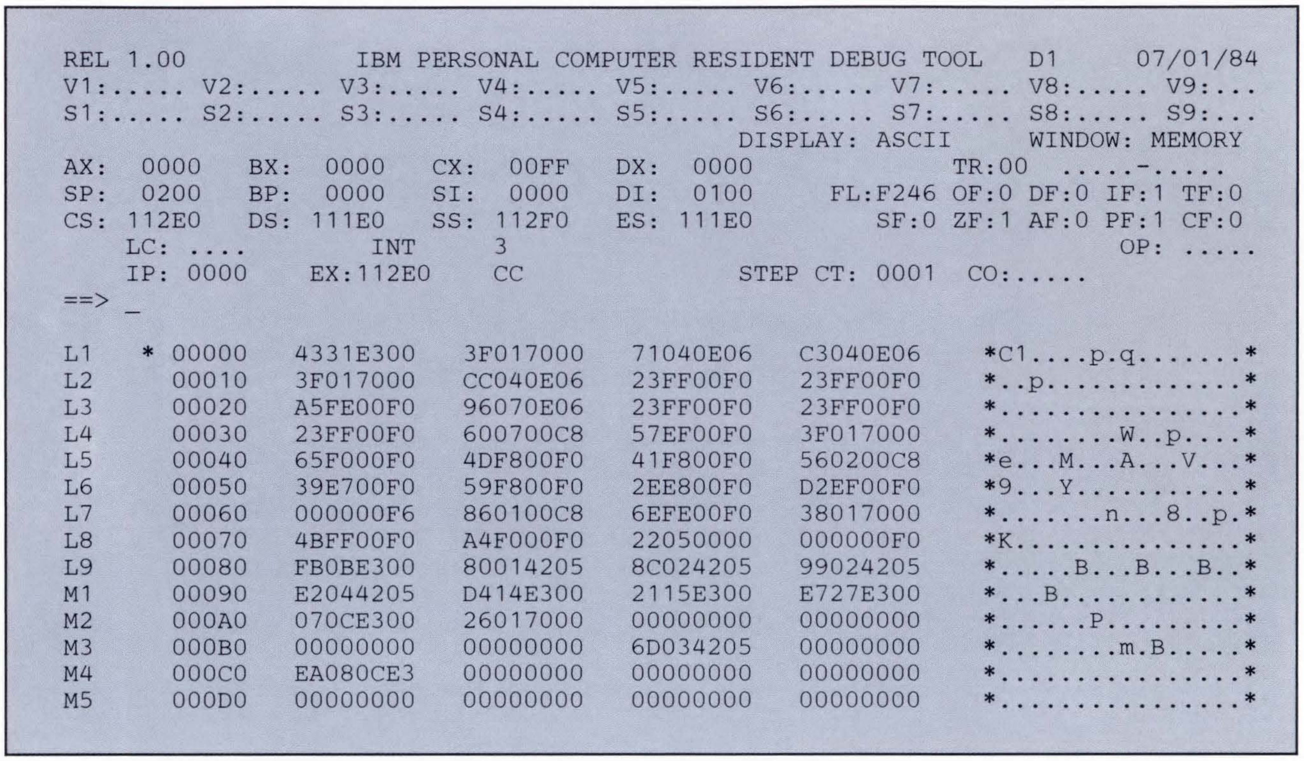
- display and alter memory
- display disassembled instructions and alter their hexadecimal representations
- display RDT's instruction trace buffer, which contains instructions saved according to the trace options you specified
- display the state of the Math Co-processor if one is installed

Features Common in All RDT Display Screens

All RDT display screens have the same information in the upper ten lines of the screen. Figure 1 on page 18 shows a typical screen that displays memory. In Figure 1, the top line gives the program release number, program title, current display number (explained later) and program date.

Lines 2 and 3 contain 18 scratchpad variables, V1 through V9 and S1 through S9, which can be set as breakpoints as well as variables. When used as breakpoints, these variables allow you to set addresses at which your program will stop so you can examine its status.





```
REL 1.00          IBM PERSONAL COMPUTER RESIDENT DEBUG TOOL    D1      07/01/84
V1:..... V2:..... V3:..... V4:..... V5:..... V6:..... V7:..... V8:..... V9:...
S1:..... S2:..... S3:..... S4:..... S5:..... S6:..... S7:..... S8:..... S9:...
                                              DISPLAY: ASCII      WINDOW: MEMORY
AX:  0000   BX:  0000   CX:  00FF   DX:  0000              TR:00  .....-.....
SP:  0200   BP:  0000   SI:  0000   DI:  0100        FL:F246 OF:0 DF:0 IF:1 TF:0
CS: 112E0   DS: 111E0   SS: 112F0   ES: 111E0           SF:0 ZF:1 AF:0 PF:1 CF:0
    LC: ....         INT     3                                        OP: .....
    IP: 0000    EX:112E0     CC                 STEP CT: 0001  CO:.....
==> _

L1    * 00000   4331E300   3F017000   71040E06   C3040E06    *C1....p.q.......*
L2      00010   3F017000   CC040E06   23FF00F0   23FF00F0    *..p.............*
L3      00020   A5FE00F0   96070E06   23FF00F0   23FF00F0    *................*
L4      00030   23FF00F0   600700C8   57EF00F0   3F017000    *........W..p....*
L5      00040   65F000F0   4DF800F0   41F800F0   560200C8    *e...M...A...V...*
L6      00050   39E700F0   59F800F0   2EE800F0   D2EF00F0    *9...Y...........*
L7      00060   000000F6   860100C8   6EFE00F0   38017000    *.......n...8..p.*
L8      00070   4BFF00F0   A4F000F0   22050000   000000F0    *K...............*
L9      00080   FB0BE300   80014205   8C024205   99024205    *.....B...B...B..*
M1      00090   E2044205   D414E300   2115E300   E727E300    *...B............*
M2      000A0   070CE300   26017000   00000000   00000000    *......P.........*
M3      000B0   00000000   00000000   6D034205   00000000    *........m.B.....*
M4      000C0   EA080CE3   00000000   00000000   00000000    *................*
M5      000D0   00000000   00000000   00000000   00000000    *................*
```

Figure 1. Sample RDT Memory Window Screen

On line 4, DISPLAY shows whether the window area contains ASCII or EBCDIC characters. WINDOW shows the current window mode; in Figure 1 it is MEMORY. RDT has four other window modes:

- DISASM for disassemble
- TRACEP for partial trace
- TRACEF for full trace
- COPROC for Math Co-processor

Lines 5, 6 and 7 display the general registers and specific registers in the leftmost four columns.

At the right of line 5 is the trace option display, TR. It shows the active trace option number and (if you chose the range option) the starting and stopping addresses for the trace. (The trace options are discussed later.)

At the right of lines 6 and 7 is the flag status display. The entire value for the flag register is shown first, followed by each flag bit. As with the registers, you can modify the contents of the flags by typing the appropriate flag command.

Line 8 contains the pseudo-location counter value (LC), and the disassembly of the current instruction (the one most recently executed). The instruction parameter address is shown under the label OP.

The instruction pointer (IP) appears on line 9, followed by the address and code of the last instruction executed (EX). Also on line 9 are the step count (STEP CT) and code origin (CO).

Line 10 is the command line, where you type all your commands. RDT returns messages to you on line 11.

The remaining 14 lines are called the window area. The window area is different in each of the five window modes; for example, in memory window mode it is called the memory window area.

The cursor can access the command line and the window area. In all five window modes, you use the arrow keys to move the cursor between the command line and the window area. You also can move the cursor from the window area to the command line by pressing the Home key.

Saving Multiple Display Screens

You can save up to nine different display screens, each with its own line settings and variable values.

When you are ready to save a display, type the command:

```
Dx [=] Dy
```

or simply

```
Dx Dy
```

where y is the number (1 through 9) of the display you want to save, and x is the new number (1 through 9) you want to give your saved display. For example, if your current memory display is D1, and you want to save it as D4, you should type D4 D1. D4 is a "snapshot" of D1. You can now continue modifying D1. At a later time, you can retrieve display D4 by simply typing D4 on the command line.

Now let's examine the facilities offered within each window mode.

Memory Windowing

Memory windowing (MW) lets you perform memory display (hexadecimal, EBCDIC or ASCII), alteration and scrolling functions.

Lines L1 through M5 are the memory window area. Each line in the memory window area contains a line number, memory address, hexadecimal representation of 16 bytes in memory, and the equivalent character representation (either ASCII or EBCDIC).

The ASCII and EBCDIC character formats may be filtered or unfiltered. When the display of characters is filtered, only the letters and numbers appear; all other hexadecimal codes are shown as dots. When unfiltered, all hexadecimal codes are displayed, although many of them will appear as special characters. To switch between filtered and unfiltered character display, place the cursor in the memory window and press the F3 key.

Displaying Memory

The memory window area is divided into windows. Each window displays a contiguous block of memory. You can display from one to 14 windows at one time. A window can begin on any line, and can be as small as one line or as large as 14 lines.

RDT places an asterisk after a line number when a new window begins on that line. The display of one window continues on subsequent lines until another window begins.

Windows need not be ordered sequentially by address. A window containing a block of high memory can precede a window containing a block of low memory.

To create a memory window and see a particular part of memory, type its address directly over the existing address on any line in the memory window area. As you type each digit of the new address, RDT instantly displays the contents of memory at the five-digit address it currently sees.

For example, suppose the address on line L1 originally shows 00000, but you want to change it to display memory beginning at address 12345. When you type the first digit, 1, you will see memory beginning at address 10000; when you type the second digit, 2, you will see memory at address 12000; and so on. (You also can enter a new address on a line by first storing it in one of the scratchpad variables and then pointing to that variable with an RDT command. For example, if variable S2 contains address 12345 and you want to enter that address on line L1, you can use the command L1=S2. The address in S2 then becomes the new address on line L1.)

Memory windowing lets you perform memory display, alteration and scrolling functions.

When you change the address on line L1, RDT places an asterisk after the line number to indicate the beginning of a new memory window.

To remove a window from the memory window area, move the cursor inside that window, then press the Del key. You also can remove a window by typing one of the L1 through M5 commands with no operand following. When a window is deleted, the asterisk that marked it is removed, and the window above it is extended.

The following is a short scenario illustrating how the memory window area behaves.

Suppose you want to see the contents of memory starting at address 12345 and you want these contents to start on line L5. After you type 12345 over the existing address on line L5, RDT places an asterisk after the line number to indicate that a new window begins on line L5. The new window extends to the bottom of the memory window area, and lines L6 through M5 have the addresses 12355, 12365, ..., 123D5. Now the first window is reduced to lines L1 through L4 with addresses 00000, 00010, 00020 and 00030.

The concept of indefinite windows has been implemented for memory patching.

Next, if you type the address 01ABC on line M1, another window is created on the last five lines, and an asterisk is placed after line number M1. The addresses on lines M2 through M5 become 01ACC, 01ADC, 01AEC and 01AFC. Also, the second window is shortened to lines L5 through L9, with addresses 12345 through 12385.

To see how the delete function works, place the cursor within the second window and press the Del key. Addresses 12345 through 12385 disappear; the window directly above is extended from L4 (address 00030) to L9 (address 00080); and the asterisk is removed from line L5. The third window, lines M1 through M5, is left unchanged.

As long as the third window exists, and you do not delete any part of it, it will not be overlaid when you create another window. To verify this, create a fourth window beginning on line L6 at address 34567. An asterisk will be placed after line number L6; lines L7 through L9 will display addresses 34577, 34587 and 34597; and lines M1 through M5 will be left intact.

Altering Memory

When the cursor is inside a memory window, you can alter the memory displayed in that window. You can alter memory with either hexadecimal, EBCDIC or ASCII input. You alter memory by moving the cursor to the appropriate byte in either the hexadecimal (left) portion or the character (right) portion of a line of memory. You then type in your alterations.

Each keystroke modifies memory. Half of one byte is modified for each hexadecimal digit entered, and one full byte is modified for each EBCDIC or ASCII character entered.

Note: Any memory modifications you make are temporary until you make them permanent. Temporary modifications will be overridden when you load the next program into the system, turn the computer off, or issue the System Reset (SR) command in RDT. To make your changes permanent, you should issue a Write Diskette (WD) instruction or modify your source program.

As you proceed to modify a line of memory, eventually you will come to the end of the line. The cursor wraps to the beginning of the next line so you can continue.

RDT has a built-in safeguard that prevents you from altering memory when you intended only to enter a new address. After you type the fifth digit of an address, the cursor will not move to the right. To move the cursor into the hexadecimal data display area, you must use one of the arrow keys or the space bar.

You can compare a line of memory as it appears now to the same line as it appeared previously, because RDT can display a "changing" line of memory next to a "saved" line of memory. As you type data into the "changing" line, memory is updated. The corresponding "saved" line is not updated, allowing you to see its contents as they were before you modified them.

Scrolling Through Memory

The concept of indefinite windows has been implemented for memory patching. When the last byte of a memory section has been modified, RDT automatically scrolls the memory display window one byte to the right. This displays the next byte of memory so you can modify it if you wish. You can therefore patch a string of bytes (hex, EBCDIC, or ASCII) of indefinite length without entering a new address.

The space bar and backspace key also cause automatic scrolling of the memory display window. If you want to move the cursor without causing automatic scrolling, you should use one of the four cursor arrow keys or the tab or Caps Lock key.

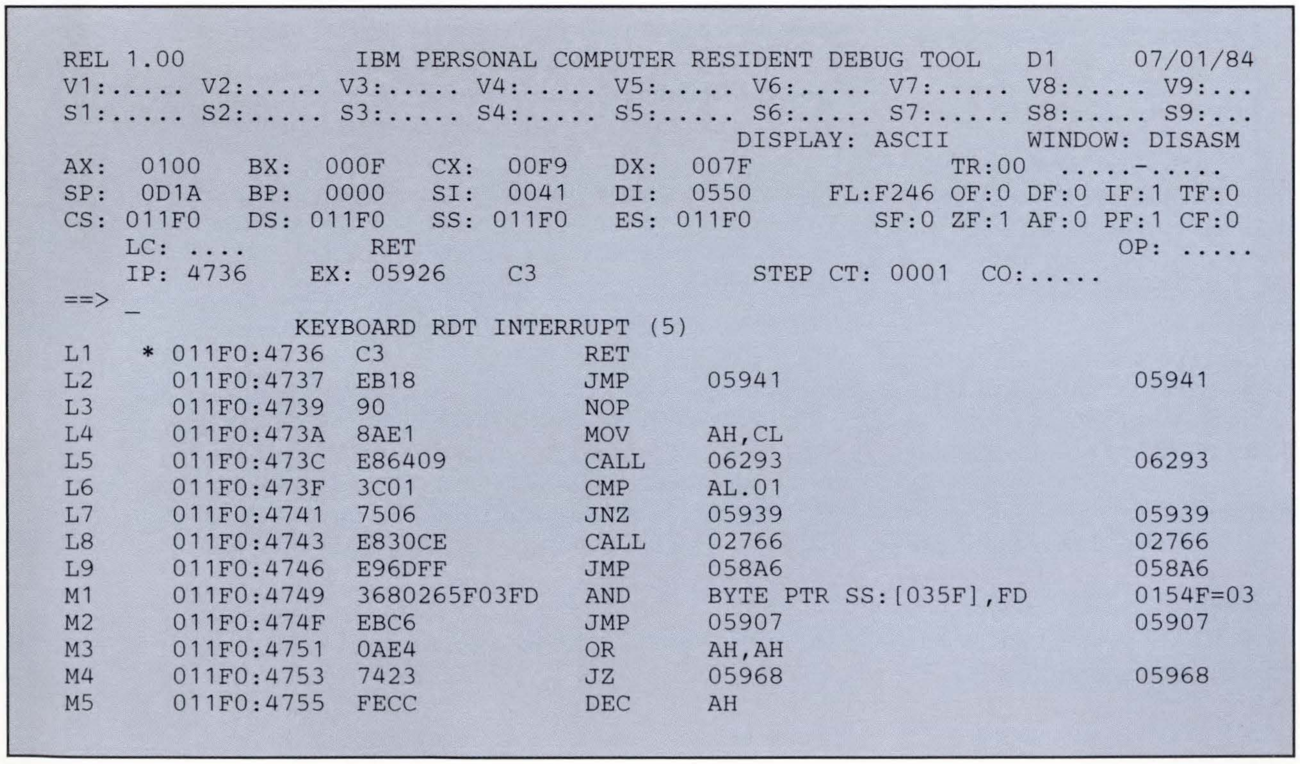
The following keys control cursor motion and scrolling when the cursor is in the memory window area:

- **Caps Lock**: Moves the cursor to the beginning of the next line down. When the cursor is already on the bottom line, the next line it moves to is the command line.
- **Space bar**: Functions similarly to the right arrow key. However, the space bar is a valid data key for entering ASCII or EBCDIC data, and it will cause automatic scrolling if you press it when you have reached the end of a hexadecimal memory block.
- **F3**: Toggles between filtered and unfiltered displays of EBCDIC and ASCII characters.
- **Ctrl + Up arrow**: Scrolls a memory block up one line. (The cursor may be located anywhere within the memory section you want to scroll.)
- **Ctrl + Down arrow**: Scrolls a memory block down one line.
- **Ctrl + Right arrow**: Scrolls a memory block one byte to the right.
- **Ctrl + Left arrow**: Scrolls a memory block one byte to the left.
- **Ctrl + Tab**: Moves the cursor to the beginning of the next EBCDIC or ASCII memory block.
- **Pg Up**: Scrolls a memory block up by the number of lines in the section.
- **Pg Dn**: Scrolls a memory block down by the number of lines in the section.

Disassemble Windowing

With disassemble windowing (DW), you can display disassembled instructions in the window area, and you can alter the hexadecimal representations of instructions and data areas in your application program.

Figure 2 shows the disassemble window on lines L1 through M5. Each line consists of a line number, CS value (code segment, the starting address of a 64KB memory segment), IP value (instruction pointer, the displacement value for an instruction within that code segment), a disassembled instruction in hexadecimal form, and the corresponding character equivalent of the disassembled instruction.



```
REL 1.00           IBM PERSONAL COMPUTER RESIDENT DEBUG TOOL   D1      07/01/84
V1:..... V2:..... V3:..... V4:..... V5:..... V6:..... V7:..... V8:..... V9:...
S1:..... S2:..... S3:..... S4:..... S5:..... S6:..... S7:..... S8:..... S9:...
                                              DISPLAY: ASCII    WINDOW: DISASM
AX:  0100   BX:  000F   CX:  00F9   DX:  007F            TR:00  .....-.....
SP:  0D1A   BP:  0000   SI:  0041   DI:  0550      FL:F246 OF:0 DF:0 IF:1 TF:0
CS: 011F0   DS: 011F0   SS: 011F0   ES: 011F0         SF:0 ZF:1 AF:0 PF:1 CF:0
    LC: ....        RET                                               OP: .....
    IP: 4736     EX: 05926     C3                STEP CT: 0001  CO:.....
==> _
                KEYBOARD RDT INTERRUPT (5)
L1   * 011F0:4736  C3            RET
L2     011F0:4737  EB18          JMP     05941                        05941
L3     011F0:4739  90            NOP
L4     011F0:473A  8AE1          MOV     AH,CL
L5     011F0:473C  E86409        CALL    06293                        06293
L6     011F0:473F  3C01          CMP     AL.01
L7     011F0:4741  7506          JNZ     05939                        05939
L8     011F0:4743  E830CE        CALL    02766                        02766
L9     011F0:4746  E96DFF        JMP     058A6                        058A6
M1     011F0:4749  3680265F03FD  AND     BYTE PTR SS:[035F],FD        0154F=03
M2     011F0:474F  EBC6          JMP     05907                        05907
M3     011F0:4751  0AE4          OR      AH,AH
M4     011F0:4753  7423          JZ      05968                        05968
M5     011F0:4755  FECC          DEC     AH
```

Figure 2. Sample RDT Disassemble Window Screen

The instruction that is disassembled is the one pointed to by the combination of the CS and IP values. It also may contain an operand or jump location, and a byte or word value if the instruction disassembly indicates that such a value exists for the referenced instruction.

The disassemble window contains disassembled instructions from one or more contiguous blocks of memory. You can display up to 14 windows at one time. A window can begin on any line, and can be as small as one instruction or as large as 14 instructions.

RDT places an asterisk after a line number when a new block begins on that line. The display of one block of disassembled instructions continues on subsequent lines until a new block begins or until the last line is reached.

A helpful feature in the disassemble window is the concept of the "current" disassemble display window. The first disassemble display window (beginning with the first line in the window area) is continually updated to display the next instruction(s) that your application program will execute.

Windows do not have to be in sequential order. A window containing a block of high memory may precede a window containing a block of low memory.

Displaying a Disassembly

To create a disassemble window and see a particular part of your application program, type its address over any existing address on the screen. As you type each digit in the new address, RDT instantly displays the program segment at the location specified by the 5-byte segment plus 4-byte offset.

For example, suppose your code segment (CS) is 19DA0 and your instruction pointer (IP) is 0756. The combination of CS and IP produces your instruction address, 1A4F6, which is displayed as EX.

Suppose you want the lower half of the disassemble window to show the program at 1F46. You can access this address by changing either the instruction pointer or the code segment.

To change the instruction pointer, press the Down arrow or Caps Lock key until the cursor is at line L8. Press the tab key once to move the cursor to the instruction pointer. Now type 1F46. As you type each character, the lower half of the window jumps. The address lines now have the addresses 19DA0:1F46, 19DA0:1F48, etc. RDT places an asterisk next to the L8 to indicate the beginning of a new window.

To change the code segment, on the command line type M5=CS+1F46-0756,0756. This sets the value of line M5 of the display to 1B590:0756 (the code segment plus the new offset minus the old offset, at the old offset). Line M5 looks like line L8.

As in memory windowing, when you press the Del key, the window containing the cursor is erased and the previous window is extended. To delete the window beginning at L8, place the cursor at line L8 using the Down arrow key, and press the Del key. Alternately, you can delete the window by typing (on the command line) L8, the number of the first line in the window. The window and its contents are removed, and the previous window is extended to and including line M4. Line M5 is still a separate window.

RDT displays disassembled instructions based on the starting address of memory in your computer, 0000H. The code segment of your program is expressed as a number of bytes relative to 0000H. Subsequent instruction addresses in your program also are based on 0000H as the starting point.

The starting address of your program can change, depending on where DOS loads it. For example, DOS might load your program starting at address 0EFBH. The disassemble window shows your first instruction address as 0EFBH, and each succeeding instruction is an increment of 0EFBH.

The starting address 0EFBH is not the same as the starting address 0000H in your assembly listing. However, RDT gives you a way to make the disassemble window look like an assembly language listing, where your program's starting address is 0000H.

On the command line, type CO=CS+IP (or the code segment plus whatever your program's starting displacement may be) and press Enter. This sets your code origin (CO) variable equal to the starting address of your program and your pseudo location counter (LC) equal to 0000H. The pseudo LC value matches the one in the assembly language listing. If an instruction is displayed in CO/LC format, a dollar sign ($) is displayed to the left of the instruction.

The rules for updating an instruction's hexadecimal representation and operands in CO/LC format are the same as the rules for CS/IP format. However, the rules for updating an instruction's address are somewhat different. In CS/IP format, you alter either the code segment or instruction pointer to change the window address. In CO/LC format, you can change only the LC values for the line.

You update the pseudo LC value by typing over it as you would type over the IP value. A new disassemble window is created by moving the cursor to any given line and typing over the LC value, or by entering one of the label (L1-M5) commands to specify a new LC for a given line. This affects the line's LC value just as though you had typed the "LC" command on the command line.

Altering a Disassembly

One of the most powerful features of the disassemble window is that it lets you interactively modify your program and the disassembly listing. You alter your program by entering new processor op-codes and values in hexadecimal. (This requires that you have some knowledge of 8088 and 80286 processor instructions.)

To do this, move the cursor to the hexadecimal instruction on any line. As you type a new hexadecimal instruction, each keystroke modifies a half-byte of memory. As this happens, RDT dynamically updates the following lines of disassembled code to reflect your change. The cursor does not automatically wrap around to the next hexadecimal instruction as it does in memory windowing. RDT allows you to modify only the hexadecimal instructions, not the disassembled portion of the code.

When doing program modification, you should split the window into two matching windows by typing the same address halfway down. Use one window for changes and keep the "original" in case you need to restore any values.

Although you normally cannot modify the disassembled instructions, you can modify the operand values of instructions that reference bytes or words in memory. From the last half-byte in the hexadecimal instruction, press the Right arrow key to move the cursor to the byte or word within the disassemble window instruction. Then type the new operand value over the old one.

Another powerful feature of the disassemble window is its ability to scroll down, left, or right.

Note: Any memory modifications you make are temporary until you make them permanent. Temporary modifications will be overridden when you load the next program into the system, turn the computer off, or issue the System Reset (SR) command in RDT. To make your changes permanent, you should issue a Write Diskette (WD) instruction or modify your source program.

Scrolling Through a Disassembly

Another powerful feature of the disassemble window is its ability to scroll down, left, or right. You scroll by pressing the Ctrl key in combination with the Down, Left, or Right arrow keys, with the cursor anywhere in the window to be scrolled. Scrolling down causes the current IP value at the top of the disassemble window to be replaced with the value of what would be the next instruction in the window, and the entire window is updated to reflect the scroll operation. Similarly, scrolling left or right causes 1 to be subtracted from or added to the window's beginning IP value, and the disassemble display window is updated.

Use the PgDn key to scroll the disassemble display window down one full screen. The last instruction in the window is found, its IP value retrieved, and its length determined. This value is added to the retrieved IP value, and the result becomes the IP value of the first instruction in the disassemble window.

Trace Windowing

Trace windowing (TW) displays the program instructions and other information placed in the RDT trace buffer as the computer steps through the program. These instructions appear in the window area of the RDT display screen, according to trace options you specify with the "TR" and the "TW" commands.

When you start RDT, you can define a larger or smaller trace buffer with the "T size" parameter, where size is a hex value for the number of program instructions to store. For DOS 1.10 the minimum size is hex 20, and the maximum is hex D0. DOS 2.00 and higher versions allow values from hex 20 to hex 924. If you do not use the parameter, hex 20 (32 instructions) is the default. Each instruction takes up 28 bytes of memory.

Starting a Trace

You may initiate a trace from any window. The format of the "TR" command is:

```
TR [= option[,begin,end]]
```

where

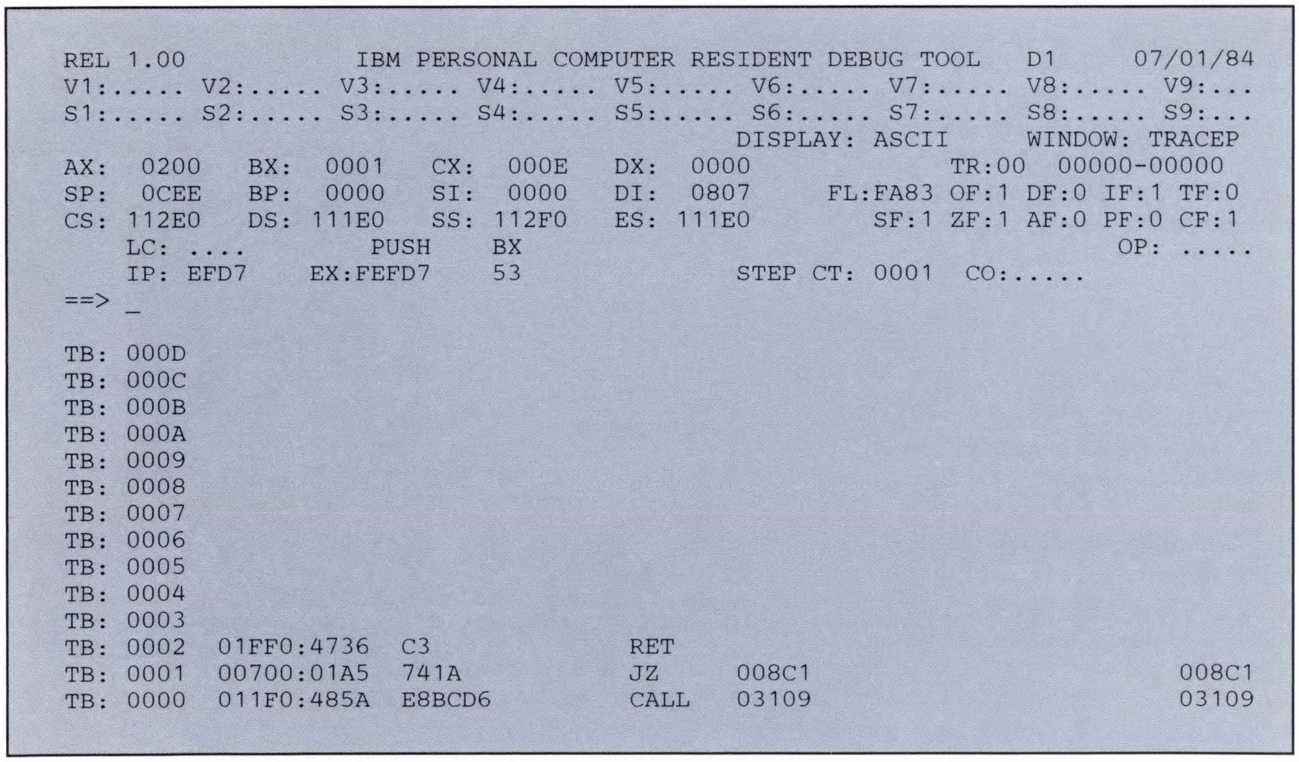
option is the type of trace activity:

00	no tracing
01	trace all instructions
02	trace all application instructions
04	trace instructions within a range
08	break on buffer entry
10	trace JUMP instructions
20	trace CALL instructions
40	trace INTERRUPT instructions
80	break on buffer full

begin is the beginning instruction address for RDT to place in the trace buffer (option 4 only).

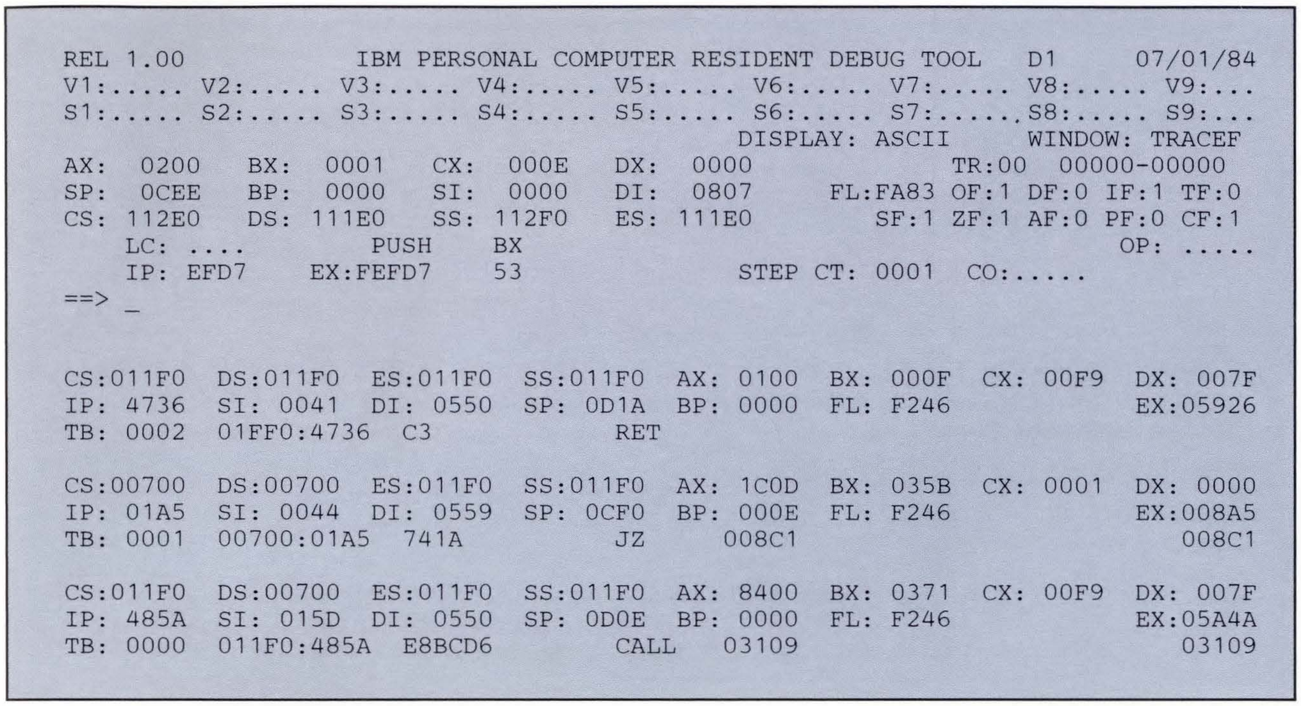
end is the last instruction address for RDT to place in the trace buffer (option 4 only).

You can use options 10, 20, and 40 in combination with options 01, 02, and 04 for greater control. You



```
REL 1.00            IBM PERSONAL COMPUTER RESIDENT DEBUG TOOL    D1      07/01/84
V1:..... V2:..... V3:..... V4:..... V5:..... V6:..... V7:..... V8:..... V9:...
S1:..... S2:..... S3:..... S4:..... S5:..... S6:..... S7:..... S8:..... S9:...
                                              DISPLAY: ASCII     WINDOW: TRACEP
AX:  0200   BX:  0001   CX:  000E   DX:  0000             TR:00  00000-00000
SP:  0CEE   BP:  0000   SI:  0000   DI:  0807      FL:FA83 OF:1 DF:0 IF:1 TF:0
CS: 112E0   DS: 111E0   SS: 112F0   ES: 111E0          SF:1 ZF:1 AF:0 PF:0 CF:1
    LC: ....        PUSH    BX                                         OP: .....
    IP: EFD7    EX:FEFD7    53                STEP CT: 0001   CO:.....
==> _

TB: 000D
TB: 000C
TB: 000B
TB: 000A
TB: 0009
TB: 0008
TB: 0007
TB: 0006
TB: 0005
TB: 0004
TB: 0003
TB: 0002  01FF0:4736  C3             RET
TB: 0001  00700:01A5  741A           JZ     008C1                          008C1
TB: 0000  011F0:485A  E8BCD6         CALL   03109                          03109
```

Figure 3. Sample RDT Partial Trace Window Screen



```
REL 1.00           IBM PERSONAL COMPUTER RESIDENT DEBUG TOOL    D1      07/01/84
V1:..... V2:..... V3:..... V4:..... V5:..... V6:..... V7:..... V8:..... V9:...
S1:..... S2:..... S3:..... S4:..... S5:..... S6:..... S7:.....,S8:..... S9:...
                                             DISPLAY: ASCII     WINDOW: TRACEF
AX:  0200   BX:  0001   CX:  000E   DX:  0000              TR:00  00000-00000
SP:  0CEE   BP:  0000   SI:  0000   DI:  0807    FL:FA83 OF:1 DF:0 IF:1 TF:0
CS: 112E0   DS: 111E0   SS: 112F0   ES: 111E0        SF:1 ZF:1 AF:0 PF:0 CF:1
    LC: ....         PUSH    BX                                      OP: .....
    IP: EFD7    EX:FEFD7    53                STEP CT: 0001  CO:.....
==> _


CS:011F0  DS:011F0  ES:011F0  SS:011F0  AX: 0100  BX: 000F  CX: 00F9  DX: 007F
IP: 4736  SI: 0041  DI: 0550  SP: 0D1A  BP: 0000  FL: F246            EX:05926
TB: 0002  01FF0:4736  C3              RET

CS:00700  DS:00700  ES:011F0  SS:011F0  AX: 1C0D  BX: 035B  CX: 0001  DX: 0000
IP: 01A5  SI: 0044  DI: 0559  SP: 0CF0  BP: 000E  FL: F246            EX:008A5
TB: 0001  00700:01A5  741A            JZ      008C1                      008C1

CS:011F0  DS:00700  ES:011F0  SS:011F0  AX: 8400  BX: 0371  CX: 00F9  DX: 007F
IP: 485A  SI: 015D  DI: 0550  SP: 0D0E  BP: 0000  FL: F246            EX:05A4A
TB: 0000  011F0:485A  E8BCD6          CALL    03109                      03109
```

Figure 4. Sample RDT Full Trace Window Screen

might mentally add the options you want to include and use the hexadecimal result in the command. A simpler method of combining options is to type each desired option separated by a plus sign. Options 01, 02, and 04 are mutually exclusive.

Displaying a Trace

There are two kinds of trace windows: a partial trace window and a full trace window. Typing TW or TW 0 on the command line places you into partial trace window mode. The WINDOW variable on line 4 shows the value "TRACEP".

Figure 3 is an example of an RDT display screen in partial trace window mode. The display window area contains 14 trace buffer entries, one to a line. Each line, if not empty, contains the hexadecimal and disassembled representation of an instruction in the RDT trace buffer. The instruction is preceded by the CS and IP register values which, when combined, determine the address of the instruction at the time it was placed in the trace buffer.

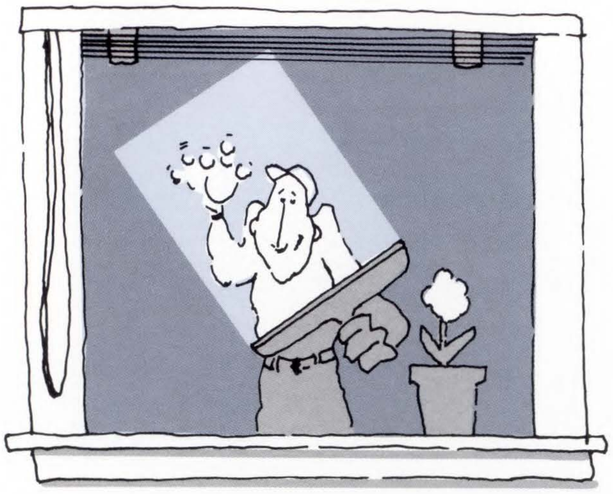
Typing TW 1 at the command line places RDT in full trace window mode. The WINDOW variable on line 4 shows the value "TRACEF".

Figure 4 is an example of an RDT display screen in full trace window mode. The display window area contains three trace buffer entries. Each entry is three lines long. If not empty, an entry displays the code segment and instruction pointer, along with the instruction in hexadecimal and disassembled form. In addition, each entry has two lines that show all of the 8088 processor registers at the time the instruction was stepped.

In both trace windowing modes, the "TB" value for each trace buffer entry shows the relative position of the instruction in the RDT trace buffer. The value of the "TB" variable is the hex offset from the end of the RDT trace buffer for that trace entry. For example, an entry with the "TB" variable of hex "0000" is the last instruction that was placed in the trace buffer. An entry with the "TB" variable of hex "0001" is the second-to-last instruction placed in the trace buffer, and so on.

Each time a new buffer entry is added, the old TB variables are incremented by 1 and moved down in the buffer. This lets you view instructions in the trace buffer in a natural top-to-bottom fashion, where the latest instruction in the trace window appears at the bottom of the screen.

Instructions accumulate in the trace window as they are trapped. You can tell the Resident Debug Tool to



interrupt when the buffer is full; otherwise, the instructions will continue to roll forward, the oldest instruction being pushed out of the buffer. You can issue a clear trace buffer command (TC) to clear the buffer yourself.

Scrolling Through a Trace

You can move through the trace window area to the limits of the size of the RDT trace buffer. Move the cursor off the command line, using the Down arrow or Up arrow keys. This moves the cursor to the bottom "TB" variable of the trace window area. From there you can change the window view of the trace buffer in two ways.

The first method is to scroll the trace buffer window. To move the window up or down one entry, press the Ctrl key plus either the Up or Down arrow keys. To scroll a full screen, press the PgUp or PgDn key. You can use either method until you reach either end of the buffer.

The second method is to type over the "TB" variable with a new value. RDT allows you to change this bottom "TB" variable only if the new value allows all the "TB" variables in the window area to fit within the valid RDT trace buffer range. This is the only modification you may make in the trace window area.

An asterisk next to a line in the trace display window area indicates that one or more instructions may be missing from that trace buffer. Hardware limitations on certain Intel 8088 processors cause the trap flag not to be recognized immediately after a MOV or POP into a segment register. Therefore, when you single-step through a MOV or POP into a segment register, one or more instructions may appear to be skipped. The instructions actually execute; however, control does not return to RDT until two instructions past the MOV or POP instruction (or one instruction past if the MOV or POP is not itself a MOV or POP into a segment register). RDT places an asterisk on the trace buffer line to indicate the possible missing instruction(s) immediately following the marked instruction.

Here is a summary of some of the keys you use to control cursor motion and scrolling when the cursor is in the disassemble window area:

Right arrow	Moves the cursor right to the next available position into which keystrokes may be entered. When the cursor reaches the last valid input position of the "TB" input field, it moves no farther.
Left arrow	Moves the cursor left to a position into which keystrokes may be entered. When the cursor reaches the first valid input position of the "TB" input field, it moves no farther.
Tab	Moves the cursor to the beginning of the "TB" input field, which is the last trace buffer line displayed in the RDT trace windowing area.
Ctrl + Up arrow	Scrolls the RDT trace buffer display window up one trace buffer entry, to the limits of the RDT trace buffer.
Ctrl + Down arrow	Scrolls the RDT trace buffer display window down one trace buffer entry, to the limits of the RDT trace buffer.
Pg Up	Scrolls the RDT trace buffer display window up by the number of trace buffer entries displayed in the window, to the limits of the RDT trace buffer.
Pg Dn	Scrolls the RDT trace buffer display window down by the number of trace buffer entries displayed in the window, to the limits of the RDT trace buffer.

Math Co-Processor Windowing

Math Co-processor windowing (CW) lets you display the state of the Math Co-processor (if one is installed) in the display window area of the RDT display screen.

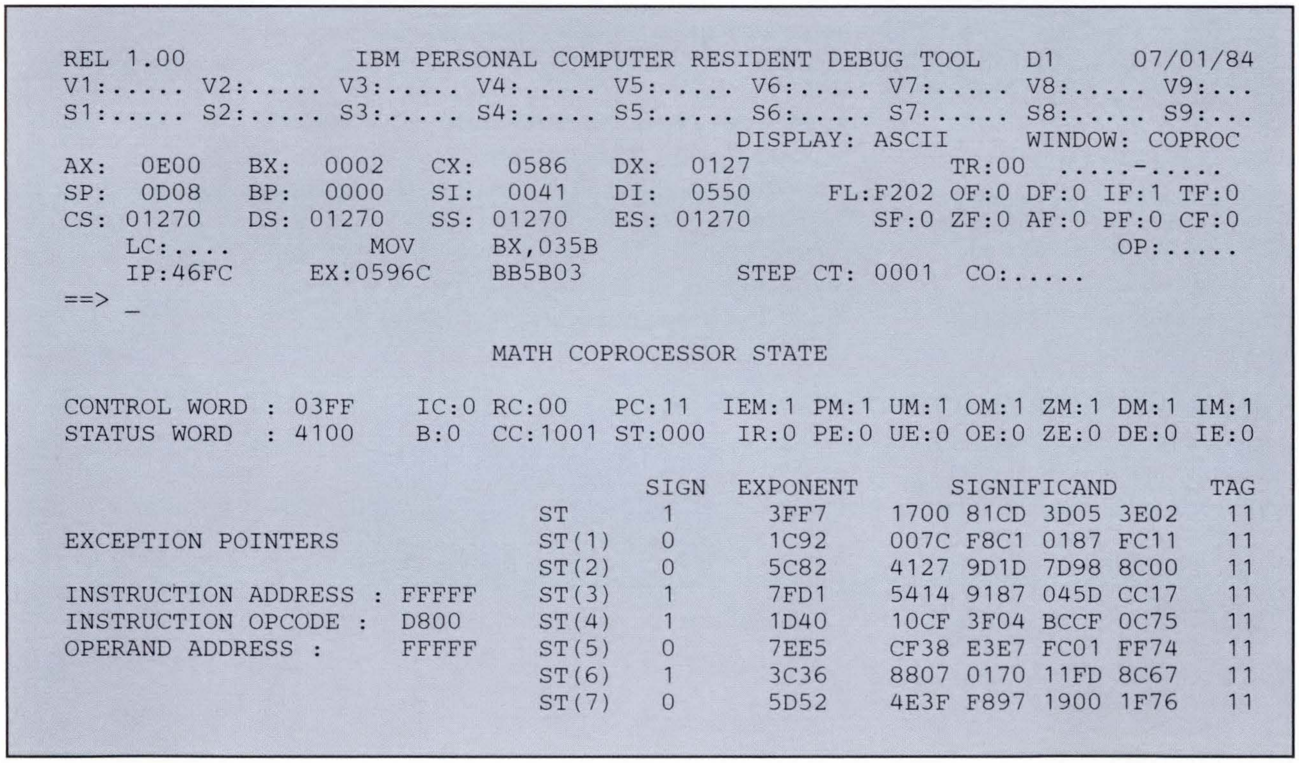
Figure 5 shows an RDT display screen in Math Co-processor window mode. The window area contains the current state of the Math Co-processor, including the control word, status word, exception pointers and register stack. The control word and status word are displayed in hexadecimal and binary formats. There are no values to modify for this window. (For more information on the content of the Math Co-processor control word, status word, exception pointers or register stack, refer to the *IBM Technical Reference* manual.)

RDT can distinguish a software or Math Co-processor NMI interrupt 2 from other types of NMI interrupts such as an NMI switch depression, memory parity error or I/O channel check. When an exception occurs and interrupts are enabled for the Math Co-processor, an Interrupt 2 (NMI interrupt) occurs on the main processor (8088 or 80286).

If the user has directed RDT to handle soft NMI interrupts, either by way of the "n" loadtime option or the "SN" (Set NMI) command, then RDT contains the logic required to handle the Math Co-processor exception. This may circumvent any exception handling routines you may have written.

If RDT is handling NMI interrupts, a Math Co-processor is installed, and a software or Math Co-processor exception NMI interrupt occurs, then RDT is immediately placed in Math Co-processor windowing mode. In this case, when the RDT display appears, the message line indicates the type of NMI interrupt, and the Math Co-processor state is displayed. This lets you immediately see which Math Co-processor exception (if any) occurred, by checking the exception bits of the status word.

After RDT displays the state of the Math Co-processor, RDT clears the exception condition. Subsequent displays will show that the exception has been cleared. When you debug a Math Co-processor exception handler, do not instruct RDT to handle NMI interrupts; your exception handler will then be invoked when an NMI occurs.



```
REL 1.00          IBM PERSONAL COMPUTER RESIDENT DEBUG TOOL    D1      07/01/84
V1:..... V2:..... V3:..... V4:..... V5:..... V6:..... V7:..... V8:..... V9:...
S1:..... S2:..... S3:..... S4:..... S5:..... S6:..... S7:..... S8:..... S9:...
                                            DISPLAY: ASCII     WINDOW: COPROC
AX:  0E00   BX:  0002   CX:  0586   DX:  0127           TR:00  .....-.....
SP:  0D08   BP:  0000   SI:  0041   DI:  0550     FL:F202 OF:0 DF:0 IF:1 TF:0
CS: 01270   DS: 01270   SS: 01270   ES: 01270        SF:0 ZF:0 AF:0 PF:0 CF:0
    LC:....        MOV     BX,035B                                   OP:.....
    IP:46FC    EX:0596C    BB5B03           STEP CT: 0001  CO:.....
==> _

                                MATH COPROCESSOR STATE

CONTROL WORD : 03FF     IC:0 RC:00   PC:11   IEM:1 PM:1 UM:1 OM:1 ZM:1 DM:1 IM:1
STATUS WORD  : 4100     B:0  CC:1001 ST:000   IR:0 PE:0 UE:0 OE:0 ZE:0 DE:0 IE:0

                                       SIGN  EXPONENT      SIGNIFICAND       TAG
                                ST      1      3FF7     1700 81CD 3D05 3E02   11
EXCEPTION POINTERS              ST(1)   0      1C92     007C F8C1 0187 FC11   11
                                ST(2)   0      5C82     4127 9D1D 7D98 8C00   11
INSTRUCTION ADDRESS : FFFFF     ST(3)   1      7FD1     5414 9187 045D CC17   11
INSTRUCTION OPCODE :  D800      ST(4)   1      1D40     10CF 3F04 BCCF 0C75   11
OPERAND ADDRESS :     FFFFF     ST(5)   0      7EE5     CF38 E3E7 FC01 FF74   11
                                ST(6)   1      3C36     8807 0170 11FD 8C67   11
                                ST(7)   0      5D52     4E3F F897 1900 1F76   11
```

Figure 5. Sample RDT Math Co-Processor Window Screen

Jump Tables in Assembly Language

Bill Claff
Boston Computer Society IBM PC Users' Group

If you have a program with several options that are selected by pressing a single key, your program must determine which key is pressed and then jump to an appropriate section of code. If you use INT 16 for retrieving a keystroke, AH will contain the code. Assuming that you are going to process the cursor keypad (i.e., Home key, arrow key, PgUp key, etc.), the brute force approach is to do many comparisons and jumps as shown in Figure 1.

```
     MOV  AH,0       ;clear AH
     INT  16         ;get new keycode
                     ;place it in AH
     CMP  AH,047H    ;is it Home key
     JNE  K2         ;
     ...  your code for Home key
     ...  jump to getting next key
K2   CMP  AH,048H    ;is it arrow key
     JNE  K3
     ...  your code for arrow key
     ...  jump to getting next key
     etc.
```

Figure 1. Assembly Code for Keystroke Selection

There are several limitations to the above technique. The major restriction is that the Jump Not Equal (JNE) is limited to distances of 128 bytes. The second limitation is that the code is hard to patch. I suggest the technique shown in Figure 2.

As the first line of code indicates, use only the BX register as a holding register when using jump tables (MOV BL,AH). The second line of code (MOV BH,0) resets the register to 0. The third line (SUB BX,047H) starts the table with the Home and key and above. The next line (SHL BX,1) gets a word address by multiplying by 2, for example, if you are 4 keys away from the Home key, you need to jump 8 bytes into the jump table to get the correct address of the service routine for that key. The rest of the code should be self explanatory.

```
     MOV BL,AH
     MOV BH,0
     SUB BX,047H
     SHL BX,1

     JMP TBL[BX]
TBL  LABEL WORD
     DW    HOME_KEY
     DW    UP_ARROW
     etc.
HOME_KEY LABEL NEAR

     ...your handler here
     ...exit back to routine to get
        next keystroke

UP_ARROW LABEL NEAR

     ...your handler here
     ...exit back to routine to get
        next keystroke

     etc.
```

Figure 2. Alternate Code for Keystroke Selection

Also, there is a neat trick for processing the cursor keypad scan codes. Read the scan code, subtract 9, read the first two bits and store that result, and read the second two bits and store that result. Those two numbers determine the row and column of the key. Treating the two-bit numbers as twos-complement's signed arithmetic, they both take on the values of -1, 0 and 1. The keypad is then represented as:

	-1	0	1
-1	Home	up	PgUp
0	<--		-->
1	End	DN	PgDn

Review of Application Display Management System

Stan Fellers
Phoenix IBM PC User Group

If you have moved beyond writing programs in BASIC and understand the assembly level of programming on the IBM PC, you certainly understand the amount of time necessary to write programs in languages other than BASIC, and you understand the determination and commitment required to program screens that are complete and understandable as well as easily modifiable. If you fit in this category, IBM has a product for you—the Application Display Management System (ADMS). ADMS is a very sophisticated development package about which I have a great deal of good to say. Having programmed on an IBM 4341 using CICS and DL1, I was a bit surprised to find IBM has provided somewhat the same power for the PC.

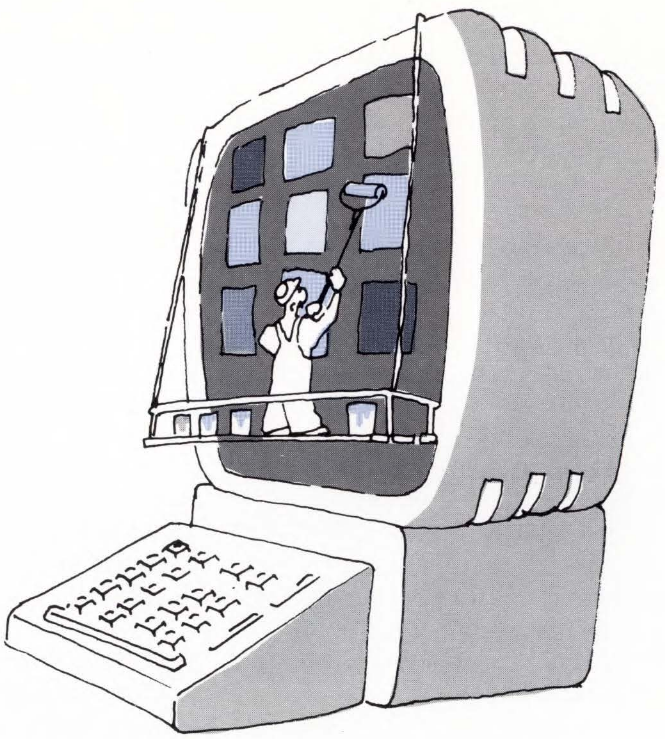
This program, actually a runtime program, allows you to paint screens, choose field definitions, choose colors, etc. You can also display the time on a dynamic clock and choose floating-point decimal notation.

One of the really powerful features of ADMS is that it can interface with the BASIC Interpreter, BASIC Compiler, COBOL, FORTRAN, Macro Assembler or IBM Pascal programs.

Like most other screen design programs, this one is menu operated, but extensive help screens are available if something doesn't seem clear.

All menu operations are chosen using the function keys. The function key assignments are displayed on the 25th screen line, removing your access to that line. One plus about this feature is that you can edit any field before pressing the function key you designate as the EXIT key. For error-prone operators, this can be a blessing.

You are first presented with a menu that requests the disk drive you want to retrieve screen information from or store information to. If you are unsure of



what to enter, you can press the F1 key to get an explanation of each requested field. You are also required to select a screen to edit or create along with an extension. The default extension is .SCR.

If you can't remember the screen name you want to edit, you have an option of viewing a directory of the default disk.

Next, a list of all screens connected with the "master screen" is displayed. This is a very comprehensive list, containing:

1. Amount of disk storage required for each screen
2. Amount of memory required to display the screen
3. Number of fields in each screen
4. Date and time of last update
5. A description of each screen (more on this later)

You can either choose one of the existing screens to edit or enter a new name. Either choice will present you with the next menu, which contains:

Text	allows you to lay out the screen's appearance.
Messages	describes the messages that will be available for this screen.
Fields	positions and defines the input, output and display fields. This definition includes minimum and maximum values, the

message number that will be displayed if the entered information is not in that range, and message number if nothing is entered in a required field.

Function Keys
gives a value to the function keys you want to use. All 40 keys are available for use. Function keys can be defined to allow switching between color and monochrome monitors.

Interrupt Keys
defines the keys that will return control to your program.

Highlighting
determines the colors that will be used for text, fields and messages. A total of 16 colors are available, plus blinking, underlining, and reverse image.

General Features
determines sound and border color. You are also given the option of entering a description for this screen that will display on the screen selection screen.

The Application Display Management System is easy to use and is consistent. It's a very useful and powerful product.

A brief explanation of field definition is in order. Besides the user defined fields, system variables are available. These include:

TIME	static time
TIMEU	dynamic time
DATE	system date from boot-up
NUML	status of Num Lock key
SCRL	status of Scroll Lock key
CAPL	status of Caps Lock key
CROW	cursor row
CCOL	cursor column
FTYP	cursor type (CHR, INT, FPT)

You can choose to protect any field by not allowing any input. This is necessary for displaying the time, date, etc. Available field types are:

C	any character
U	any character with conversion to uppercase
I	whole number, minus signs
Fn	floating point decimal with up to 4 decimal places
S	system variables mentioned above

The documentation gives an in-depth explanation of fields and would be worthwhile for any serious use of this product.

Since the screens created with the ADMS are external to your program, screens can be changed and edited without changing the program if fields are not deleted or added.

The program interface is created by defining and giving a value to specific fields and then "calling" ADMS. Control is then assumed by ADMS. When control is returned to your program, you must check the value of the completion code to determine if control was returned without an error. Data fields are returned as alpha characters and must be converted if numeric fields are needed for calculations.

For BASIC programmers, a facility on one of the disks can start your programs with all "include" files that are necessary when you decide to compile your program. That is true for all the supported languages.

The documentation contains enough information to get you started, with a section devoted to each of the supported languages.

If you have plans to market software created using ADMS, IBM has a stipulation that you register your package and purchase a distribution license for $100. This is a one-time charge.

The Application Display Management System is easy to use and is consistent. It's a very useful and powerful product.

Memory in the IBM Personal Computer

Milt Hull
Sacramento PC Users Group

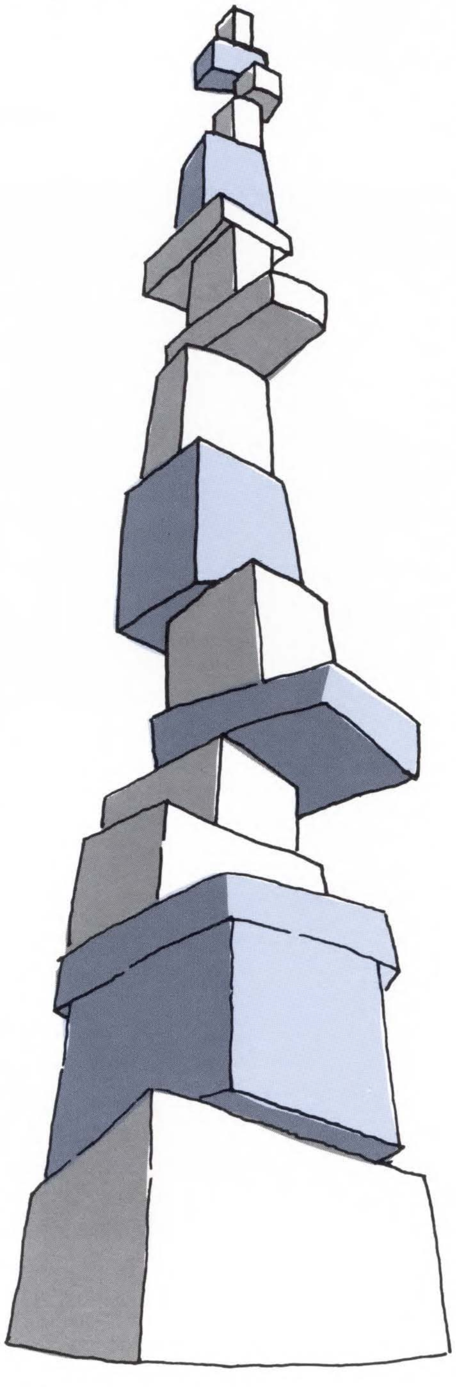
Computer memory is an internal part of the computer that stores information where the processor can directly access it. The heart of the IBM PC, the Intel 8088 microprocessor, has the capability of internally accessing a megabyte (MB) of memory (1,048,576 bytes).

Little Boxes

Think of the PC's memory as a million little boxes stacked on top of each other, each of them with eight lights mounted on the side. Each box can be thought of as a byte, and each of the eight lights on the box can be thought of as bits. The eight lights (bits) on each box can be either on or off in various combinations to represent the byte of information stored in the box. Thus each byte has eight bits; since a bit has the value of either being on or off, each byte has 256 possible combinations (that's 2 multiplied by itself 8 times). A kilobyte (KB) is 1024 bytes (where 1024 is 2 multiplied by itself 10 times), so 64KB is actually 65,536 bytes (rather than exactly 64,000 bytes).

Memory Allocation on the Personal Computer

Memory starts at location zero and goes to 1,048,576 (FFFFFH, where the H stands for hexadecimal). Each box or byte has a number assigned to it, like an address on a mailbox. The memory segments discussed below are also shown on the chart in Figure 1.



Random Access Memory (RAM): This portion of memory ranges from zero to 655,360 (A0000H), which is the limit of the disk operating system (DOS). The first part of this RAM is used by DOS itself for interrupt vectors, data for the basic input and output (BIOS) control routines, and working storage. The rest of the RAM (up to 640KB, depending on how much you have installed) is available for your programs to read and write.

Display Memory: After A0000H comes a 16KB block of memory that is not used; it ends at location A4000H, where the display memory begins. From location A4000H to location C0000H, memory is reserved for the displays. The displays actually use only a small portion of this memory. The monochrome uses 4KB to drive its screen, while the Color/Graphics Display uses 16KB. Even if you have both a color display and a monochrome display, 92KB of this reserved memory is left unused. It appears that IBM left room for expansion without requiring changes to the PC. (The IBM Enhanced Graphics Adapter can use more, perhaps one reason why so much memory was set aside for the display.)

Read-Only Memory: The read only memory (ROM) begins at location C0000H and goes to the end, location FFFFFH. However, little is used until location F6000H, which is where the first of four BASIC ROM chips begins. The four BASIC ROM chips are internal to the machine, and each takes up 8KB. These chips contain the cassette BASIC that appears when you start your computer without a system disk. When you invoke BASIC from a disk after loading DOS, it just enhances the ROM BASIC with functions not already contained in the chips.

After the memory reserved for the four BASIC ROM chips comes the last ROM chip, in the last memory location: the Basic Input and Output System (BIOS) chip. Since each of these ROM chips is only eight kilobytes, the PC has a total of 40KB of ROM. Even at this memory address there is room for expansion, because the empty socket next to the BASIC ROMs on the system board can be used for the 8KB just before location F6000H.

Memory Addresses

But just how do you point at any one location in memory? The PC contains a 16-bit processor, so you could start with a 16-bit number such as "0000 0000 0000 0101" in binary or 0005H (remember hexadecimal?). This number points to the sixth byte because the PC starts counting from zero. The binary number "1111 1111 1111 1111" (or FFFFH) would point to the 65,536th box, which is the number that actually corresponds to the well-known 64KB. But this is as big as a 16-bit binary number can get, and it's nowhere near the millionth byte.

Segment Address: The designers of the PC microprocessor decided to use two 16-bit numbers to label memory locations. The first 16-bit number is multiplied by 16 (binary) which moves it to the left four places (it has four zero bits added to the end of it), so that it really represents a 20-bit number. This 20-bit number is called the

Segment Address			
Decimal	Hex	Function	
0 to 640KB	00000 to A0000		640KB Random Access Memory (RAM)
656KB	A4000		128KB Reserved for Displays
672KB	A8000		
688KB	AC000		
704KB	B0000	Monochrome Display	
720KB	B4000		
736KB	B8000	Color/Graphics Display	
752KB	BC000		
768KB	C0000		192KB Read Only Memory Expansion and Control
784KB	C4000		
800KB	C8000	Fixed Disk Control	
816KB	CC000		
832KB	D0000		
848KB	D4000		
864KB	D8000		
880KB	DC000		
896KB	E0000		
912KB	E4000		
928KB	E8000		
944KB	EC000		
960KB	F0000		64KB Base System ROM
976KB	F4000		
992KB	F8000	BASIC ROM	
1008KB	FC000	BIOS ROM	

Figure 1. Memory Table

segment address. Notice that multiplying 65,536 by 16 gives 1,048,576, a full megabyte. But the segment address has only zeros in its last four places (notice that if you write the segment address in hexadecimal, the last digit is always 0), so it can actually label every sixteenth byte. These sixteen byte segments are called paragraphs.

Relative Address
The segment address narrows the memory location down to a paragraph, but to find the exact memory location within the paragraph, a second 16-bit number, called the relative address, is added to the segment address. So the 20-bit number (segment address) is added to a 16-bit number (relative address) to get the exact location within the entire memory space. Here's an example in hexadecimal:

```
12340     segment address
 5678     relative address
179B8     actual location
            in memory
```

This is why, when you are in Debug, you get numbers that look like "1234:5678".

Use Debug to read the date of your BIOS chip. Load Debug and give the command "D F000:FFF5 L 8". This will display the contents of the eight bytes that start at segment address F000H and relative address FFF5H. You should recognize the result as a calendar date when it appears on your display screen.

PC Memory Organization

David Betts
San Francisco PC Users Group

The amount of usable memory in the IBM Personal Computer is limited to the amount the microprocessor can address. At the hardware level, this is set by the number of address lines or wires the chip's designers chose to use.

Each line represents one binary digit of the address; each wire either conducts (on) or does not conduct (off). This is represented in binary as either a 1 (on) or a 0 (off). Therefore, the more lines, the larger the binary address number that can be used. In the case of the 8088/6 chips, this is 20 lines, or 2 raised to the 20th power, which is 1,048,576 bytes or 1 megabyte (MB).

Imagine yourself as a postman with the job of putting a letter in a specific one of 1,048,576 different pigeon-holes! To divide this chore into manageable chunks, memory is divided into "pages" or segments. A memory page contains 64 kilobytes (KB), or 65,536 bytes, and there are 16 pages within the 1MB address space. Each page is further divided into four "paragraphs" of 16KB per paragraph.

Segment Addressing

To read or write the contents of a particular byte of RAM, the microprocessor has to calculate its address. That is, while the set of address lines—the address "bus"—is 20 bits wide, the arithmetic registers inside the 8086/8 are only 16 bits wide. Thus, the scheme is to use two registers added together in a special way. These registers always include one of either the code, data, stack or extra segment registers (CS, DS, SS, or ES registers) plus any one of the other registers.

These registers are combined by multiplying the segment register by 16 (binary) or "left shifting" it one hexadecimal place, then adding it to the contents of the other register, commonly called the offset. This produces the five-digit hex number or a 20-bit binary number which is a byte's absolute address.

Visualize twenty boxes in a row. The segment part of the address first fills the rightmost 16 boxes with either ones or zeros. Then the contents of the boxes are shifted left four places so that the right four boxes are empty. This is what the segment part of the address looks like. Its least significant digit is in the fifth place in the boxes. Incrementing or decrementing that digit is like counting by 16s.

Then in comes the offset. It's also 16 bits wide. It is put into the boxes like the segment, except each digit is added to the existing contents of each box according to the rules of binary addition.

The 640KB Limit to RAM

This method of segment addressing can point to each byte of the PC's full megabyte. So why is there a 640KB limit to random access memory? Where is the rest?

The answer is that the PC reserves sections or blocks of this memory space for certain dedicated functions. The IBM read-only memory (ROM) BIOS is located in a specific location. So is the data used for the screen display. That's not to say you can't access these blocks of memory; you just can't store programs in them and expect the disk operating system to function properly or the programs to work. If data or program information is loaded into the section of memory reserved for the screen display, the monitor instantly reflects the intrusion.

The simple program in Figure 1 illustrates how to access various memory locations within the computer. This program loads the full IBM character set from ROM into the display memory location. The program shown below is for a color monitor. If you want this to run on a monochrome monitor, change the first command to read MOV AX, B000.

This program will fill the visible part of video memory with the ASCII characters burned into the permanent storage (the ROM).

MOV AX, B800; MOV DS, AX
This moves the video segment address into the DS segment register. Video has its own memory page that begins at B800 (hexadecimal) for color, and B000 for monochrome.

MOV AX, 0000; MOV SI, 0002; MOV CX, 07D0
The SI (source index) register will be used as an offset into the video segment. It will iterate just enough times to fill all the visible page, or 7D0 (hex) times.

INC AX; ADD SI, 0002
This pair of instructions increments the character to print (AX) and the next address to put it in (SI).

Note that SI is incremented by two each time. Why? A byte in video memory that holds a character to print is followed by a byte that holds the information about how the character is to be printed. The character attribute bytes are odd and the character bytes are even.

```
                     ; CP—This pro-
                     ; gram prints all
                     ; pc characters on
                     ; a color display
                     ;  in various
                     ;  colors.
                     ; This program may
                     ; be used freely.
                     ;
     MOV AX, B800    ; move the video
     MOV DS, AX      ; seg addr into DS
     MOV AX, 0000    ; initialize AX
     MOV SI, 0002    ; initialize SI
     MOV CX, 07D0    ; initialize CX
                     ; set loop counter
010E INC AX          ; increment AX
     MOV [SI], AX    ; move the char
     ADD SI, 0002    ; point SI at
                     ; next addr
    LOOP 10E         ; decrement CX,
                     ; check for zero
     INT 20          ; end gracefully
                     ; return to DOS
                     ;
                     ; machine code:
                     ; B8 00 B0 8E D8
                     ; B8 00 00 BE 02
                     ; 00 B9 D0 08 40
                     ; 89 04 83 C6 02
                     ; E2 F8 CD 20
```

Figure 1. Program to Display IBM Character Set

LOOP 10E
This is the loop action. The CX register is decremented by 1. When the CX register is zero, the program moves to the next command. When CX is not zero, the program jumps back to 010E.

INT 20
The INT 20 terminates the program (and returns to DOS).

Assemble this program using Debug, either with the (A)ssemble command or with (E)nter command using the machine code above. Remember to save the program before you run it.

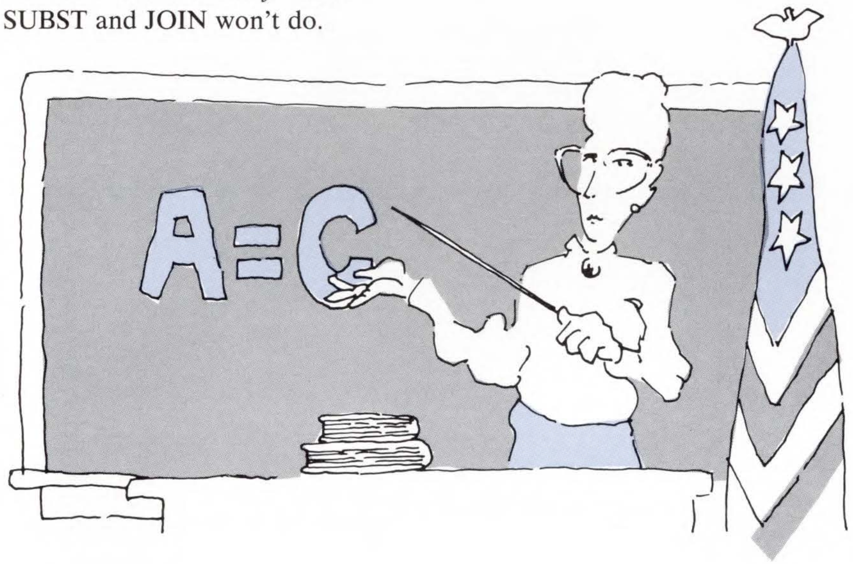
You can experiment by putting different values into the AX register. The top byte of the AX register ends up in the attribute byte mentioned above, so if you want to examine some funny effects, change the line above to read MOV AX, 0100 or 0200, etc. Have fun.

Use Care With DOS ASSIGN Command

Ralph Keuler
Pacific Northwest IBM PC Users Group

The ASSIGN command in DOS 3.00 and 3.10 reroutes all disk input/output requests for one drive to a different drive. This allows you to run programs from your fixed disk or from a virtual disk that would otherwise run only in drive A or B (reASSIGNing does not apply to copy-protected software). Software designed to perform all disk operations only on drives A or B can be redirected to the specified drive.

However, you should use the ASSIGN command *only* when SUBST and JOIN won't do.



During normal operations, the computer gives no indication that a drive has been reassigned, and any filing or copying operations done to a reassigned drive will actually be done on the drive it has been reassigned to. The warning in the *DOS Reference* manual states that the BACKUP, RESTORE, LABEL, JOIN, SUBST, or PRINT commands may cause problems if used when ASSIGN is active.

For example, if drive A has been assigned to C, and drive A is the target disk when you invoke BACKUP, BACKUP will *erase the directory of your fixed disk* and replace it with entries of backed-up versions of your fixed-disk files. Also, normal commands like COPY C:\MYFILE.TXT A: become hazardous when drive A is reassigned to C. DOS will write a copy of MYFILE.TXT, not on A, but right over itself on C, and it may scramble the File Allocation Table, as well.

To safely ASSIGN one drive to another, use BATch files to call all programs that require the ASSIGN command. The BATch file will set the proper ASSIGN parameters and then reset them when the program exits. For example:

```
ASSIGN A=C
DATABASE
ASSIGN
```

The ASSIGN command entered without parameters resets ASSIGNed drives to their normal values and avoids potential problems.

To be completely safe, use BATch files to reset any ASSIGNed drives to the default values before you invoke other DOS commands that might create problems if invoked while a drive is reassigned. For instance, use a BATch file to invoke the BACKUP procedure. This can be done as follows: 1) Rename the BACKUP.COM file to BKUP.COM; 2) Create and run the following BATch file named BACKUP.BAT:

```
ASSIGN
BKUP.COM %1 %2 %3 %4
```

Disabling Call-Waiting

Bob Hutchinson
Lilly Computer Club

The problem with having both a modem and call-waiting is that when your modem is on line, the beep that announces a second call interrupts the carrier tone long enough to cause your modem to break the connection. There are at least three ways to deal with this problem.

Flexible Call-Waiting

In many areas the phone companies have activated an enhancement to the call-waiting feature that allows you to temporarily turn it off before placing a call. To suspend call-waiting on touch-tone phones, dial *70; for rotary phones, dial 1170. You will hear a double beep followed by the dial tone. Then just dial the number you want to call. Call-waiting will be suspended until you hang up. You can incorporate this in an auto-dial sequence so that you will not be interrupted while on line.

This feature is part of the conversion the phone companies are making to the "equal-access/1-plus" service for alternate long-distance carriers. It may not yet be available in your area, even by special order. Check with your local phone company to see if your exchange has installed this feature. If not, you may want to try one of the other ways to work around call-waiting.

Extend the Carrier-Loss Detection Time

This method works, but unless you have good communications software that checks the data transmission, it won't be satisfactory.

Most communications software packages allow you to increase the amount of time the carrier can be lost without breaking the connection. If you set this value to a duration longer than that of the beep, the connection will be maintained. Unfortunately, unless you are using an error-free protocol, you will experience some loss of data and the addition of extraneous noise characters in your data because of the beep.

Call-Forwarding

If you also have the call-forwarding feature, you can put your phone on call-forwarding to another number (time of day, weather, your office, your mother-in-law), and all incoming calls will automatically be forwarded without causing a beep on your line.

While your phone is on call-forwarding, it will attempt to produce a short ring to indicate that call-forwarding was in effect when the call came through. This ring is handled differently from the call-waiting beep and will not cause a problem. Don't forget to deactivate call-forwarding when you're through, or you won't receive your calls.

Easy End-of-File Marker

Philip Mayes
Santa Barbara IBM PC UG

In the July issue of *Exchange*, Sig Rosenthal describes how various software applications often have difficulty reading files created using the redirect features of DOS 3.00 and 3.10, as well as files created using the DOS COPY command. He correctly identifies the problem as stemming from the file's not having an end-of-file marker. However, there is an easier way to add the end-of-file marker than having to use DOS Debug.

The /a parameter of the DOS COPY command adds the end-of-file marker to any file copied.

For example, to add an end-of-file marker to the ANYNAME file, type the following:

```
COPY ANYNAME TEMPFILE /A
ERASE ANYNAME
RENAME TEMPFILE ANYNAME
```

The new ANYNAME file will have the end-of-file marker added.

Memory Intensive Programs on the PCjr

Jack Spitznagel
Boston Computer Society IBM PC Users' Group

If I needed a personal productivity setup for my home today, I would still buy a PC*jr*, as much memory as I could add to it, and a copy of the program I needed.

Why? Because with only a slight modification to the IBMDOS.COM file, you can run large, memory-intensive programs like *Symphony*, the program I am currently running on my PC*jr*. It runs well on my PC*jr* and it saves having to buy several programs of moderate expense to do all the same tasks, not to mention the time it takes to learn unfamiliar software.

Modification to IBMDOS.COM

I was mystified at first by the apparent inability of the PC*jr* to run Symphony. I had my PC*jr* set up with 640KB of RAM on a multifunction card, and there was no logical reason for the random problems I was experiencing. The program seemed to lockup periodically during COPY, MOVE, and ERASE command procedures.

I had almost given up when I noticed an article by Don Awalt in the November, 1984 *PC Tech Journal*. I realized that the IBM-supplied patch reproduced there in that article would cure the problem I was having.

Unlike the other PCs, the PC*jr* uses the non-maskable interrupt (NMI) to check its keyboard for input. Apparently, IBMDOS.COM (one of the two hidden disk operating system files) does not properly restore the segment register and stack pointer after encountering a non-maskable interrupt (NMI). The problem often causes the PC*jr* keyboard to lock, and the computer must be restarted to recover from the lockup. By using Debug to modify the IBMDOS.COM file as shown in Figure 1, you can solve the problem.

The following procedure assumes that a DISKCOPY of the DOS 2.10 system diskette is in drive A. Enter the commands shown in normal type. Debug's responses are shown in blue type. If this is to work reliably, you must first enter the Debug corrections, then transfer the modified operating system files to your other system diskettes using the SYS command.

```
A>DEBUG
-L 100 0 5 1
-D 12B L4
XXXX:012B 27 00 00 00
-E 12B
XXXX:012B 27.20
-W 100 0 5 1
-Q

A>DEBUG IBMDOS.COM
-UCAC L A
XXXX:03AC      CS:
XXXX:03AD      MOV SP,[02A6]
XXXX:03B1      CS:
XXXX:03B2      MOV SS,[02A8]
-A3AC
XXXX:03AC      CS:MOV SS,[02A8]
XXXX:03B1      CS:MOV SP,[02A6]
XXXX:03B6      [ENTER]
-UCD1 L A
XXXX:0CD1      CS:
XXXX:0CD2      MOV SP,[02D1]
XXXX:0CD6      CS:
XXXX:0CD7      MOV SS,[02D3]
-ACD1
XXXX:0CD1      CS:MOV SS,[02D3]
XXXX:0CD6      CS:MOV SP,[02D1]
XXXX:0CDB      [ENTER]
-U1522 L A
XXXX:1522      SS:
XXXX:1523      MOV SP,[02D1]
XXXX:1527      SS:
XXXX:1528      MOV SS,[02D3]
-A 1522
XXXX:1522      CS:MOV SS,[02D3]
XXXX:1527      CS:MOV SP,[02D1]
```

Figure 1. Modification for DOS 2.10

```
XXXX:152C      [ENTER]
-U311D L 8
XXXX:311D      MOV SP,[02D1]
XXXX:3121      MOV SS,[02D3]
-A311D
XXXX:311D      MOV SS,[02D3]
XXXX:D121      MOV SP,[02D1]
XXXX:3125      [ENTER]
-U325F L 4
XXXX:325F      MOV SP,ES
XXXX:3261      MOV SS,SP
-A325F
XXXX:325F      MOV BP,ES
XXXX:3261      MOV SS,BP
XXXX:3263      [ENTER]
-U409B L 7
XXXX:409B      MOV SP,4325
XXXX:409E      MOV AX,CS
XXXX:40A0      MOV SS,AX
-A409B
XXXX:409B      MOV AX,CS
XXXX:409D      MOV SS,AX
XXXX:40A2      MOV SP,4235
XXXX:40A5      [ENTER]
-W
WRITING 4280 BYTES
-Q

A>DEBUG
-L 100 0 5 1
-D 12B L 4
XXXX:012B 20 00 00 00
-E 12B
XXXX:012B      20.27
-W 100 0 5 1
-Q
```

Figure 1. Modification for DOS 2.10 (cont.)

Note: The segment of code marked XXXX:409B may show some variation from the addresses shown after the first one (XXXX:409B). I experienced this and the result was just fine.

Save this diskette and use it to format all disks in the future. You can also use it with the SYS command to transfer the modified system files to your other system diskettes.

Symphony runs well with my configuration, but it also works well using the IBM memory boards with at least 512KB of memory. Be sure to use the technique described in the notes on the enhancement disk. In brief, you will have to create a CONFIG.SYS file with the following line or add the following line to your existing CONFIG.SYS file:

```
DEVICE=PCJRMEM.COM
```

Then add the appropriate commands to your AUTOEXEC.BAT file to have your program load and execute properly.

Be sure your program diskette contains all of the files your BATch file executes. Also, if you use a Personal Computer AT to set this up, *do not use the high-capacity drive*. The PC*jr* may not properly read diskettes written using the high-capacity drives.

Setting Environment Space

Dave Hoagland
Lawrence Livermore National Laboratory

Editor's note: The method of setting the memory reserved for environment information described in this article is **not documented** *in the* DOS Reference *manual and is* **not supported** *by IBM. Because this method is undocumented and unsupported, it is subject to change or removal in the future.*

When you increase the number of subdirectories on your fixed disk, path statements tend to grow ever longer. These, along with prompt definitions and other SET commands, all occupy "environment space."

One of the features of DOS 3.00 and 3.10 is that the environment space automatically increases to meet requirements until you load a memory-resident program. Unfortunately, DOS considers the AUTOEXEC.BAT file to be a memory-resident program.

Inside DOS BACKUP

Carrington Dixon
Central Texas IBM PC Users Group

When you back up your disk files using DOS BACKUP, the program creates a BACKUPID.@@@ file on each backup disk and also puts a header on each file written to the backup disk. This ID file and file header enable the system to keep track of which files were backed up, when they were backed up, and which directory to RESTORE them to. Until you restore the back up copy of your files, this header remains on each file, preventing you from using the files as you would if you had made a copy of the file using the DOS COPY or DISKCOPY command.

Each backup disk contains a 128-byte file named BACKUPID.@@@. Figure 1 contains the format for the BACKUPID.@@@ file.

Figure 2 shows the format of the 128-byte header that DOS BACKUP places on each file.

If you are using DOS 3.00, the BACKUP command will occasionally fail to update the BACKUPID.@@@ file on one or more of the backup diskettes. The most common symptom (perhaps the only one) is for BACKUPID.@@@ to appear as zero bytes in the directory when you enter the DOS DIRectory command. When this happens RESTORE is not able to open the file to read the diskette number and therefore cannot continue. RESTORE will reject the diskette and prompt you to

Byte	Value	Use
00	00/FF	Indicates whether or not this is the last diskette of the backup group.
01-02	nn	Diskette number in low byte, high byte, decimal format (e.g. "10" is stored as 00 01).
03-04	nnn	Full year in low byte, high byte format
05	1-31	Day of the month
06	1-12	Month of the year
07-0A	nnnn	Standard DOS time if /T parameter used
0B-7F	00	Not used

Figure 1. The BACKUPID.@@@ File

Byte	Value	Use
00H	00/FF	Indicates whether or not this is the last diskette on which this file resides
01-02H	nn	Diskette number
03-04H	00	not used
05-52H	nn	Full Filespec except for the drive designator
53H	nn	Length of the filespec + 1
54-7FH	00	not used

Figure 2. The 128-Byte File Header

And many of the statements that use this space are included in the AUTOEXEC.BAT file. By then, it's too late—the size of the environment space is frozen, usually at 127 bytes (characters).

Many computer users have discovered that 127 bytes is not sufficient for all statements that occupy the environment space. I have been greeted with the error message "Out of environment space" too often for my liking.

The DOS SHELL command provides a simple and elegant solution. I have tried this successfully with DOS 3.10, but it may not work in other versions of DOS. Include the following statement in your CONFIG.SYS file:

```
SHELL=C:\COMMAND.COM /P/E:nn
```

where C:\COMMAND.COM tells DOS to load the command processor; /P makes COMMAND.COM permanent in memory, and /E:nn specifies the number of 16 byte paragraphs reserved for the environment space. The value of nn can be any value between 10 (160 bytes) and 62 (992 bytes).

insert the correct one. At this point, there is nothing you can do but Ctrl-Break out of RESTORE with only some of your files restored.

IBM has fixed the problem and has issued a revised BACKUP.COM that has been available from any IBM Authorized Personal Computer Dealer since fourth quarter 1984. The best way to avoid this problem is to upgrade to DOS 3.10. However, getting an updated version of BACKUP.COM does not help much *after* you notice that a BACKUPID.@@@ file, done under DOS 3.00, shows 0 bytes. However, you *can* repair the faulty ID file.

Even though the directory entry for the BACKUPID.@@@ file reads 0 bytes, the complete BACKUPID.@@@ file is really out there on the disk. All you have to do is fill in the information missing in the directory. This is not as formidable as it might sound, because every BACKUPID.@@@ files is the same size, and each is in the same place on all diskettes. You can correct the file with any program that lets you read and modify specific sectors on the disk (e.g., DOS Debug, *IBM Professional Debug Facility*, Norton Utilities, etc). Take a look at the entry for BACKUPID.@@@ on one of the diskettes that shows a normal 128 bytes for its BACKUPID.@@@ file. You will see the information displayed as follows:

```
4241434B 55504944 40404020 00000000
00000000 0000DA60 7A0B0200 80000000
```

The incorrect directory listing will look like this:

```
4241434B 55504944 40404020 00000000
00000000 00000861 7A0B0000 00000000
```

Notice that there are two numbers in the last two groups of the correct entry that are zeros in the incorrect one. The "02" is the starting cluster number and the "80" is the file size in bytes. On every diskette backed up with dos 3.00, the size of the BACKUPID.@@@ file is *always* 80 hexadecimal or 128 decimal bytes, and the starting cluster is always 02. The other numbers that differ between the two entries are the time and date the respective BACKUPID.@@@ files were created. Obviously, the time and date entries do not have to be changed.

If you have the Norton Utilities or the Disk Repair utility of IBM's *Professional Debug Facility*, you should have no problem changing the faulty directory entry. However, since DOS Debug is more unwieldy

```
A>debug
-l ds:100 1 5 6
-d ds:100

5EDF:0100  42 41 43 4B 55 50 20 44-49 53 4B 28 00 00 00 00   BACKUP DISK(....
5EDF:0110  00 00 00 00 00 00 72 5F-7A 0B 00 00 00 00 00 00   ......r_z.......
5EDF:0120  42 41 43 4B 55 50 49 44-40 40 40 20 00 00 00 00   BACKUPID@@@ ....
5EDF:0130  00 00 00 00 00 00 08 61-7A 0B 00 00 00 00 00 00   ......a.r.......
5EDF:0140  43 48 49 4D 45 4E 45 20-53 43 52 20 00 00 00 00   CHIMENE SCR ....
5EDF:0150  00 00 00 00 00 00 EB 05-70 08 03 00 A9 07 00 00   ......k.p...)...
5EDF:0160  50 49 4E 54 45 20 20 20-53 43 52 20 00 00 00 00   PINTE   SCR ....
5EDF:0170  00 00 00 00 00 00 D4 09-70 08 05 00 13 13 00 00   ......T.p.......
The first two lines contain the diskette label, BACKUP DISK.
If your diskette does not have a label, the BACKUPID.@@@ file would
appear on the first two lines.

In the example above, the offset address 013A is the beginning cluster.
It reads 00 and needs to be changed to 02.  The address 013C reads 00
and needs to be changed to 80 using the Debug Enter Command:
-e 013a
5EDF:013A  00.02   00.00   00.80
Before you quit Debug, you must write your changes to the disk using the
Debug Write command:
-w ds:100 1 5 6
-q
A>
```

Figure 3. Debug Instructions for Changing Faulty BACKUPID.@@@ File

than the other debugging utilities, the instructions shown in Figure 3 will allow you to change the faulty directory entry using DOS Debug. Enter the code shown in black type. The blue type indicates Debug's response. The comments interspersed between the instuctions have been added for better understanding. A copy of DEBUG.COM must be on your disk in drive A, and the disk with the faulty BACKUPID.@@@ file entry must be in drive B.

Once you finish correcting the directory entry, the only thing left to do is check the data in BACKUPID.@@@ itself. The first byte of the file will probably be "FF". The FF is correct *only* if this is the last diskette of the backup set; all other diskettes would have "00" in their first byte (see Figure 4 on page 41). When a BACKUPID.@@@ file entry has the "zero length" problem, it usually has an "FF" in its first byte as well.

To check the file using DOS Debug, enter Debug and the file name:

That's all there is to it. Once you have corrected the directory and checked the first two bytes of the BACKUPID.@@@ file, you may want to see if RESTORE.COM will accept the disk by "restoring" some non-existent file off the backup:

RESTORE A: C:\SUBDIR\ANYFILE.TXT

If the only error message displayed is "file not restored," then your backup disk is in good shape. A more demanding check would be to rename your hard disk files that are backed up across two diskettes (especially those that had "problems") restore them and then compare the restored and renamed files with the DOS COMP command.

Alternate Method

Another method is to (1) load a good BACKUPID.@@@ file (using DEBUG) from the diskette that preceded the faulty one, (2) use the Enter command to change the second byte of the file to the correct diskette number (see Figure 2 on page 39), then (3) write the corrected BACKUPID.@@@ file to the diskette that had the bad file.

When you're finished, see if you can get the fixed version of BACKUP.COM from your dealer or upgrade to DOS 3.10.

```
A>debug b:backupid.@@@
-d
5EDF:0100  FF 02 00 C1 07 12 0B 00-00 00 00 00 00 00 00 00   ...A............
5EDF:0110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
5EDF:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
5EDF:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
5EDF:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
5EDF:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
5EDF:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
5EDF:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
If this is the last diskette in backup set, leave the FF; if not, change
the FF to 00 using the Enter command:
-e 0100
5EDF:0100  FF.00
Then write your changes to the diskette:
-w 0100
Writing 0080 bytes
-q
A>
```

Figure 4. Instructions for Changing the First Byte of the BACKUPID.@@@ File

The Last Word

Hugh Christian
Athens PC User Group

I've been smiling quite a bit since I got my first IBM Personal Computer. It hooked me; I loved it. After experimenting with several PCs, I finally bought an IBM Personal Computer AT. The world of the hard disk is a whole new world—almost as big a step as my first computer. And it is wonderful.

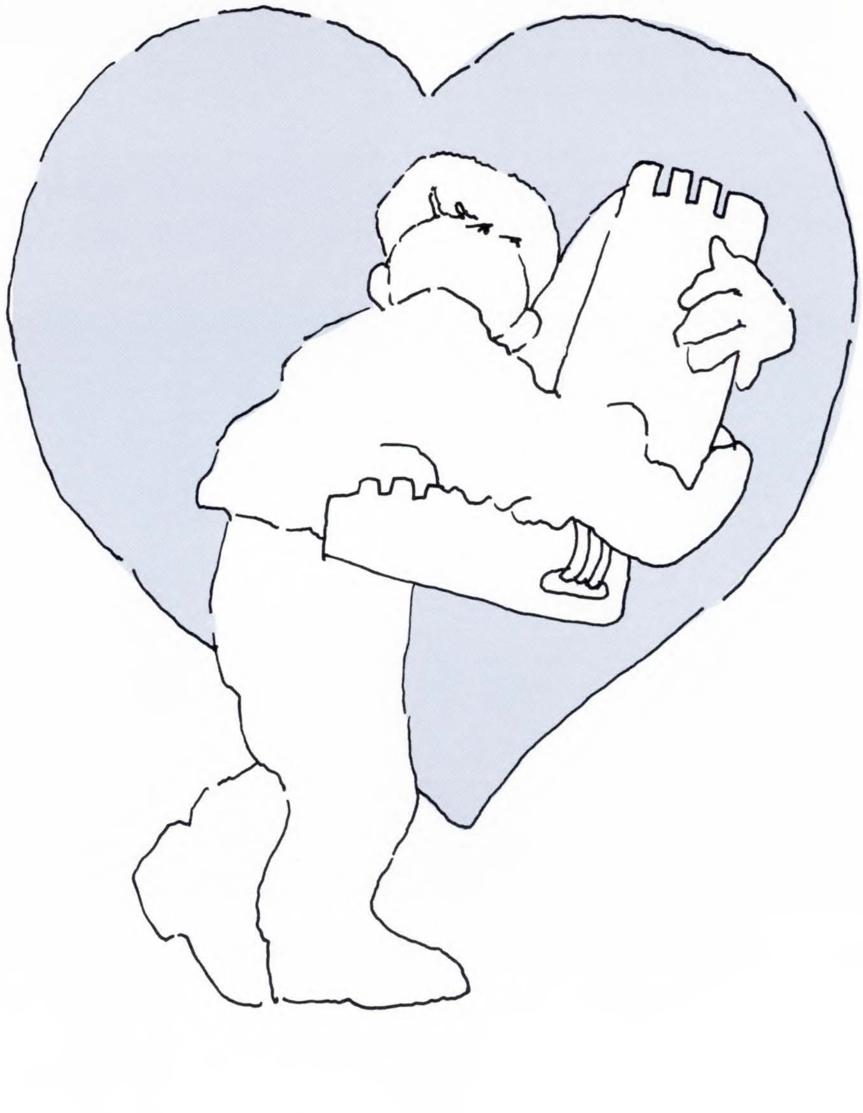
I dreamed about having access to a computer when I was just a little guy reading science fiction. All I dreamed about was having my own terminal. (At the time, owning a computer wasn't something you could even dream about.) And here, still in my youth, I've already owned six computers. It's even better than a dream come true, and it makes me happy whenever I think about it, and especially happy when I'm at the keyboard.

Now for the big question: *How do I justify spending $10,000 to have a computer balance my checkbook?* Well, I don't. In fact, I don't even try to use the computer to balance my checkbook. I have an accounting degree, and it takes only two minutes to do it by hand, so why bother the computer with it? It was meant for bigger and better things.

I don't understand all this concern with financially justifying a hobby. Does a fisherman ever figure out the cost per pound of catching, rather than buying, fish? In addition to the time he spends—which might be days—he should add up the costs of the boat, the trailer, the four-wheel-drive truck, the cabin at the lake, and the divorce settlement.

Hobbies aren't meant to be financially justified. The mistake that computers make is that they are useful at work. What should be a bonus is turned into a liability. I mean, the reason you never ask a fisherman to justify his fishing is because no matter how well he catches fish, no one asks him to bring his boat to work.

On the other hand, as soon as people know you have a computer, everyone is thinking of ways for you to use it—for work, not just for fun. I have found a few worthwhile uses for my computer, but I refuse to let them interfere with the enjoyment I get from playing with it. Yes, playing with it. I play with it just like I used to play with my electric train, and I have just as much fun.



New Products

Hardware

IBM 3812 Pageprinter

The IBM 3812 Pageprinter provides non-impact, letter-quality, sheet-fed printing for the IBM Personal Computer family. All-points-addressable 240 x 240 dots-per-inch output at 12 pages per minute (maximum) give the Pageprinter extensive graphics capabilities and text output at reasonable speed. Two paper cassettes (500 sheets and 300 sheets) support automatic sheet feeding for different paper types and sizes.

The 3812 Pageprinter emulates the basic functions of the IBM Personal Computer Graphics Printer. A *3812 PC Demo Diskette* shipped with each Pageprinter includes printer description tables to support DisplayWrite 3 versions 1.00 and 1.10. The diskette also includes a device driver to access programs written to the virtual device interface (VDI). The Pageprinter currently supports 61 different type fonts, including eight typographic fonts. Its easy-to-replace toner cartridge, photoconductor unit, developer unit, and fuser unit make maintaining the Pageprinter a simple task.

The Pageprinter attaches to a single Personal Computer through an asynchronous adapter, and supports up to eight Personal Computers with an optional Sharing Card.

IBM Wheelprinter E

The IBM Wheelprinter E is an impact printer that provides letter-quality output for the IBM Personal Computer family. Printing at speeds up to 16 characters per second, the Wheelprinter E supports four type pitches, multiple type styles, a 13.2 inch writing line and a drop-in ribbon cartridge. Wheelprinter E accepts paper 2-1/2 to 14-1/2 inches wide and 3 to 15 inches long. Options include a sheet feeder to handle 8-1/2 inch paper or a pinwheel feeder to handle continuous forms.

The IBM Wheelprinter E requires the IBM Personal Computer Printer Attachment Cable and the appropriate printer adapter feature.

IBM Token-Ring Network

The IBM Token-Ring Network is a high-speed communications network designed to connect IBM Personal Computers and other data processing equipment at a local site. The Token-Ring Network uses the IBM Cabling System for physical interconnection of up to 260 coaxial, twinaxial, loop, and Token-Ring Network devices; or type 3 specified telephone media for physical interconnection of 72 coaxial and Token-Ring Network devices. The network is a baseband Token-Ring Network that conforms to the IEEE 802.5 and IEEE 802.2 standards as well as the ECMA 89 standards.

IBM Token-Ring Network PC Adapter

The IBM Token-Ring Network PC Adapter is an IBM Personal Computer feature adapter containing a microprocessor that lets the adapter transmit and receive information at 4 million bits-per-second over the IBM Token-Ring Network. Adapter resident microcode controls the processor and performs simple diagnostics and error detection. The adapter comes with a diskette that includes a diagnostic program.

The Token-Ring Network PC Adapter is available for the IBM Personal Computer, IBM Personal Computer XT, IBM Portable Personal Computer, or IBM Personal Computer AT and must be attached to the IBM Cabling System with the optional IBM Token-Ring Network PC Adapter Cable, or to type 3 telephone media with the optional Type 3 Media filter.

To run on the Token-Ring Network, each Personal Computer requires DOS 3.10. Two application program interfaces are included. One supports the IEEE 802.2 data link control programming interface, the other supports the IEEE 802.2 direct physical control programming interface. Each requires 7KB of memory in addition to DOS and application program requirements.

Software

IBM Token-Ring Network NETBIOS Program

The IBM Token-Ring Network NETBIOS (Network Basic Input/Output System) Program provides a programming interface to let application programs run on both the IBM Token-Ring Network and the IBM PC Network. NETBIOS, the network control program for the IBM PC Network, allows communications programs like the IBM PC Network Program to access the IBM PC Network. The IBM Token-Ring Network NETBIOS Program lets programs written for the NETBIOS of the IBM PC Network Program run on the Token-Ring Network by translating NETBIOS functions into Token-Ring Network protocol requests.

The IBM Token-Ring Network NETBIOS Program requires an IBM Personal Computer, IBM Personal Computer XT, IBM Portable Personal Computer, or IBM Personal Computer AT; the IBM Token-Ring Network Personal Computer Adapter; one double-sided diskette drive; and DOS 3.10. The IBM Token-Ring Network NETBIOS Program requires 46KB of memory with an additional 7KB of memory for the network adapter handler program in addition to the memory requirements for DOS and any application programs.

The IBM Advanced Program to Program Communication for the IBM

Personal Computer (APPC/PC) lets an IBM Personal Computer support the Systems Network Architecture (SNA) programming interface for logical unit (LU) type 6.2, and physical unit (PU) type 2.1. These standards let programs communicate over the IBM Token-Ring Network and over synchronous data link control (SDLC) communication links with application programs running on other IBM computers.

The APPC/PC includes an application interface that supports two conversation modes, and has assembly language macros to support it. The APPC/PC supports multiple logical units, parallel sessions, security for user applications at session and conversation levels, and concurrent multiple logical links on the IBM Token-Ring Network and/or SDLC adapters. System configuration parameters are set through menus and stored in a configuration file.

The IBM Advanced Program to Program Communication for the IBM Personal Computer (APPC/PC) requires an IBM Personal Computer, IBM Personal Computer XT, IBM Portable Personal Computer, or IBM Personal Computer AT; one double-sided diskette drive; the IBM Token-Ring Network Personal Computer Adapter and/or SDLC Adapter; DOS 3.10; and 185KB memory for SDLC operation, or 195KB memory for Token-Ring Network operation, or 208KB memory for combined operation. The IBM Token-Ring Network Personal Computer Adapter handler program requires an additional 7KB of memory.

IBM Token-Ring Network/IBM PC Network Interconnect Program

The IBM Token-Ring Network/IBM PC Network Interconnect Program lets IBM Personal Computers attached to the IBM Token-Ring Network exchange information with IBM Personal Computers attached to the IBM PC Network. A dedicated IBM Personal Computer, running only the interconnect program, is attached to both networks to permit communications. Applications on either network must use NETBIOS communication protocols to communicate with each other.

The IBM Token-Ring Network/IBM PC Network Interconnect Program requires an IBM Personal Computer XT or IBM Personal Computer AT with 256KB memory, one double-sided diskette drive, an IBM Monochrome display or IBM Color Display and appropriate adapter, the IBM Token-Ring Network Personal Computer Adapter and adapter handler program, the IBM PC Network Adapter, and DOS 3.10.

IBM Asynchronous Communications Server Program

The IBM Asynchronous Communications Server Program lets IBM Personal Computers attached to the IBM PC Network or IBM Token-Ring Network share one or more network PC's acting as an ASCII communications server and its switched communications lines. The program can access the Rolm CBX II, a PBX (private branch exchange), or public switched network (by modem) and acquire data from information providers like the IBM Information Network, Dow Jones News/Retrieval Service, Compuserve, or others.

Individual Personal Computers do not require a telephone or modem connection to access dial-up applications. Each computer executes a user-supplied program that establishes a connection with the Asynchronous Communications Server Program. The communications request is then queued to the first available communications server on the network. Personal Computers acting as communications servers can still be used to run other application programs.

The Asynchronous Communications Server Program supports both digital and analog data switching systems, allowing personal computers to share and switch between ASCII applications (e.g., information services, ASCII hosts, and other Personal Computers). Each Personal Computer must have an asynchronous communications program that uses the IBM Asynchronous Communications Server Program protocol through the Network's basic input/output system (NETBIOS) interface.

Personal Computers running the IBM Asynchronous Communications Server Program require one or two identical communications ports. Each communications server must have a DCM, IPCI, IPCI/AT, or DTI to attach to the CBX II. The CBX II must be an 8004 or 9004 system at the current software level and have the CBX II Data Communication software feature installed for use with the DCM, IPCI, IPCI/AT, or DTI. These products are available through the Rolm Corporation.

Otherwise, the IBM Personal Computer or IBM Personal Computer XT require either the Asynchronous Communications Adapter, RS-232-C cable, and IBM 5841 Modem; or IBM Personal Computer 1200 BPS internal modem; and an analog telephone line. The IBM Personal Computer AT requires either a Serial/Parallel Adapter, serial 10 inch connector or serial 10 foot cable and IBM 5841 modem; or the IBM Personal Computer 1200 BPS internal modem; and an analog telephone line.

Each Personal Computer acting as a server must also have an IBM Monochrome Display, IBM Color Display, and appropriate adapter. In addition to DOS 3.10, each server requires the appropriate network support program (the IBM Token-Ring Network requires the Token-Ring Network NETBIOS program) and appropriate memory. The IBM Asynchronous Communications Server Program requires 160KB memory in addition to the previous requirements.

Copyrights, Trademarks, and Service Marks

"PCWATCH can expose a host of potential problems by showing precisely what a program is trying to do. (page 8)

"Memory windowing lets you perform memory display, alteration and scrolling functions. (page 19)

"The concept of indefinite windows has been implemented for memory patching. (page 20)

"Another powerful feature of the disassemble window is its ability to scroll down, left, or right. (page 23)

"The Application Display Management System is easy to use and is consistent. It's a very useful and powerful product. (page 30)

G320-0848-00