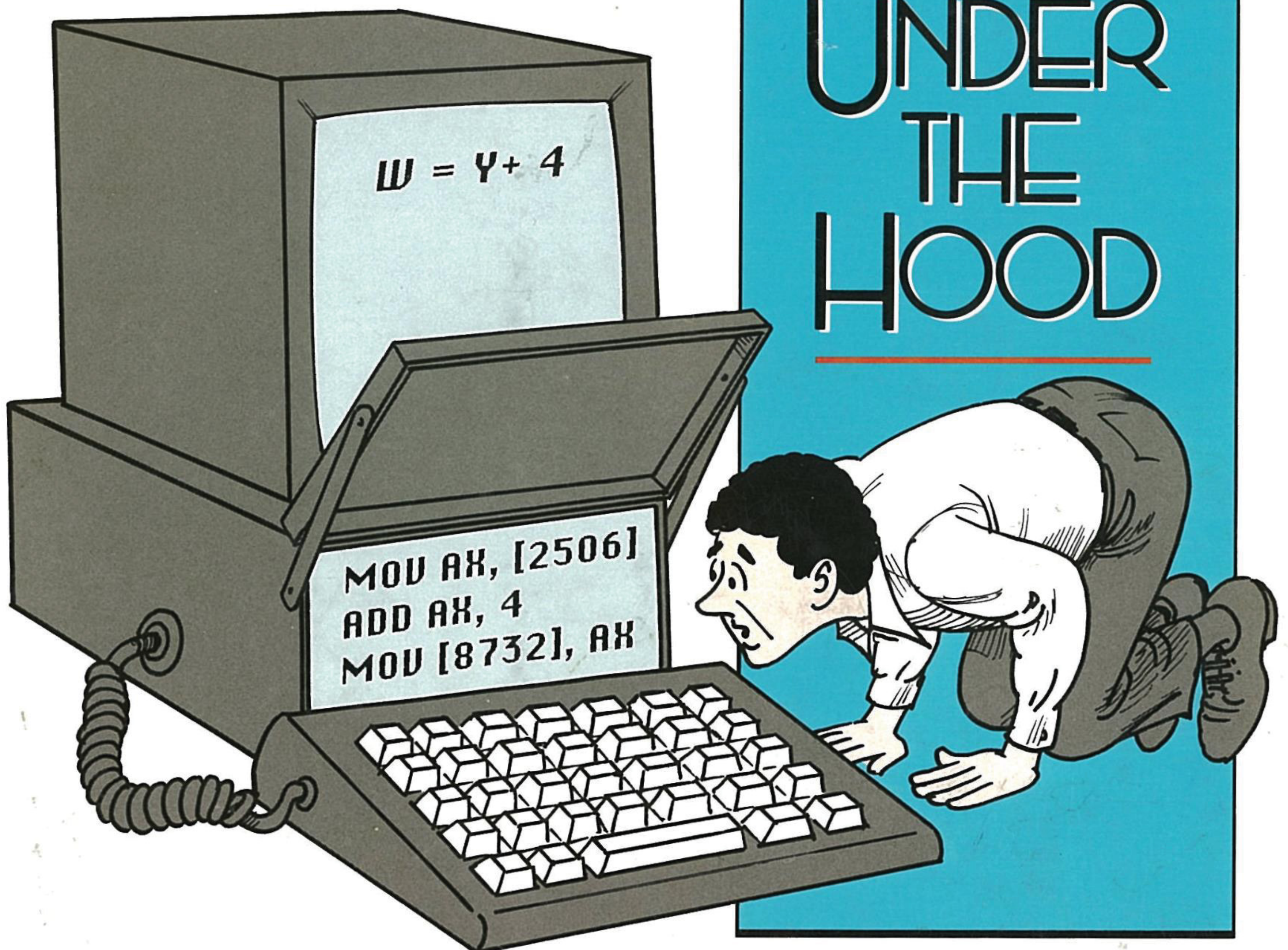
IBM® MICROCOMPUTER ARCHITECTURE AND ASSEMBLY LANGUAGE

A LOOK UNDER THE HOOD



NORMAN S. MATLOFF

ServiceNow's Exhibit No. 1013
IPR2015-00631

IBM® Microcomputer Architecture and Assembly Language

A Look Under the Hood

Norman S. Matloff

University of California, Davis

Prentice Hall
Englewood Cliffs, New Jersey 07632

ServiceNow's Exhibit No. 1013
IPR2015-00631

Library of Congress Cataloging-in-Publication Data

Matloff, Norman S.
IBM microcomputer architecture and assembly language : a look under the hood / Norman S. Matloff.
p. cm.
Includes index.
ISBN 0-13-451998-1
1. IBM microcomputers. 2. Computer architecture. 3. Assembler language (Computer program language) I. Title.
QA76.8.I259193M38 1992
004.2'565-dc20 91-27994
CIP

Acquisitions editor: *Marcia Horton*
Editorial/production supervision
and interior design: *Kathleen Schiaparelli*
Copy editor: *Brian Baker*
Cover design: *Lundgren Graphics*
Prepress buyer: *Linda Behrens*
Manufacturing buyer: *Dave Dickey*
Editorial assistant: *Diana Penha*

© 1992 by Prentice-Hall, Inc.
A Simon & Schuster Company
Englewood Cliffs, New Jersey 07632

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Apple and Macintosh are registered trademarks of Apple Computer, Inc.
Borland, Turbo-PASCAL, Turbo-C, Turbo Assembler, and Turbo-Debugger are trademarks of Borland International.
Codeview is a registered trademark.
Cray is a registered trademark of Cray Research.
IBM, PC-DOS, OS/2, and PS/2 are registered trademarks of International Business Machines Corporation.
Microsoft, Microsoft-PASCAL, Microsoft-C, Microsoft Assembler, and MS-DOS are registered trademarks of Microsoft Corporation.
Motorola is a registered trademark.
NEC is a registered trademark of Nippon Electric Company.
Sun and SPARC are trademarks for Sun Computers.
UNIX is a registered trademark of AT&T (Bell Labs).
VMS and VAX are registered trademarks of Digital Equipment Corporation.

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-451998-1

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericano, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

ServiceNow's Exhibit No. 1013
IPR2015-00631

Contents

PREFACE ix

INTRODUCTION: WHY LOOK UNDER THE HOOD? 1

1 REPRESENTATION AND STORAGE OF INFORMATION 6

1.1 Bits and Bytes 6

1.2 Representing Information as Bit Strings 9

1.2.1 Representing Integer Data, 9
1.2.2 Representing Real Number Data, 12
1.2.3 Representing Character Data, 14
1.2.4 Representing Machine Instructions, 15
1.2.5 What Type of Information Is Stored Here? 15

1.3 Organization of Main Memory 18

1.3.1 Words and Addresses, 18
1.3.2 Storage of Variables in HLL Programs, 21
Analytical Exercises, 29
Programming Projects, 33

2 MAJOR COMPONENTS OF COMPUTER "ENGINES" 34

2.1 Major Hardware Components of the "Engine" 35

2.1.1 System Components, 35

ServiceNow's Exhibit No. 1013
IPR2015-00631

2.1.2 *General CPU Components, 38*
2.1.3 *iAPX CPU Components, 41*
2.1.4 *Motorola 68000 Family CPU Components, 48*
2.1.5 *The CPU Fetch/Execute Cycle, 49*

2.2 Software Components of the Computer "Engine" 53

2.3 Speed of a Computer "Engine" 56

2.3.1 *CPU Architecture, 56*
2.3.2 *Parallel Operations, 56*
2.3.3 *Clock Rate, 58*
2.3.4 *Memory-Access Time, 59*
2.3.5 *OS Efficiency, 62*
Analytical Exercises, 63
Programming Projects, 65

3 INTRODUCTION TO THE iAPX INSTRUCTION SET AND ADDRESSING MODES 68

3.1 An Introductory Program 68

3.2 A Brief Look at Instruction Formats 78

3.3 Allowable Combinations of Operations and Operands 83

Analytical Exercises, 85
Programing Projects, 86

4 GENERATING, LOADING, AND EXECUTING PROGRAMS 89

4.1 Use of DEBUG for Loading and Executing Programs 90

4.2 Introduction to iAPX Assembly Language 96

4.2.1 *Hire a Clerk! 96*
4.2.2 *A First MASM Example, 98*
4.2.3 *Command Sequence and Syntax for MASM and LINK, 118*
4.2.4 *Debugging Assembly Language Programs, 121*
4.2.5 *Further MASM Examples, 130*
4.2.6 *Tools Developed So Far, 138*

4.3 More on Program Loading and Transfer of Control 139

4.4 Loading and Executing Programs Derived from HLL Sources 143

Analytical Exercises, 144
Programming Projects, 147

5 MODULAR PROGRAMMING: SUBPROGRAMS, LINKERS, AND MACROS 151

5.1 Stacks 152

ServiceNow's Exhibit No. 1013
IPR2015-00631

5.2 Procedures 156

5.3 Machine-Level Aspects of Procedures in High-Level Languages 170

5.3.1 What the Compiler Produces from HLL Procedure Calls, 171
5.3.2 Mixed-Language Programming, 179

5.4 Macros 193

5.5 MAKE: A Maintenance Utility for Program Modules 197

Analytical Exercises, 199
Programming Projects, 201

6 A FURTHER LOOK AT THE iAPX ARCHITECTURE 204

6.1 A Further Look at the iAPX Flags Register 204

6.1.1 The Carry Flag, 205
6.1.2 The Overflow Flag, 207
6.1.3 Other Flags, 211

6.2 A Further Look at iAPX Addressing Modes 212

6.2.1 Register Mode, 212
6.2.2 Immediate Mode, 214
6.2.3 Direct Mode, 214
6.2.4 Indirect Mode, 215
6.2.5 Indexed Addressing, 215
6.2.6 Based Mode, 221
6.2.7 Combined Indexed and Based Modes, 223
6.2.8 Use of the Addressing Modes in JMP and CALL Instructions, 224
6.2.9 Segment Override, 224

6.3 A Further Look at the iAPX Instruction Set 226

6.3.1 Other Arithmetic Instructions, 226
6.3.2 Logical (Boolean) Operations, 233
6.3.3 String Instructions, 245
6.3.4 Loop Instructions, 251
6.3.5 Miscellaneous Instructions, 253
Analytical Exercises, 256
Programming Projects, 257

7 INPUT/OUTPUT 260

7.1 Introduction to I/O Ports and Device Structure 261

7.1.1 I/O Address Space Approach, 262

ServiceNow's Exhibit No. 1013
IPR2015-00631

7.1.2 Memory-Mapped I/O Approach, 263
7.1.3 I/O Ports and Device Structure in the IBM Microcomputer Family, 266

7.2 Interrupt-Driven I/O 277

7.2.1 Basics of the Interrupt Sequence, 279
7.2.2 Arranging Priorities among Devices, 293

7.3 I/O through System Calls 296

7.4 I/O in HLLs 306

Analytical Exercises, 309
Programming Projects, 310

8 INTRODUCTION TO OPERATING SYSTEMS 315

8.1 Mechanisms to Call OS Services 316

8.2 "Cooking" Services 319

8.3 File Systems 320

8.4 Process Management 334

8.4.1 TSR Programming, 335
8.4.2 The Infrastructure of Time-Sharing, 340

8.5 Memory Management 345

8.5.1 Memory Sharing, 345
8.5.2 Virtual Addressing, 347
Analytical Exercises, 351
Programming Projects, 352

Appendices 354

I ASCII AND SCAN CODES 354

II THE iAPX INSTRUCTION SET 359

III COMMANDS FOR ASSEMBLING, COMPILING, LINKING AND DEBUGGING 382

IV SELECTED DOS AND BIOS SERVICE ROUTINES 385

V PASCAL/C TUTORIAL 388

INDEX 395

ServiceNow's Exhibit No. 1013
IPR2015-00631

two sectors of the file PRIME.ASM happened to be adjacent on the disk. Thus, if we access them sequentially, no seek will be needed to read the second sector after the first. But in general, storage of consecutive portions of the file will *not* be in contiguous sectors of the disk.

This problem can be solved by having, say, two sectors per cluster. Since the several sectors in a cluster are contiguous by definition, having two sections per cluster will force the desired contiguity. The drawback, though, is that if one has a large number of small files—say, less than one sector in size—each file would take up *two* sectors, since they could not occupy only part of a cluster. Thus, space on the disk would be wasted.

Unix file systems are largely similar to the MS-DOS system we have been discussing. Both DOS and Unix treat a file merely as a long stream of consecutive bytes. If the file is a **text** file, meaning that its bytes are supposed to be interpreted as characters, the *user* will think of the file as being broken down into lines of text, and utility programs such as the VI text editor will display the file in this way on a monitor screen. But from the OS's point of view, the carriage return and line feed characters—which define those lines that the user sees—are just characters, with the ASCII codes 0DH and 0AH, respectively, and are no different from the alphabetical or other characters. All that the OS must do is find space on the disk for these characters—in noncontiguous sectors if necessary—and maintain a list of pointers showing where these sectors are. There is no concept in DOS or Unix of lines in a file, even an ASCII file.

In some other OSs the structure of a file may be quite different: the OS itself may keep track of "lines" within the file (called **records**), and keep pointers to each line. But in the DOS system we have presented here, DOS does not record where, say, the 124th line of a text file begins; all we can do is read the file from the beginning, counting carriage return and line feed characters until we accumulate 123 such pairs. Of course, we *can* write our program so that it creates a table of correspondences between line numbers and sectors for our file, if we will access the file by line number often enough to make it worthwhile to create such a table. But the point is that DOS and Unix do not do this for us, whereas some OSs do. On the other hand, those OSs may waste space, either by storing such a table or by making sure that all lines are the same length, say, 80 characters per line (padded with blanks if necessary). Some OSs do the latter so that we can find the position of a certain line just by multiplying by 80.

8.4 PROCESS MANAGEMENT

It is often very useful to have several user programs alternate execution. In Example (a) of Section 7.2, for instance, we briefly discussed a situation in which one might run a game program to entertain oneself while waiting for a print program to complete its work of printing out a large file. An extension of this concept that will be familiar to many readers is **time-sharing**, an environment in which many users (or one user running several programs) appear to be running programs simultaneously, but are actually alternating the programs in execution.

ServiceNow's Exhibit No. 1013
IPR2015-00631

In this section, we will discuss how to set up the alternation of several such programs. MS-DOS offers the capability of writing **terminate-and-stay-resident** (TSR) programs, which would enable us to run the previously mentioned game program and print program "simultaneously." We will present an example of TSR programming in Section 8.4.1. MS-DOS does *not* offer time-sharing services, but the IBM microcomputer hardware is capable of them, and several OSs available for IBM microcomputers, such as OS/2 and Xenix, MINIX, and other Unix OSs for PCs, do implement time-sharing. We will show how time-sharing is done in Section 8.4.2.

First, we introduce the notion of a **process**. This might be defined as an instance of execution of a program. The word "instance" is important, because if several users, (say six) are currently using the same program, they account for six processes, rather than one. Thus, in the discussion that follows, we will speak in terms of various processes being active, instead of various programs.

Incidentally, a given user might have several different processes active at the same time. For example, readers who have used Unix may have had some experience with the '&' command, such as in the command line

```
% cc g.c &
```

The '%' is the prompt symbol on many Unix systems. The command

```
cc g.c
```

specifies that we want to run the C compiler on a source file named g.c. The symbol '&' means that we wish to be able to submit new commands while the compilation is in progress. If we now type

```
% vi x
```

to use the vi editor on the file x, we will initiate a new process, with the vi editor, in addition to the process cc.

Again, process management strongly depends on the availability of interrupts, both hardware and software, but especially hardware.

8.4.1 TSR Programming

In TSR programming, we terminate a program but tell the OS to keep it in memory. Consider DOS service number 21/4C, which we have been using all along to terminate our programs, i.e., with the familiar sequence

```
MOV AH,4CH
INT 21H
```

Our program has finished execution, and we use this DOS service to return to DOS, which will then print out the usual '>' prompt, inviting us to submit our next command.

ServiceNow's Exhibit No. 1013
IPR2015-00631

But this service also does something else: it notifies DOS that we no longer need the program in memory, and thus, DOS is free to overwrite it by loading some other program in the same area of memory.

By contrast, DOS offers terminate-and-stay-resident services, such as DOS service number 21/31 and DOS service number 27. (We will use the latter, but the two are very similar.) These services enable us to return to DOS but tell DOS *not* to overwrite the area of memory occupied by our program.

For instance, in the game-and-printer example mentioned earlier, the printer program would initiate the printing of the first character and then terminate and stay resident. The terminate action would allow us to submit another command at the '>' prompt, in this case, the command to run the game program. But the stay-resident action would mean that the printer program would stay resident in memory: once the printer was ready for more characters to print, the printer program would be right there in memory, ready to supply the printer. We are, in effect, telling the OS not to overwrite the printer program with anything else.

As outlined earlier, the key to switching back and forth between the new program and the TSR program is interrupts. In the game-and-printer example, we would start the printer program first. It would then start the printing of the first character and perform a TSR exit. We would then start the game program. When the printer became ready for more characters, it would send an interrupt, which would suspend execution of the game program. The printer program would then initiate some more printing and, subsequently, let the game resume. And the person playing the game would probably not notice any slowdown in computer response during this switching between programs.

Program 8.3 is a program that constantly displays the time of day at the top-right corner of the monitor screen—no matter what program we are running. The display is done by a TSR program that runs whenever there is an interrupt from the Intel 8253 timer chip, which occurs 18.2 times per second. DOS's ISR for this chip includes a call to BIOS service number 1CH, which is actually a dummy procedure consisting only of an IRET instruction. (The reader is urged to verify this by using DEBUG.) We will instead point the interrupt vector for INT 1CH toward our TSR, which will refresh and update the time-of-day display at the top-right corner of our screen.

Let us suppose Program 8.3 is in the file TOD.ASM. We would assemble and link the program. Then we would produce a .COM file from the .EXE file, using the EXE2BIN utility:

```
> EXE2BIN TOD TOD.COM
```

(For technical reasons we will not present here,TSR programs should be run as .COM files instead of as .EXE files.)

We would then run TOD, typing

```
> TOD.COM
```

upon which the program will almost instantly "finish" (it is not really finished, since it will be repeatedly reactivated, 18.2 times per second), and we will see the DOS '>' again. Suppose we then use the VI editor, typing

ServiceNow's Exhibit No. 1013
IPR2015-00631

```
> VI Z.C
```

to edit some file Z.C (having no relation to TOD.ASM). Then, while viewing the file Z.C on our screen, the time of day would appear there, too, even as the time changes, because our TSR will update the displayed time every five seconds.

The reader should assemble, link, and run the TOD program to get a feel for what it does. The latter is extremely important; make sure to run this program before reading about it. Especially vital are the interrupts, so make sure you understand their role thoroughly.

```
CSG SEGMENT
   ASSUME CS:CSG,DS:CSG
   ORG 100H

NTRYPT:  JMP INIT

NTICKS DW (?)  ; number of timer ticks

INT1C PROC

   ; since this procedure is called by an INT, we need to reallow interrupts
   STI

   ; save registers
   PUSH AX
   PUSH BX
   PUSH CX
   PUSH SI

   ; increment count of clock ticks, and check if ready to redisplay time
   INC NTICKS
   CMP NTICKS,100D
   JL XIT

   ; read real-time clock
   MOV AH,2
   INT 1AH

   ; display time
   MOV BX,CX
   MOV SI,152D
   CALL ADISPBX

   ; start over
   MOV NTICKS,0

XIT:
   POP SI
   POP CX
   POP BX
```


ServiceNow's Exhibit No. 1013
IPR2015-00631

```
   POP AX
   IRET
INT1C ENDP

ADISPBX PROC   ;  adaptation of DISPBX in Program 4.1

   PUSH AX
   PUSH BX
   PUSH CX
   PUSH DX
   PUSH ES

   ;  point ES to screen memory
   MOV AX,0B800H
   MOV ES,AX

   ;  each nibble to be printed will be taken from bits 15-12 of BX
   ;  so we will keep rotating BX by 4 bits at a time, each time
   ;     moving a new nibble into bits 11-8

   ;  this will have to be done 4 times, for the 4 nibbles of BX,
   ;     and we will use DH as the loop counter, starting at 4,
   ;     then 3, 2, and 1
   MOV DH,4

   ;  the rotation of 4 bits will be indicated by CL, so put 4 there
   MOV CL,4

  LP: MOV DL,BH  ;  put a copy in DL, to work on it there

   ;  prepare the nibble for printing
   AND DL,0F0H  ;  put 0's in the lower 4 bits of DL, leaving our nibble unchanged
   ROR DL,CL  ;  rotate so that the nibble is in bits 3-0 of DL

   CALL APRNIB  ;  print that nibble

   ROL BX,CL  ;  rotate BX to get to next nibble

   ;  decrement loop count and check if we are done with all nibbles yet
   DEC DH
   JNZ LP

   POP ES
   POP DX
   POP CX
   POP BX
   POP AX
   RET  ;  return to calling program
ADISPBX ENDP

APRNIB PROC  ;  adaptation of PRNIB in Program 4.1

   ;  must convert numeric value in DL, which is in the range 0-F, to ASCII
```


ServiceNow's Exhibit No. 1013
IPR2015-00631

```
   CMP DL,9  ;  is it 0-9 or A-F?
   JG A_F   ; if so, go to the code to handle the A-F case
   ;  if not, we are in the 0-9 case
   ADD DL,30H   ;   ASCII codes for the characters '0'-'9' are 30H-39H
   JMP WR_CHAR  ;  OK, ready to write to screen
A_F: ADD DL,37H  ;   ASCII codes for the characters 'A'-'F' are 41H-46H

   ;  here is where the actual writing to the screen takes place
   WR_CHAR:
   MOV BYTE PTR ES:[SI],DL
   ADD SI,2

   ;  OK, nibble printed, so return to caller
   RET
APRNIB ENDP


INIT PROC NEAR

   ; set DS
   PUSH CS
   POP DS

   ; point ES to low memory
   MOV AX,0
   MOV ES,AX

   ; set vector for INT 1C
   CLI  ; don't allow interrupts while changing vector
   MOV WORD PTR ES:[4*1CH],OFFSET INT1C
   MOV WORD PTR ES:[4*1CH+2],CS
   STI  ; reallow interrupts

   MOV NTICKS,0  ; start count

   ;  OK, terminate program but stay resident
   ;  first set program size
   MOV DX,OFFSET INIT
   INT 27H
INIT ENDP

CSG ENDS

END NTRYPT
```

Program 8.3

A requirement for .COM files is that they fit into a single segment. Thus, no separate data segment is declared in Program 8.3; instead, the data (here consisting of only one item, NTICKS) is simply declared in the code segment. Another requirement is that the code begin at offset 100 of the segment, which the pseudo-op ORG 100H accomplishes.

ServiceNow's Exhibit No. 1013
IPR2015-00631

DOS service 27, at the end of the program, performs the TSR operation. Its parameter in DX is the number of bytes to be reserved in memory. Since the INIT procedure, as indicated by its name, does only initialization, we do not need it to stay in memory; we only need what precedes it, and there are OFFSET INIT bytes in that portion of the program.

Now let us look at INT1C, our ISR for INT 1CH. The 8253 timer issues an interrupt 18.2 times per second. Whatever program we are running at the time—say, VI—will thus be interrupted 18.2 times per second, and the procedure INT1C will run each time. INT1C keeps track of how many timer ticks have occurred, and after every 100th tick—approximately every five seconds—it prints the time of day on the screen. It acquires the time of day from the **real-time clock**, via BIOS service number 1A/02. (The real-time clock is available on most machines; however, even those machines which do not have a real-time clock can still keep track of the time by using the 8253 timer itself, again by counting ticks within the 8253's ISR.)

Many popular commercial products are TSR programs. For example, the so-called "hot key" applications, in which a certain key can be used to suspend the current program suddenly and temporarily take some other action, rely on the TSR approach. One could make a TSR program that provides access to a dictionary stored on disk. Then the person writing the "Great American Novel" using a word-processor program, or the person playing a word-game program, could quickly access the dictionary by typing a given key—say, F1—without having to "pack up and leave" the program he or she was using.

The typical pattern is to replace a DOS or BIOS ISR. For example, consider the disk cache application discussed in Section 2.3.5.1. BIOS service number 13 reads and writes disk sectors. We could record the original value of this interrupt vector, so that we remember where the BIOS ISR for INT 13H is, but change the vector to point to our TSR. The TSR would check to see whether the given sector is in the cache (which itself would be an array declared within the TSR); if not, the TSR would relay the request to the original BIOS ISR for INT 13H, which would do the work of accessing the sector on disk. (The TSR would "call" the original BIOS 13 routine by simulating an INT, via a PUSHF and a far CALL.)

A caveat is necessary here: do not make calls to any DOS services from within your ISR in a TSR program. If a DOS service is in progress when an interrupt occurs, and then the ISR calls a DOS service too, there is a high risk that the stack will be ruined. It is possible to write the program to sense whether another DOS procedure is in progress before entering DOS, but the code is extremely delicate; it is better to limit calls for OS services in your TSR program's ISR to BIOS routines.

8.4.2 The Infrastructure of Time-Sharing

In time-sharing, the most important hardware interrupt is that of a timer, such as the Intel 8253 timer. This chip will generate periodic interrupts, and since in a time-sharing OS the timer's interrupt service routine (ISR) is part of the OS, the OS has periodic opportunities in which to "take a look around" and decide whether to switch control of the CPU to another process.

ServiceNow's Exhibit No. 1013
IPR2015-00631

Let us make the latter notion more concrete. Suppose that users X and Y are using the same computer, from different terminals attached to that computer. Suppose also that user X requests a program to be run that will have an execution time of five hours, and a split second after X hits the return key, user Y requests a program to be run that will finish after only one second. It would be terribly unfair to have Y wait five hours for X's program to finish.

To avoid such an inequity, time-sharing systems make processes take turns running. A turn is called a **time slice**, or a **quantum**. The **quantum size** is a fixed time interval, say 50 milliseconds (ms). In the example in the last paragraph, suppose, for simplicity, that neither X's nor Y's program has any system calls and that those two programs are the only two processes in the system right now. Then X's process would run for 50 ms, then Y's process would run for 50 ms, then X would have another 50-ms turn, then Y again, and so on. This scheduling policy, which is called **round robin**, will result in Y's program being done after 20 turns and with Y being delayed only by one second (due to waiting during X's first 20 turns), instead of having to undergo a five-hour delay.

The other big advantage to time-sharing is that the computer is not wasting time in wait-loop I/O operations. If, say, user X's program has reached a **readln** operation in its Pascal source code, we certainly do not want our expensive computer to waste its time looping until the user types a key. Instead, we give some other program a turn and rely on the keyboard interrupt to notify the OS when user X finally gets around to hitting a key. In other words, time-sharing generalizes the ideas suggested in example (a) in Section 7.2.

Note again that even in single-user systems, the typical case in PC applications, the one user could have several programs active at once. This is in fact one of the major reasons for the interest in extensions of MS-DOS such as OS/2 and Windows 3.0—a desire for easily setting up several program executions at once. Thus, even though we have been speaking in terms of several users, what really counts is that several programs are active at once, whether they are invoked by different users or all by the same user. So we will just refer to *programs* X and Y in what follows rather than to *users* X and Y. (Referring to *processes* X and Y would be even better.)

To see why the timer interrupt is so important, suppose that the OS, during its initialization period, programs a timer to interrupt 60 times per second. Then three such interrupts will occur during a 50-ms turn. Thus, we can write our timer ISR to count these interrupts and to end the turn when the count reaches 3.

The reader should convince him- or herself that we simply could not implement this taking-turns policy without timer interrupts. For example, without the timer interrupts, once program X got control of the CPU, it would simply run for five hours to completion. The OS could not intervene and stop that program, since *that* program would be running, not the OS. The CPU would be, as always, simply stepping repeatedly through its step A, step B, step C, etc., cycle, so without interrupts, X would continue to run, and the OS would be completely dormant—completely powerless to stop X. The timer interrupt is thus crucial in forcing the process to relinquish the CPU to the OS.

ServiceNow's Exhibit No. 1013
IPR2015-00631

Program 8.4 is an outline of how the timer's ISR might be written for a time-sharing OS for an IBM microcomputer. As indicated earlier, it keeps a count of timer interrupts that have occurred so far in the current program's turn and ends the turn when this count reaches 3. This is evident in the code

```
INC TICKS ; TICKS contains # of timer interrupts so far this turn
CMP TICKS,3 ; see if this is the third timer interrupt
JE ENDTRN  ; if so, then end current program's turn
```

When a turn is over, say, for program X, OS will record X's current values of IP, CS, FR, AX, BX, etc.—that is, all the registers in the CPU. Clearly, recording these values is necessary, so that when X's next turn comes, X will be able to resume execution in precisely the same setting as existed when its last turn ended. The OS keeps a record of all processes in a table; we will call this table the process table. For each process, the process table will include a pointer to a "save area" in memory for that process; all the register values will be saved there.

After saving X's register values, the OS will look at its process table to determine which process should be given a turn next. If program Y is due to run, the OS will need to restore all of Y's saved register values first, as indicated in the comments in the latter portion of Program 8.4. After restoring these values, the OS will give control of the CPU to Y, and Y's turn will begin.

It is worth noting how the OS passes control to Y. As can be seen at the end of Program 8.4, control is passed via an IRET instruction. But the operation is not quite as simple as it may seem. To see why, look at the KISR procedure in Program 7.6. It, too, ends with an IRET instruction. But the difference is that KISR is entered when Program 7.5 is interrupted, and the IRET in KISR will make us return to Program 7.5. In other words, our entry to and return from KISR involve the same program, Program 7.5.

By contrast, when CLKINT in Program 8.4 executes for the third time during X's turn, we will *return* from CLKINT to Y's program, even though we *entered* CLKINT from X's program! This may seem strange at first, but it becomes clear if one notes by that before the IRET there is code in which the OS restores Y's SS and SP values. Thus, it will be Y's stack that will be popped during the execution of IRET, not X's stack, and we will return to Y, not X, since Y's stack will contain Y's IP and CS values that were saved at the end of Y's previous turn.

This whole sequence of ending X's turn and starting a turn for Y is called a *context switch*: we have changed the "context" of the machine—i.e., the set of values in all the registers—from that of X to that of Y. Note that this switch-over period is time that is unproductive—necessary overhead in order to achieve the illusion of simultaneity between X and Y. Some CPUs, such as those of VAX machines, have the capability to do all the saving and restoring within the same single special instruction, thus reducing the overhead.

The code in Program 8.4 has been simplified, to make the presentation easier to follow. The use of the word "restore" in the comments (e.g., just above MOV TICKS,0) assumes that all the processes that are currently running programs (actually, taking turns running programs) have already had at least one turn. So there should be code to check

ServiceNow's Exhibit No. 1013
IPR2015-00631

```
CLKINT:  PUSH DS
         ; move OS's data segment location into DS (code not shown),
         ;  so that OS can access its own variables, e.g., Process Table,
         ;  TICKS, etc.
         INC TICKS ; TICKS contains # of timer interrupts so far this turn
         CMP TICKS,3  ; see if this is the third timer interrupt
         JE ENDTRN  ; if so, then end current process' turn
         POP DS     ; if not, then restore current process' DS value
         IRET       ;    and return to current process

ENDTRN:  PUSH BX
         ; look at process table and determine where current process' save area
         ;  is, and point BX to it (code not shown)
         PUSH BX
         MOV [BX],AX       ;save current process' AX value
         ADD BX,2
         POP AX;        ;recover current process' BX value
         MOV [BX],AX       ;save current process' BX value
         ADD BX,2
         ; save current process' CX, DX, DI, SI in the same way (code not shown)
         ; save current process' DS, SS, and SP values (code not shown)
         ; note that at this point, SS and SP are still pointing to
         ;  current process' stack, and current process' values of IP, CS
         ;  and the flags register are still on current process' stack
         ;  so we do not have to save those as we did the other registers
         ; look on the process table for a new job, the "next" process
         ;  to start a turn for (code not shown)
         ; restore next process' AX,BX,CX,DX,DI,SI,SS, and SP (code not shown)
         MOV TICKS,0  ;  initialize the number of interrupts this process' turn
         ; restore next process' DS value (code not shown)
         ; next process' values of IP, CSs, and the flags register are still
         ;  on next process' stack, left over from this user's last turn
         ;  (so no code needed for restoring values of these from the
         ;  process' entry in the OS process table)
         ; so the IRET below will take us back, exactly to the point
         ;  which was executing when this user's last turn ended
         ;but first, send EOI to 8259A (code not shown)
         IRET
```

Program 8.4

whether the next process has already had a turn, and if it has not, then other code would be executed, with all the references to "restore" changed to "initialize" instead.

In illustrating round robin scheduling with programs X and Y, we made the simplifying assumption that neither of the programs makes any system calls and that no other processes are currently in the system. Let us now discuss what happens in general.

Clearly, there may be more than just two processes in the system, perhaps many more. In its simplest form, round robin scheduling will service each process in turn, in

ServiceNow's Exhibit No. 1013
IPR2015-00631

circular fashion. In more sophisticated OSs, more complicated priority systems might be imposed.

Furthermore, a process' turn often ends early, before 50 ms have elapsed. Suppose X occasionally makes system calls. Since such calls execute procedures within the OS, X is voluntarily relinquishing control of the CPU to the OS. This ends X's turn early, before 50 ms are up. Since most system calls deal with I/O, and since I/O is a relatively slow operation, it makes sense to start a new turn for another process—say Y—rather than to wait patiently for X's I/O request to be completed. In fact, after the new turn for Y is completed, X's I/O request may *still* not be complete, so we could then give Y another turn or a give a turn to some third process. Of course, another possibility is that X's system call is, say, in MS-DOS, to DOS service number 21/4C. In that case, X will have finished execution and will not need the rest of its turn.

More detailedly, in addition to its process table, the OS will keep a ready list. This list shows which processes are ready to get another turn. Those which are not ready are waiting for something, e.g., completion of an I/O operation. Assuming an equal-priority system, the OS would simply continue to gives turns cyclically to all processes on the ready list. When a process' turn ends—due to 50 ms elapsing, not due to making a system call—it simply joins the end of a queue, and the OS gives a turn to whichever process is currently at the head of the queue.

As an example, suppose that the system currently has three processes, A, B, and C, and that A is executing and B and C are on the ready list. Suppose A reaches a system call, say, to read a character from the keyboard. The system call, in the form of a software interrupt as in Section 8.1, transfers control to the OS. The OS ends A's turn and puts A on a blocked list, where **blocked** refers to the fact that A could not run right now even if the OS were to give it another turn—A is blocked by its pending I/O action. The OS then looks on the ready list for a program that is ready to start a new turn. The OS then starts a turn for B, since B is at the head of the list. After that, the ready list consists of C only.

Suppose that B's turn ends after the normal 50 ms—that is, B had no system calls during this time. Then again, the OS takes over, due to the interrupt from the timer, and again, the OS looks at the ready list, seeing C. The OS starts a turn for C and puts B on the ready list; now the ready list consists of B only.

Suppose now that during the middle of C's turn, the user running process A finally types a character. The keyboard will generate an interrupt, and again, the interrupt routine will be some section of the OS. The OS notes from the blocked list that A was waiting for this I/O, and thus, the OS puts A back on the ready list. Then C's turn will be resumed, and after C has run 50 ms, the turn will be over, and B's next turn will start. Then the next turn will be A's, since it has now rejoined the ready list.

Once again, it is very important to keep in mind the role of interrupts in passing control from a user program to the OS. Control can be passed via a hardware interrupt, either from the timer or from an I/O device, or via a software interrupt, when the currently executing process makes a system call. Keep in mind that *the OS has no power whatsoever when a user program is running.* The OS cannot simply "step in" and stop a user program; either an I/O device forces the user program to relinquish control of the CPU, or the program voluntarily does so, via an INT instruction.

ServiceNow's Exhibit No. 1013
IPR2015-00631