IBM

International Technical Support Centers

IBM PC INTERNALS FUNDAMENTALS COURSE NOTES

GG24-3057-00

IBM PC Internals Fundamentals
Course notes

Document Number GG24-3057-00

March 21st, 1986

International Technical Support Center
Department 91J, Building 235-2
Boca Raton, Florida

First Edition (March 1986)

This edition applies to Version 1.00 of PC Internals Fundamentals class.

This document describes the different PC software layers and the architecture of the 8088 and 80286 processors.

The contents of this book is also designed as additional course material for teachers and attendants of a PC Internals Workshop.

WS (192 pages)

PREFACE

Over the years the IBM Personal Computer has become an industry-wide standard in personal computing. For this and other reasons it is essential that PC support people have in-depth technical expertise on the internals of members of the Personal Computer family. The objective of the PC Internals Fundamentals is to provide you with a solid framework for understanding the technical concepts of Personal Computers, and to enable you to give quality technical and marketing assistance for PC products. This course may be considered a prerequisite for more in-depth courses on existing and future PC hardware, operating systems and software. Anyone in a support position for Personal Computers will find this course an asset.

The Introduction contains information on course prerequisites, but is also an integral part of the course. If you choose not to read the introduction you will discover that you have missed a good many valuable technical details. If you plan to teach the course, you should also read the Note to Instructors.

Because the Introduction incorporates most details usually covered by abstracts and overviews, we have not included separate sections for these topics. When you are ready to begin the course, please turn to the Introduction.

RELATED PUBLICATIONS

The following publications can be useful as complements to the information contained in this bulletin:

- Macro Assembler Manual Version 2.0
- DOS Manual Version 3.1
- DOS Technical Reference Manual
- PC Technical Reference Manual
- Intel[1] iAPX 88 Book With An Introduction To The iAPX 188, Order Number 210200-002
- Intel[1] iAPX286 Programmer's Reference Manual, Order Number 210498-001
- Intel[1] iAPX286 Operating Systems Writer's Guide, Order Number 121960-001

---

[1] Intel is a registered trademark of Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

TABLE OF CONTENTS

1.0 NOTE TO INSTRUCTORS

This guide, and the accompanying course materials, were developed at the International Technical Support Center of IBM Corp. in order to teach a PC Internals Fundamentals Course. The immediate objective of the development was to allow us, the course developers, to prepare our own class versions of the course; a more global objective was to provide a set of materials which would enable anyone with a strong knowledge of PC Internals, or anyone having graduated from the original course, to teach the course after minimal preparation and research, using the teaching materials provided.

Three sets of deliverables are required to teach this course. The first is this guidebook, which covers each of the topics of the course in considerable detail. The guide contains all of the technical information which the original lectures were designed to contain (except for some of the bad jokes), as well as some additional details for instructors and for graduates of the course who may wish to enhance their PC Internals knowledge.

The second deliverable consists of a set of PC Storyboard presentations. The presentations step through each lecture in approximately the same order as does the guide. We have used the animation features of PC Storyboard in order to illustrate more concretely some of the difficult concepts of PC internals. We have also attempted to make the presentations visually stimulating and interesting. No matter how well a course on PC internals is taught, there will be students who will simply not understand some of the materials presented, and the verbal and visual asides will at least keep these students awake so that they don't miss those aspects of PC internals which they are able to grasp.

The third deliverable consists of a set of student handouts. These handouts are essentially black-and-white copies of the more important Storyboard displays. We have attempted to reduce the volume of the handouts by not including more than one or two pages' worth for each animated sequence, and by excluding some of those displays which were designed to keep the students awake.

If you plan to teach this course you should obtain all the materials well in advance. You will obtain these deliverables by attending the course. You may have to print your own student handouts from diskette files the ITSC provides, and this takes time; you should also print out overhead-projection transparencies ("foils") if you do not wish to use a video projector for the course, or for contingency purposes should the video projector fail.

When preparing to teach the course, you should go through each Storyboard lecture carefully, note the sequence of screens, practice explaining the information on each screen, and become familiar with how screens blend into one another. You should try to use the information provided in the guide to orally complement the information on the displays. We have not provided a detailed script of the Storyboard presentations for two rea-

sons. One is that we wish to provide instructors with the freedom to present the materials their own way, with any changes, additions or deletions they see fit to make. The second, related reason is that, as any student of such a course can testify, a lecture in which the instructor merely reads or recites the script is much less engaging and no more effective than providing the script directly to the student.

We have provided three lab exercises to reinforce students' understanding of some of the materials. The lab scripts should be distributed to the students; students should do the labs in groups of two or three people per PC. As instructors you will be responsible for installing the Resident Debug Tool on the machines, for providing the sample programs to the students, and for providing guidance and assistance in the course of the lab. One implication of this is that you should become fully acquainted with RDT, and should know intuitively how to do every step of the labs.

The only other general guidance we can provide is that, if you do choose to teach the course using a video projector and the Storyboard presentations, you may want to do some of the lecturing using overhead transparencies instead of PC Storyboard. Students will find it difficult to sit through 24 hours of lectures presented in a single medium. By having back-up transparencies of the Storyboard presentations, you will be able to switch between video and foil presentations according to the mood of your audience.

After preparing or teaching the PC Internals Fundamentals course, if you have criticisms or suggestions for improvement of the course or the teaching materials, by all means submit your comments to the address on the reader comment form at the back of this guide. We will attempt to implement your suggestions in future releases of this course.

2.0 INTRODUCTION

Imagine the perfect personal computer: one which has an integrated range of complex applications built right into its hardware, that has a very high-level interface so that even the least computer-literate user can quickly learn to operate it; a computer which has all conceivable I/O resources built right into it, all the memory you would ever want. A personal computer with an architecture so thoroughly thought out, and with applications so bug-free and easy to use, that no one would ever need to program it.

While the above description seems to have some attractive characteristics on first reading -- an integrated set of applications, I/O devices as standard rather than optional, lots of memory, and so on -- it places serious limitations on the eventual owner of such a computer. The most obvious is that the customer will pay for every component of the computer, whether s/he needs that component or not. Another limitation is that the applications, or even the hardware, may not exactly meet the customer's requirements. Every bell and whistle has been incorporated into the integrated text processing package, but you may only want to use the program to write letters to your grandmother -- you don't need the ability to create 9,999 page documents, or to see underscore and italics on the screen. Conversely, you may decide that you need a program to calculate the spawning patterns of Pacific Salmon; you won't find this in the integrated software built into the system's hardware.

What we learn from the above is that a closed-architecture personal computer with a single set of fancy hardware and software components may address the needs of a small number of users, but that as soon as a requirement arises which isn't addressed by the computer, the computer becomes rather useless; conversely, as soon as a feature of the computer is not actually needed by a user, the customer is wasting money on unneeded components.

When IBM designed their Personal Computer the developers firmly believed that an open architecture was essential to the success of the machine; that the more hardware manufacturers and software developers knew about the PC, and the more flexibility users had in choosing what hardware and software configurations they needed, the more widely the PC would proliferate. The belief paid off -- there are more programs and OEM hardware available for the IBM PC than for any other architecture of personal computer, and the PC family is one of the most successful ventures IBM has undertaken. But there were negative aspects to the open architecture as well, as far as IBM was concerned -- it took little time for IBM-compatible PC's to appear on the market, because the open architecture was well-known and easy to copy -- and OEM adapters and I/O resources for the Personal Computer are as widely sold as IBM adapters and I/O.

Because of the PC's open architecture, and because of the resulting abundance of software and peripherals available for the PC, it is crucial that workstation support professionals, within IBM and elsewhere, understand the inner workings of that architecture and know its strengths and

weaknesses. The course on which this guide is based was designed for IBM workstation specialists who have minimal knowledge of the Personal Computer, with the intent of taking them step by step through the physical, electronic and software components of the PC internals. If a workstation specialist is to find viable solutions to an individual's or company's personal computing needs and to provide support, the specialist must know how to diagnose any problems the customer may encounter. The key to determining the cause and possible remedy of IBM PC problems is to know the PC architecture.

2.1 PREREQUISITES

The course was designed for people who have little prior training on the IBM PC, but who are familiar with programming and computing concepts. It is important that you understand, before participating in this course, concepts such as hexadecimal and binary arithmetic, the difference between processor storage, main storage and secondary storage, and other fundamental computer concepts. You should also have some programming experience in any programming language. If you do not have at least a passing acquaintance with at least one language, we suggest you spend a day or two learning IBM PC "C" or Pascal or BASIC before you begin the course.

You should also have worked with IBM PC's before. You should know how to use DOS commands and services such as Copy, Dir, Debug etc., and have worked with applications on the PC. You should know what a diskette drive looks like, and how to power on the PC.

The course was **not** designed for people who know PC Assembler, who understand the operation of the 8088, and who know how the internals of BIOS and DOS work. The course will add little to the expertise of people in this category. PC gurus should be teaching this course, not studying in it. However, if you have only played around with Assembler or DOS/BIOS internals, PC Internals Fundamentals will broaden your PC knowledge.

2.2 TEACHING THE COURSE

These course notes are designed to address three different audiences. Students who have successfully completed the in-class version of this course can use these materials for later reference and to clear up any aspects of a lecture which they did not understand. Students who wish to study PC Internals Fundamentals on their own can also do so, using the International Technical Support Center Red Guide version of the course. And successful graduates of the course who wish to teach it to others can use the notes as a teaching guide. The course is designed in such a way that it can be taught by anyone who has participated in it, after minimal additional research and preparation. If you do plan to teach it, you should read the Note to Instructors before this introduction; it will

provid e you with information on how to obtain the diskettes used for teaching, and on how to prepare the course. Nevertheless this text is designed primarily for study and reference.

2.2.1 Sample "C" Program

Before we plunge into the individual components of the PC internals, and before we even identify what those components are, we will examine a simple program written in the "C" language, and follow one of its steps through different layers of the Personal Computer. Over the remainder of the course, we will examine the details of many of these layers, so that you have a basic understanding of the fundamentals of PC Internals. The "C" program is written to run on an IBM PC. We have removed two lines from the example listed below because they are not relevant to understanding the program itself; however, if you wish to compile the program in order to test it or demonstrate it to students, you can refer to the complete source code listing in Appendix A. The sample program follows:

```
int count; int keystroke; main()
     {
     while (count <20)
          {
          keystroke=getch();
          putch(toupper(keystroke));
          count++
          }
     }
```

We will explain the program line by line, before seeing how the various aspects of the PC are used to accomplish some of its work.

1. Int count and int keystroke: The "int" statement declares the variable which follows it as an integer variable. "Int count" means "declare the variable 'count' as an integer". These two statements are required so that later references to the 'count' and 'keystroke' variables can be understood by the compiler.

2. Main(): Every "C" program must have a "main" routine. This is the routine with which execution of the program begins. Notice the empty brackets which follow the "main" statement. In "C" it is normal to declare a function or routine by following it with brackets. The brackets contain any parameters (also called arguments) which are passed to the routine by routines invoking it. As the "main" routine does not receive any parameters, the brackets are left empty.

Notice also the braces on the fourth and last lines. These braces delineate the beginning and end of the "main" routine.

3. While (count < 20): You should know the meaning of the "While" statement if you have met the language prerequisites for this course. The

"While" statement performs the operation or series of operations it precedes until the condition in the "While" statement is valid. So, "While (count < 20)" repeatedly executes the three statements in the inner set of braces until the value in "count" exceeds 20. Each time these three statements are executed, the "while" loop returns to the condition in the "while" statement and tests that condition. If it is true, the loop is executed again. If not, processing will continue with the step following the end of the "while" loop. In this program that step would be the final brace of the "main()" routine, which ends the program.

4. keystroke=getch(): This statement assigns the value in "getch" to the variable "keystroke" which we declared earlier. What is the value in "getch()"? Well, "getch()" is, as the brackets which follow it suggest, a routine or function, just like "main()" was a routine. "Getch()", which stands for "get character", is in fact a function built into "C" whose purpose is to retrieve a keystroke from the keyboard. So this statement assigns the value of the key pressed at the keyboard to the variable "keystroke".

5. Putch(toupper(keystroke));: We will examine this line in two stages. First, the function "toupper(keystroke)". "Toupper(string)" is a "C" function which converts any lowercase characters in the string contained in its brackets to uppercase. So "toupper(keystroke)" converts the keystroke to uppercase if it is lowercase; otherwise "keystroke" remains unchanged.

The statement surrounding the "toupper(keystroke)" function is the "putch" function, which stands for "put character". "Putch()" is a "C" function which displays a character on the screen. The parameter normally used with "putch()" is the character to display; in this case, we actually use the value returned by another function ["toupper(keystroke)"] as the parameter for the "putch()" function. This is an example of nesting functions, where the parameter used in one function is actually a value returned by another function. If we were to write this line of source code out in English, it might read:

Display the character (returned by converting to uppercase the character (keystroke)).

If we wanted anyone to understand it, we would say "This line converts the character "keystroke" to uppercase, then displays it on the screen."

6. Count++: This line adds 1 to the value in the variable "count". In terms of the result it produces, it is equivalent to the statement "count=count+1" in BASIC. By adding 1 to "count" each time the "while" loop is executed, the condition tested at the beginning of the loop ["while (count < 20)"] will cause an exit from the loop after the loop has executed 20 times. If we did not increment "count" or change it in any way, the "while" loop would continue executing until we aborted the program.

2.3 WHAT DOES THIS PROGRAM DO?

Look the program over again, and try to determine exactly what the purpose of the program is. It is not a particularly useful program, but if you can figure out what it does, you will find the rest of this course relatively easy to understand. If you can understand the program after some study or after having it explained by someone who knows the "C" language, you will be able to follow this course but will find it challenging. If you cannot understand the program at all, you should stop taking the course until you have really met the requirement of understanding programming concepts and knowing how to program in at least one language.

We'll list the program again below for your reference. We have included comments, marked by "/*" at the beginning and "*/" at the end. Because the comments are marked off with these symbols, the "C" compiler will not pay any attention to them.

```
int count;                          /*initialize "count" and          */
int keystroke;                      /*"keystroke" as integers         */

main()                              /*this is the main routine        */
  {                                 /*delineated by the outer braces  */

  while (count <20)                 /*do the following routine while  */
                                    /*the "count" variable is smaller */
                                    /*than 20                         */
      {                             /*the "while" routine is delineated*/
                                    /*by these inner braces           */

      keystroke=getch();            /*place a keystroke in "keystroke" */

      putch(toupper(keystroke));    /*display the uppercase version of */
                                    /*the keystroke on the screen     */

      count++                       /*add 1 to the variable "count"   */
      }
  }
```

If you can't figure out what the program is doing, we will tell you. It retrieves a character from the keyboard, converts it to uppercase, and displays it on the screen. It does this set of actions twenty times (for twenty different keystrokes), and then exits. Now that you know what the program does, you should be able to understand every line of the source code.

2.3.1 How does it all work?

We will look at some of the levels the "C" program has to pass through in order to do its work. Rather than examine every step of the program, we will focus on a single step: the retrieval of a keystroke from the keyboard. This step is contained in the statement "keystroke=getch()".

For those with some knowledge of the PC and certain compilers, the information below may seem inaccurate or overly simplified. However it is important for students to grasp the fundamentals of this process before we examine individual aspects of the process. We would prefer to teach the rules before the exceptions.

1. Source code: In order to run the program, someone must first write it. The source code can be written out by hand, or typed directly; eventually it must be entered into the computer as an ASCII file.

2. Compilation: The computer is only capable of executing machine instructions; it does not understand English or the mnemonics of our "C" program. Therefore, we go through a compilation process, which in simple terms converts our source code to an executable machine language program. Don't worry about all the stages involved in compilation, we'll look at them in a later module.

3. Get character: The "getch" routine actually calls a function which was contained in a "C" routine library and which was merged with the program during compilation. Language compilers generally have a set of routine libraries, and individual statements within a program often call a routine taken from one of these libraries. This is one way of allowing high-level programming steps to be translated into low-level machine instructions.

4. "C" library: The "C" library routine for "getch" does not physically retrieve the keystroke itself; instead it issues a call to the PC Disk Operating System (DOS) which will take care of retrieving a character from the keyboard.

5. DOS: DOS manages system resources for applications; it contains a high-level machine-language interface which allows programs easy access to input and output. However, in many situations DOS itself merely acts as a messenger between the application and the low-level I/O routines. When our "C" program requests a keystroke from DOS, DOS actually invokes the BIOS routine for keyboard input instead of processing the request itself.

6. BIOS: The BIOS (Basic Input/Output System) is the low-level interface to I/O devices on the PC. It includes a routine which will return a keystroke from the keyboard to whoever requests it. Assuming no key had been pressed when the BIOS call was made, BIOS will simply wait for a key to be struck. In all the time BIOS waits, our program does nothing, and DOS does nothing. In some situations an I/O routine may allow processing to continue while the I/O event is pending, but for simplicity's sake let's assume that nothing else happens in our example.

7. A key is pressed: The act of pressing a key can occur while the system is sitting and waiting for a keystroke, or while other work is proceeding. Normally, if a program is calculating an equation, for example, that calculation will be interrupted whenever a key is pressed, so that the input may be processed. The calculation will then resume. In our case however, we assumed that the program just stopped while waiting for the keystroke. So now the keystroke must be processed.

8. The keystroke is received: When the key is pressed, a BIOS routine is automatically invoked which receives the keystroke into the system. This is not the same BIOS routine as the one which is waiting for a keystroke; we will see the difference between the two later in the course. Once the BIOS routine which the keyboard invoked has received the input into the system, the BIOS routine which DOS invoked retrieves the keystroke from the system, and passes it up to DOS.

9. DOS returns the character to the program: The "C" routine which called the DOS routine for keyboard input now receives the keystroke from DOS, and regains control so that program processing may continue.

10. The "C" program receives the keystroke: The "C" routine which was called by our program now returns the character to our program. This character is stored in the variable "keystroke" which we declared. However, as far as the compiled program is concerned, "keystroke" does not exist -- the keystroke is simply stored in a location in memory. Once the keystroke is received into the program, processing continues with the next program step.

2.3.1.1 Who did all the work?

Throughout all of this, the 8088 microprocessor, which is the heart of the IBM Personal Computer, performed every single instruction of the "C" user program, the "getch" function, the DOS input request, the BIOS input request and the BIOS input handling from the keyboard, and the return of the value all the way up the chain. We say that "the BIOS does this" or "DOS does that" but in fact what is happening is that instructions in the BIOS or DOS are executed by the 8088, and allow the 8088 to retrieve the I/O. BIOS, DOS, the "C" function, the program, are merely steps or instructions the 8088 must follow.

If we review the order in which events occurred, we obtain the following sequence:

1. I/O request

Program
V
DOS
V
BIOS

In all this, the 8088 performed every instruction.

2. I/O return

Program
A
DOS
A
BIOS
A
System board
A
Keyboard

In all this as well, the 8088 performed every instruction except the signal from the keyboard which told the system board a keystroke was ready.

If we modify some of the steps in this list, and re-order them from the lowest level to the highest, we obtain the major topics we will discuss in the course. They are:

- 8088 Processor
- Personal Computer Architecture (system board and I/O devices)
- Assembly language
- BIOS
- DOS

In addition we have included one other topic: The 80286 Processor.

We will study all of the above topics, in the order they appear.

A schedule of the in-class version of the course is provided in Appendix B for your reference. Prospective teachers will find this of use in planning their own version of the course; independent students may use it to gauge their own progress through the reading materials.

2.4 OBJECTIVES

Upon completion or this course, the student should understand the following topics:

* 8088 architecture

  - History
  - Structure of 8088-based microcomputers
  - 8088 registers and flags
  - Memory addressing
  - Instruction format
  - Input/output
  - Transfer of control
  - Impact of 8088 architecture on PC architecture

* PC architecture

  - Implementations of PC architecture
  - Hardware architecture
  - Software architecture
  - Hardware details

* 8088 Assembly language

  - Origin of assemblers
  - Format of instructions
  - Instruction types
  - Example instructions
  - How data are coded
  - Other programming languages

* IBM PC BIOS interface

  - Origin and purpose of BIOS
  - Power-on self-test
  - Major I/O routines of BIOS
  - Optional BIOS modules
  - How and why programs bypass BIOS

* IBM PC DOS functions
  - compact
  - Purpose and components of DOS
  - IPL of DOS
  - Entering a command
  - Disk format
  - Memory allocation
  - Program interface
  - Types of program
  - Program linking and loading
  - Error handling
  - Device management

* IBM PC DOS extensions

  - Device drivers
  - Multitasking
  - PC Network

* Basics and applications of the 80286

  - Real vs. protected/virtual address mode
  - Types of protection
  - Task and state transitions
  - Input/output
  - Interrupts and extensions

2.5 LABS

During the class version of this course, several labs will be conducted to help students gain a well-rounded understanding of the PC internals through hands-on experience. If you are taking the course only through reading this guide and do not intend to take the course in a class, we suggest that you attempt to carry out the lab exercises described in the various sections. They will aid your understanding of particularly complex subjects such as 8088 Assembly language, the BIOS, and DOS functions.

Now we will proceed to our first technical module, and discuss the 8088 microprocessor, which is the heart of the IBM PC.

3.0 THE INTEL 8088 MICROPROCESSOR CHIP

The Intel 8088 microprocessor is the heart of the IBM Personal Computer. This 16-bit processor is a sophisticated general-purpose CPU, and also has support for a wide range of peripherals. In order to understand any of the other aspects of PC internals it is essential that you understand the technical details of the 8088, and the reasons for which IBM chose the 8088 in its design of the IBM PC. This module will introduce you to the history of the 8088, its architecture and the format of its instructions. By the end of the lecture you should be able to meet all of the following objectives:

- Discuss the development of the Intel 8088, its ancestors and successors
- Describe the internal structure of the 8088
- List all the registers of the 8088 and describe their uses
- Explain the memory addressing scheme of the 8088
- Discuss the format of machine instructions for the 8088
- Understand the significance of I/O port addressing in the 8088
- Discuss the various means of transferring control in the 8088

We will discuss the various topics in the same order as they are listed in the objectives.

3.1 HISTORY

Intel was one of the earliest companies to develop a single chip processor. Prior to this all CPUs consisted of multiple chips, even multiple boards. Intel's first single chip processor was the 4004, a 4-bit processor, used mainly in desk calculators. It was followed by the 8008, an early 8-bit chip which was used in a few programmable peripherals. The 8008 design needed considerable work, and so the 8080 was born. The 8080 was an extremely popular chip, and formed the basis for a number of derivative chips, both from Intel (the 8085 being the best known), and from other companies, the most famous of which is Zilog's Z80. It was with the 8080 generation of microprocessors that the single user computer really began. Previous chips tended to be used simply as programmable components of embedded systems, rather than as computers in their own right. The 8080 generation are still regarded as the epitomal 8-bit machines, and a whole range of peripherals (memory, diskette drives, printers, cassette recorders) was developed to support them.

Intel saw a market for a more powerful microprocessor, and invested considerable research and development into the area. The new processor should manipulate 16-bit quantities as easily as it handled 8-bit bytes. This would enable it to perform arithmetic on numbers of a useful size, for often 8 bits is insufficient (8 bits give us absolute values from 0 to 255; 16 bits give us values from 0 to 65536). A family of 16-bit peripherals would be developed to support this new processor, but these peripherals would be somewhat more expensive than the older 8-bit offerings, so Intel designed two variants of its 16-bit processor - the 8086, which expected to use a 16-bit bus, and the 8088, which was designed to use an 8-bit bus, allowing it to take advantage of the pre-existing and cheaper 8-bit peripherals. The 8086 was introduced in 1978.

In 1980, when IBM Entry Systems Division was looking for a processor chip on which to base the new "personal computer" the Intel 8088 and 8086 were proven chips -- the 8086 had even been used in the IBM Displaywriter. Moreover, Intel had continued development, and new, faster, and more sophisticated Intel processors were in progress, offering an upgrade path from the 8088, should this ever prove necessary. The 8088 was chosen for a number of reasons: it was available cheaply and in quantity, it was capable of supporting a wide range of inexpensive 8-bit peripherals, and it was noticeably more powerful than the 8-bit processors used in virtually every other large-selling microcomputer system. The 8088 is the processor used in the IBM PC, the XT, the Portable PC, the PCjr, the PCjx, and on the Professional Graphics Controller.

Intel followed the 8088 and 8086 with the 80188 and 80186. These were fundamentally the same processors, but with a number of the usual support chips integrated into the processor chip, allowing the design of microcomputers with far fewer support chips. These processors were aimed at portable computer manufacturers, and at computer-on-an-adapter construction (the PC Network adapter uses an 80188). Similarly the 80C88 -- a low-power version of the normal 8088 -- allows the function of an 8088 in a low-power environment, such as for laptop computers. None of these new chips was an advance in processor architecture, just an improvement in chip design.

The 80286 is different. The 80286 is a considerably more powerful and versatile processor than the 8088. It is capable of emulating the 8088 almost exactly at many times the speed -- this emulation is called "real address mode". It is also capable of running in its native mode -- called "protected mode" -- where it realizes its true power. The 80286 has a few additional instructions over the 8088, but its instruction set is otherwise the same, which means that, in real mode, programs can run equally well on the 8088 and 80286, discounting timing differences. Theoretically, a program which does not interfere with the segment registers should be able to run in 80286 protected mode as easily as on an 8088, but in practical terms the segment registers are too integral to most programs. The 80286 is used in the PC AT, running in real-address mode for PC DOS, and in protected mode for Xenix. Because of its increasing importance in the evolution of the PC family, a separate module is devoted to the 80286 at the end of this course.

Intel recently released the specifications for the latest in this family of processors -- the 80386. The 80386 is a 32-bit processor, with some amazing capabilities. One of its most powerful features is its ability to support multiple "virtual processors", where these may be either 8088 or 80286 based, as well as native 80386. This facility, together with its significantly higher speed (approximately five times that of the 80286) make it a processor to watch for the future. Production of the 80386 is planned to start about June 1986.

3.2 STRUCTURE OF 8088-BASED MICROCOMPUTERS

The 8088 microprocessor consists of three major elements:

- Execution Unit
- Bus Interface Unit
- Registers

The Bus Interface Unit provides the interface between the microprocessor and the outside world. The BIU fetches values from memory whenever instructed to do so by the Execution Unit. When the BIU is not fetching operands for the Execution Unit it fetches instructions and places them in the instruction queue for the Execution Unit. This allows the overlap of instruction fetching and execution in the 8088. The instruction queue in the 8088 is four bytes in length.

The Execution Unit is divided into two parts -- the Control Unit and the Arithmetic/Logic Unit. The Control Unit decodes instructions, and controls their execution. The Arithmetic/Logic Unit provides the facility for performing 8-bit and 16-bit calculations on many of the registers of the 8088.

The structure of the 8088 is not important to an understanding of the function of the 8088. It is useful to understand, however, to appreciate the way in which the 8088 achieves notably better performance than its 8-bit equivalents. The overlap of instruction fetching and execution is one of the more important aspects of this performance improvement. Others involve the advantages of built-in 16-bit operations compared with simulated 16-bit operations in an 8-bit microprocessor, and the more sophisticated instruction set.

3.3 8088 REGISTERS AND FLAGS

The architecture of the 8088 is designed around a number of registers, many of which are devoted to special purposes. There are:

- four general purpose registers

- AX (Accumulator)
- BX (Base)
- CX (Count)
- DX (Data)

- two index registers

  - SI (Source Index)
  - DI (Destination Index)

- two base registers

  - BP (Base Pointer)
  - BX (Base Register -- this is also a general purpose register)

- two special pointer registers

  - SP (Stack Pointer)
  - IP (Instruction Pointer)

- four segment registers

  - CS (Code Segment)
  - DS (Data Segment)
  - ES (Extra Segment)
  - SS (Stack Segment)

- a flags register, with nine flag bits in a 16-bit register

  - CF (Carry Flag)
  - PF (Parity Flag)
  - AF (Auxiliary Carry Flag)
  - ZF (Zero Flag)
  - SF (Sign Flag)
  - TF (Trace Flag)
  - IF (Interrupt Flag)
  - DF (Direction Flag)
  - OF (Overflow Flag)

All of the above registers are 16 bits in length. Each of the general purpose registers can also be addressed as two 8-bit registers. Both the high (more significant byte) and low (less significant byte) can be addressed separately. The 8-bit registers have the same initial letter as the 16-bit register they come from, with H or L indicating high or low. That is, AH is the high byte of AX, and CL is the low byte of CX. Certain of these 8-bit registers have special uses, in particular AH, AL, and CL.

Arithmetic and logical operations may only be performed on eight of the 16-bit registers, and on the 8-bit registers. The 16-bit registers that can be manipulated are the general purpose registers, the index registers, and BP and SP. CS and IP can only be affected by control instructions (jumps, calls and interrupts), and the other segment registers can only be changed by moving a new value into them.

It is important to note that each of the registers is different from its fellows. Certain 8088 machine instructions require their operands to be in specific registers. Some of these instructions are:

```
MUL    :  assumes data in AX (16-bit) or AL (8-bit)
XLAT   :  assumes translation table address in BX
LOOP   :  assumes count is held in CX
IN     :  assumes port number in DX
POP    :  assumes top of stack address in SP
SCAS   :  assumes address in DI
LODS   :  assumes address in SI
```

(These instructions are provided here as examples of assumed register usage only. You don't need to memorize them or their register assumptions. Those assembly instructions which are important to this course will be reviewed in the 8088 Assembly Language module.)

The BP register is not assumed for any operation in the 8088 architecture, but is invariably used by high level languages for addressing a thing called a stack frame pointer. (We will learn more about the Stack as we go through this course. For now, think of the Stack as a location in memory, in which the contents of registers can be stored for future reloading into the registers.) This stack frame pointing is so important that the 80286 adds two instructions (ENTER and LEAVE) which use the BP as the stack frame pointer, and implement the actions a high level language goes through on entering and leaving a procedure.

This specialized usage of registers may seem somewhat foreign to someone acquainted with other processor architectures, like the System/370. Even in the S/370, however, some registers had special uses, like registers 14 and 15 in a BALR instruction, and many others had conventional uses, like registers 0 and 1 in calls to subroutines. The 8088 has taken this much further, implementing special uses for all the registers in the processor hardware.

Every one of these registers is contained inside the 8088 chip. Memory is located off the chip, and a value held in memory has to be fetched into the chip before it can be manipulated in any way. Registers are much faster to deal with because they are on the chip.

3.4 MEMORY ADDRESSING

The 8088 can address one megabyte of memory. As there are 2 ** 20 bytes in a megabyte this means that the 8088 uses a 20-bit address. To obtain a 20-bit address using 16-bit values it uses a combination of two values. It shifts one of these values left four bits -- this is called the Segment value, and is held in a segment register. The segment value, being shifted four bits left, cannot address individual bytes -- it can only address groups of 16 consecutive bytes (called paragraphs). To resolve the address to the byte level, a second 16-bit value is used -- called the Offset value. The Offset is added to the Segment without being

shifted, meaning that the Offset can address anywhere within 64K bytes above the Segment. For greater versatility, the Offset value may be a single value, or it may be the sum of up to three 16-bit values.

The segment value of an address must be held in a segment register -- CS, DS, ES, or SS.

The offset value may be:

the sum of (choose one to three values):
- a literal value
- a base register (BP or BX)
- an index register (DI or SI)

The conventional way in which a segment and offset address are written is with a colon between them, segment first. For example "SS:SP", or "DS:[DI + BX]". A literal segment address cannot be used by the 8088, but it can be written: 0040:0078. Conventionally, all addresses used by the 8088 are written in hexadecimal; this makes for easy computation of the 20-bit address, as the 4-bit shift of the segment is a single hexadecimal digit:

```
0040:0078 =  0040 shifted 4 bits (1 digit) plus 0078
          = 00400 plus 0078
          = 00478       (a 20-bit address, represented as five digits)
```

Because of the way 8088 addresses are made up, the same location may be addressed in a number of different ways, depending on the segment and offset values used. For example, the address used above (00478) can be represented by a segment value of 0040 and an offset of 0078, or as a segment value of 0000 and an offset of 0478, or by a segment value of 0020 with an offset of 0278.

In 8088 instructions addresses are represented in a number of ways. Some instructions do not use addresses -- they deal exclusively with registers. Some instructions use special addresses -- stack instructions always use SS:SP. The instructions which do address memory use an addressing byte to indicate what form of address to use. This byte specifies the offset address to use, but does not mention the segment. There is a default segment register to use with each form of address, which need not be specified. This default segment register can be overridden using a segment override prefix instruction.

The default segment register depends on the type of instruction. Each of the segment registers may be thought of as addressing a separate section of memory, containing different kinds of information. The SS (Stack Segment) register points to a stack area -- any instruction using SP (Stack Pointer) will use SS (this cannot be overridden). The BP register, when used alone or with a literal displacement, addresses the stack segment. Typically BP might be used to address some temporary variables and procedure parameters on the stack, while the SP always points to the top of the stack. (These temporary variables and procedure parameters are known as the stack frame)

The CS (Code Segment) register points to the current program area. The IP (Instruction Pointer) points to the next instruction to be executed. When an instruction is fetched from memory IP is moved to point to the byte following the memory location of the current instruction. Unless the current instruction modifies IP, its contents are interpreted as the next instruction.

The DS (Data Segment) register points to the current data area. Any address involving BX, SI, or a sum of registers, with or without a literal displacement, is assumed to lie in the data segment.

The ES (Extra Segment) register can be used wherever an extra segment register is required to address data. The only offset which defaults to the extra segment is DI, with or without a literal displacement. ES is particularly important when using string instructions.

There are two ways to address data which lies outside of this model: either alter a segment register, or use a segment override. For example, to address a location which lies in the code segment, at an offset of 0222, you may load the segment value of CS into DS and use an offset of 0222 (defaulting to the DS register), or you may use a segment override of "CS:" to specify that the location may be addressed by the code segment rather than by the data segment.

It is worth mentioning here that 16-bit values are stored in memory with the less significant (ie. low) byte first. This allows the less significant byte to be fetched before it is known whether the value is 16-bit or 8-bit. It does not affect the use of 16-bit values, just the appearance of memory when examined using a debugging tool of some kind -- the two bytes of a 16-bit value will appear transposed.

3.5 8088 INSTRUCTION FORMAT

The format of 8088 instructions is:

opcode	[addressing byte]	[displacement]	[immediate operand]
1 byte	1 byte	1 or 2 bytes	1 or 2 bytes

An instruction can be anywhere from one byte to six bytes long, depending on its opcode and addressing byte (where present). The opcode specifies what the instruction is, be it a movement of data or an I/O operation. Instructions may involve registers and/or memory; this is specified by the addressing byte. An address which involves the addition of a literal value will involve a displacement. An instruction which uses a constant value will have an immediate operand.

With the exception of the string instructions no instruction can operate on two memory values; if two operands are involved one of them must be a register or immediate value. Two register operands can be used. One of the operands is termed the source, the other the destination. The oper-

ation is performed on the source and destination values, and the result is placed in the destination location, be it register or memory location.

The following paragraphs provide details on a bit-by-bit basis for how machine language instructions on the 8088 are constructed. Students need not memorize the usage of the bits of an instruction if they do not wish to; it is not required for an understanding of later topics. The important point to retain from this is that various bits in the instruction determine the operand types (register or memory), the addressing techniques, and the operand lengths (byte or word).

Where an instruction may apply to either 16-bit or 8-bit values the choice is indicated by the lowest bit of the opcode (1 = 16-bit, 0 = 8-bit). This is called the w-bit.

Where an instruction can operate in either direction, that is, where either of the two operands may be the destination (not applicable with immediate operands, for example), then the destination is indicated by the second-lowest bit of the opcode. This is called the d-bit.

Where an instruction involves immediate data, the second lowest bit indicates whether the immediate value is a 16-bit value, or an 8-bit value which should be sign-extended to produce a 16-bit value. This is called the s-bit, and is located in the same spot as the d-bit. It is ignored if the w-bit indicates an 8-bit operation.

The addressing byte is split up into three pieces:

Mode	Reg	R/M
2 bits	3 bits	3 bits

The mode value indicates how to interpret the R/M value. The reg value indicates which register is one of the operands. The R/M value indicates which register or combination of registers is the address of the other operand.

The mode value can take on four values:

00 = no displacement is added to the R/M value

01 = a one byte displacement (-128 to +127) is sign extended and added to R/M

10 = a two byte displacement is added to R/M

11 = register addressing -- R/M is a register, not a memory operand

The reg value can take on eight values, indicating one of eight registers. The w-bit determines from which set of registers the choice is made:

Reg	w-bit = 1 (16-bit)	w-bit = 0 (8-bit)
000	AX	AL
001	CX	CL
010	DX	DL
011	BX	BL
100	SP	AH
101	BP	CH
110	SI	DH
111	DI	BH

The R/M value can take on eight values, indicating a choice of addressing modes modified by the mode value. The effect is:

R/M	Mode = 00	Mode = 01	Mode = 10	Mode = 11
000	BX+SI	BX+SI+d1	BX+SI+d2	AX or AL
001	BX+DI	BX+DI+d1	BX+DI+d2	CX or CL
010	BP+SI	BP+SI+d1	BP+SI+d2	DX or DL
011	BP+DI	BP+DI+d1	BP+DI+d2	BX or BL
100	SI	SI + d1	SI + d2	SP or AH
101	DI	DI + d1	DI + d2	BP or CH
110	direct *	BP + d1	BP + d2	SI or DH
111	BX	BX + d1	BX + d2	DI or BH

* : "direct" is a literal address within the DS segment. There is no offset register used, just the displacement.
d1: a single byte displacement sign-extended to two bytes before adding.
d2: a two byte displacement.
choice of AX or AL (when mode = 11) is controlled by the w-bit.

Note that the same offset registers are used for a given value of R/M for each of the mode values 00, 01, 10. With a mode value of 11, the R/M value indicates registers in the same way as the reg value.

The w-bit does not affect the memory addresses (mode values 00, 01, 10), it simply affects the number of bytes taken from memory at the resulting address.

The 8088 instruction set has a few irregularities. A number of operations can be coded with shorter instructions if the accumulator (AX if 16-bit, AL if 8-bit) is used rather than another register. Certain operations are limited to specific registers (multiplication and division, input/output, string operations, and some data translation operations). These limitations were imposed to provide a powerful instruction set without incurring overly complex chip design problems. These irregularities simply increase the challenge of programming the 8088.

A few of the 8088 instructions do not execute immediately. They alter the instruction which follows them, and so are called prefix instructions. One of them has been mentioned already -- the segment override prefix. A segment override prefix alters the segment register to be used in the memory address of the next instruction. Another prefix instruction is the LOCK prefix -- it locks the bus during the execution of the next in-

struction, and is usually use in a multiprocessor environment. The most powerful prefix instruction of all is the REP prefix. This specifies that the following instruction be executed until the CX register is zero -- it decrements CX before testing it each time. The REP prefix is not particularly useful for most instructions, because reexecuting most instructions is a waste of time. String instructions, however, combine very nicely with the REP prefix to produce extremely powerful results.

The instruction format in the 8088 is "tight", packing a lot of function into its opcodes. At the same time, it is a rich machine language, with all manner of powerful and useful instructions.

3.6 INPUT/OUTPUT

The 8088 microprocessor supports 65,536 I/O port addresses. I/O ports resemble ordinary memory in that they can be read and written, but otherwise they are quite different. An I/O port is used to control or use an external device. Writing to an I/O port that is part of an asynchronous communications adapter may cause the data written to be transmitted to a modem. Reading from an I/O port that is part of a diskette adapter may read consecutive bytes from a diskette sector. Unlike memory, consecutive reads from the same I/O port are quite likely to read different data.

The I/O ports of the 8088 are completely separate from memory. They require special instructions to read and write them (IN and OUT respectively). Data may be read or written either one or two bytes at a time, although few devices support two-byte transfers. An IN instruction reads from an I/O port and places the data in the accumulator (AL for one byte, AX for two). An OUT instruction takes data from the accumulator and writes it to the specified I/O port. The I/O port address may be specified either literally (if the port is from 0 to FF), or by placing the address in the DX register (port address from 0 to FFFF).

I/O ports are only one of the ways in which the outside world can communicate with the 8088. Two other ways are memory mapping, and DMA.

Memory-mapped I/O devices look exactly like memory to the 8088. The most common is a memory-mapped display. The display accesses the memory to produce a screen image. Whenever the 8088 alters the contents of the screen memory, the display reflects the change. This provides a fast and effective screen interface, but is not particularly useful for other peripherals. We will examine it in more detail in the PC Architecture module.

DMA (Direct Memory Access) allows devices other than the processor to read and write memory directly. The DMA device writes to memory by locking the 8088 out for a few cycles -- this technique is referred to as cycle stealing. DMA can be used to transfer large amounts of data without processor intervention -- once the processor has set up the parameters within the DAM controller, the DMA will take care of the I/O itself. DMA

is a more expensive I/O method to implement than using I/O ports, but it offers considerably more power. It is used by fixed disks.

3.7 TRANSFER OF CONTROL

The 8088 supports three different ways of transferring control from one piece of a program to another. These three control transfers are termed Jump, Call and Interrupt. Each of these is applicable in a different context, and a typical program will probably use all three. The way in which they all effect the transfer of control is by altering where the CS:IP combination points, that is, by altering the pointer to the next instruction. The jump may be any of three different "distances", where "distance" refers to the relative address range within which the jump can occur. The three possible distances are Short, Near and Far. Short transfers can transfer control within 128 bytes from the current instruction, up or down; Near transfers can move within the current code segment, and Far transfers can move anywhere in memory.

The call may be either of two distances, Near or Far; an interrupt is always Far. The different distances require different lengths of instructions -- for example, a near jump requires three bytes of instruction, while a far direct jump requires five.

Jump: A jump simply alters IP, the instruction pointer, and sometimes CS, the code segment, as well. There is no record of where the jump came from. The three jump distances are described below:

- Short -- a short jump alters IP by adding a signed 8-bit value to it. Thus a short jump may move IP by -128 to +127.

- Near -- a near jump loads a new value for IP. Thus a near jump may move IP to anywhere in the current code segment.

- Far -- a far jump loads a new IP and a new CS value. Thus a far jump may move CS:IP to anywhere in memory.

Call: A call saves the current instruction pointer on the stack before altering IP, and sometimes CS. This enables the program to return to the current spot after executing the called code. The return is effected by a RET instruction, which must match the CALL. There are two different forms of call:

- Near -- a near call stacks only the current IP value, and loads a new value into IP. A near call must be matched with a near return, which restores the IP value from the stack. A near call works within the current code segment.

- Far -- a far call stacks both CS and IP, then loads new values for both of these. A far call must be matched with a far return, which restores both IP and CS from the stack. A far call works anywhere in memory, provided the address of the code to be called is known.

Interrupt: An interrupt saves the current flag register as well as the instruction pointer and code segment register before altering CS:IP. This enables the complete state of the machine to be saved. An interrupt service routine can return to interrupted code and continue execution no matter what was being done when the interrupt occurred. There are two kinds of interrupt in the 8088 -- hardware (external / involuntary), and software (internal / voluntary). Both function in exactly the same way: the flags register, then CS, then IP are saved on the stack, then CS and IP are loaded from the appropriate interrupt vector. All interrupts may be considered Far. The return from an interrupt is achieved with an IRET, which restores IP, CS, and the flags register from the stack. Interrupts are identified by interrupt numbers. Each interrupt number refers to a separate entry in an interrupt vector table; each entry in this table contains a 4-byte segment:offset address which the 8088 uses to find and invoke the interrupt routine. This interrupt vector table is stored at the bottom of memory (locations 0 to 3FF). The interrupt number, when multiplied by four, gives us the offset into the interrupt vector table at which the interrupt routine address can be found.

The different means for transferring control are used in different contexts. Typically, the jump is used in the control of program flow, while the call is used to address functions and subroutines of the executing program. Software interrupts are usually used to access system services, such as device drivers. Hardware interrupts are generally associated with the functioning of peripherals, such as hard disks and communications adapters.

3.8 IMPACT OF 8088 ARCHITECTURE ON PC ARCHITECTURE

The choice of the 8088 as the CPU for the IBM PC had a dramatic effect on the architecture of the PC, dictating its memory range, approximate speed, and general performance characteristics. At the same time the designers had a lot of latitude in the way in which they could build a PC around the 8088. They chose a structure which was simultaneously quite versatile, and a little restrictive, so as to guide third party manufacturers into a path that would lead to the widest possible compatibility of diverse hardware. This is the open architecture of the PC, the part of the announcement in 1981 that surprised more people than anything else about the PC.

3.9 SUMMARY

This concludes our module on the 8088 microprocessor. At this point you should understand the following:

- the development of the Intel 8088, and its successors
- the internal structure of the 8088

  - the Bus Interface Unit
  - the Execution Unit
  - the registers

- the registers of the 8088 and their uses

  - the general registers AX, BX, CX, DX
  - the base registers BX, BP
  - the index registers DI, SI
  - the pointer registers BP, SP, IP
  - the segment registers CS, DS, ES, SS
  - the flags register, flags: AF, CF, DF, IF, OF, PF, SF, TF, ZF

- the memory addressing scheme of the 8088

  - segments and offsets
  - default segment registers and overrides
  - forming of offsets from combinations of registers

- the format of machine instructions for the 8088

  - the opcode, with the w-bit, d-bit, s-bit
  - the addressing byte, with the mode, reg, and r/m fields
  - displacement
  - immediate value
  - prefix instructions

- I/O port addressing in the 8088

- the various means of transferring control in the 8088

  - jump, either short, near or far
  - call, either near or far
  - software interrupt

If you have not grasped any of the above concepts then it is strongly recommended that you review the relevant section again, as many of these concepts are essential to the understanding of later parts of this course. You may also wish to consult an IBM Workstation Specialist who is familiar with the 8088, for additional assistance. When you feel you can meet all the objectives identified at the beginning of this module you should proceed to the PC Architecture module, which builds on this background to discuss how the IBM PC is built around the 8088 microprocessor.

4.0 PC ARCHITECTURE

If we consider the 8088 to be the engine of the Personal Computer, we must be familiar with its operation to understand how the car works; however we will also need to understand how the petrol supply and exhaust are hooked up to it, how the wheels connect to their axles, how the windows roll up and down, and how many doors are on the car. We could drive this analogy many miles more; the point is that while the Personal Computer does revolve around the 8088, other components in its design are equally complex, and are equally important in improving our understanding of PC internals.

By the time you complete this section you should understand the PC architecture extensively, by meeting the objectives identified below. You should be able to:

- distinguish between an architecture and an implementation
- describe the elements of the PC architecture
- define and describe the important parts of the PC:
  - Memory
  - Video
  - Keyboard
  - Interrupts
  - I/O Ports
  - DMA
  - Timers and System Clock
  - System Bus
- describe the considerations involved in designing for the PC
- be ready to learn about the BIOS that completes the PC design

You will need to understand the materials covered in the preceding module, which dealt with the 8088 microprocessor. If you do not have at least some knowledge of the 8088, go through the previous chapter again.

4.1 IMPLEMENTATION VS ARCHITECTURE

An architecture is quite separate from pieces of hardware or software. Frequently a piece of hardware or software is said to conform to an architecture -- this means that it follows the rules of the architecture. Sometimes it may even be said to be an implementation of that architecture. A computer architecture is an abstract concept, generally designed so that future expansion will be simple and clean. Sometimes the architecture is designed along with the first implementation, which can result in the architecture being contaminated by practical considerations. An implementation quite often does not exploit the full capabilities of the architecture, for pragmatic reasons.

The IBM PC can be considered an implementation of the Intel 8088 architecture, in that it is a hardware realization of the possibilities in-

herent in the 8088 microprocessor. At the same time, the IBM PC can be considered an architecture of which the PC, the XT, and the AT, are all implementations. The IBM PC architecture is more limited that the 8088 architecture in some aspects, but considerably more detailed in others. Because of the greater detail in the PC architecture it is possible to design hardware and software which will work with the entire range of PC products, something that cannot be said for the 8088 architecture.

This discussion of the PC architecture will ignore individual machine differences, but rather concentrate on the unifying identity between the machines.

4.2 HARDWARE ARCHITECTURE

The IBM PC is more than just an 8088 surrounded by peripherals, and so the architecture of the PC has to cover more than just the 8088. The things that distinguish the PC from other 8088 based computers lie, in part, in the other hardware on the system board, the support hardware. Included in this is the 8259 interrupt controller, which turns hardware interrupts received by I/O devices into software ones which the 8088 can process; the 8237 DMA controller, the 8254 timer, and the 8042 processor which controls the keyboard. The complete layout of the system board for each of the members of the PC family can be found in the appropriate Technical Reference manual, but all members have at least the functions described below (with the exception of the PCjr, and the JX, neither of which can be considered full PCs due to their lack of DMA logic.).

The 8088's one-megabyte memory address space is split into two pieces in the IBM PC. The lower 640K is used for RAM, with installed RAM starting at address 00000; installed RAM must be contiguously addressed for it to be checked by the automatic memory tests. The upper 384K of address space is used for system purposes. The first 128K (A0000 to BFFFF) is allocated to video RAM; so far the EGA uses A0000 to AFFFF for its native mode graphics, the Monochrome Display and Printer adapter (MDPA) uses 4K starting at B0000, and the Color Graphics Adapter (CGA) uses 16K starting at B8000. Adapters may have BIOS extensions in ROM modules; these may be anywhere in the region C8000 to DFFFF, and we will discuss these in detail in the BIOS module. The addresses from E0000 to EFFFF are reserved for I/O adapter ROMs in the PC, XT, and Portable PC, but for system board ROMs in the AT. Addresses F0000 to FFFFF are used by the system BIOS ROM, and ROM Basic. Apart from the system identification byte (located at FFFFE) and the BIOS ROM date (located at FFFF5 to FFFFC), none of the addresses above the video RAM should be of particular interest to a programmer, because the addresses should not be used. All reference to the BIOS should be made using software interrupts, which provide a consistent interface across the entire PC family. The video RAM layout is consistent within a given video mode, but will vary from one mode to another.

The full 8088 I/O port address space is available in the PC - all 64K port addresses may be used, but the addresses below 100 are reserved for the system board, and many of the addresses up to 3FF have been assigned to

specific adapters. Addresses not currently used may be reserved for new adapters, or may be free for use by other I/O devices. The complete map of used ports may be found in the Technical Reference manual.

The PC has eight hardware interrupt levels on an Intel 8259 interrupt controller. An interrupt level can be thought of an access line through which an I/O device can gain the system's attention. Two of these interrupt levels are fixed in purpose: level 0 is used for the system timer, and level 1 serves the keyboard. The other six levels are available on the I/O bus for use by adapters. Many of the I/O adapters are interrupt driven. The map of interrupt levels used by different adapters can be found in the Technical Reference manual. In some cases several adapters use the same interrupt level; this can cause problems when two such adapters are installed in the same PC. The new PC AT Technical Reference manual describes a hardware and software solution to sharing interrupt levels. A hardware interrupt is usually used to indicate the completion of some form of I/O processing: a disk may interrupt to indicate completion of a seek operation, or an asynchronous communications link may interrupt to indicate that another character has been received.

The PC has four DMA channels on an Intel 8237 DMA controller. DMA allows data transfers to proceed without the intervention of the 8088. A channel of the 8237 is programmed to transfer data between memory and I/O adapters, and the transfer takes place with the DMA controller "stealing" bus cycles from the 8088. Up to 64Kb can be transferred at a time. One DMA channel is dedicated to memory refresh. The other three are available on the I/O bus. A number of I/O adapters use DMA; a list of what channel is used by what adapter can be found in the Technical Reference manual.

The PC supports the full range of interrupt numbers of the 8088. Interrupts can be from 00 to FF, but interrupts 00 to 1F are reserved for system hardware functions, including the system BIOS interface. Interrupt number 20 to 3F are reserved for operating system functions (DOS uses 20 to 27). 40 to 5F, and 70 to 77, are reserved for BIOS enhancements. 60 to 6F are available for use, as are F1 to FF. 80 to 85 are reserved for Basic, and many of 86 to F0 are used by the Basic interpreter while it is running.

Let's refresh our memory on interrupt vectors, as these are a central component of the PC architecture. The interrupt vector table is located at the start of system RAM. The location of the interrupt vector for a given interrupt number can be obtained by multiplying the number by four. The word at that location is the IP value of the entry point of the interrupt service routine; the CS value for the entry point is located immediately after the IP value. This means that the first 400H bytes of RAM are used for the interrupt vector table.

4.3 SOFTWARE ARCHITECTURE

The IBM PC has more than just a hardware architecture. Built into the PC is a considerable quantity of software, held in ROM to provide the

machine with a power diagnostic system, a self booting facility, and a device-independent programming interface. This is the PC Basic I/O Subsystem (BIOS), and the subject of a later module. On top of the BIOS, most PCs are running PC DOS as their operating system, providing yet another layer of architecture.

4.4 HARDWARE DETAILS

The details of a number of the elements making up a PC will be considered in turn. The elements to be considered are:

- Memory - both RAM and ROM
- Video - the screen you see things happen on
- Keyboard - the way you make things happen
- Interrupts - the way the hardware gets service
- I/O Ports - the way much hardware talks to the system
- DMA - the way some devices transfer blocks of information
- System Bus - about which the PC revolves
- Timers and System Clock - that make the PC tick!

4.5 RAM

The RAM in a PC can be any of a wide number of kinds, and different kinds can be mixed in one PC. Any size of module from 16K to 256K can (and has been) used. The most common varieties of RAM used are all dynamic RAMs. Dynamic RAM has two major advantages, and one serious disadvantage. The advantages of dynamic RAM lie in its high density and low price. The disadvantage lies in that it must be refreshed. Dynamic RAM cannot hold the values it is asked to remember for very long - it starts to forget. To prevent this it must be read from or written to on a regular basis, several times a second. The PC uses one counter of the system timer to initiate RAM refreshes at a preset rate, and one channel of DMA to request dummy transfers from RAM. These dummy DMA transfers access the RAM, thereby keeping the data in it alive. Any application which locks out interrupts for an extended period can interfere with the refresh of memory, and can cause a loss of information.

A bank of RAM in a PC typically consists of nine dynamic RAM chips, all the same kind. Each chip holds the same number of bits. In the case of 64K chips, each chip holds 64K bits (not bytes). Eight of the chips provide one bit in every byte of the 64K bytes the bank holds. The ninth chip provides a parity bit for each byte. Thus the data being stored in the bank of memory may be considered to be stored "across" the chips, rather than one byte wholly in one chip, and another wholly in another. Parity is calculated by counting the number of "1" bits in a given byte value, and setting the parity bit to ensure that, including the parity bit itself, there are an even number of ones. If the byte value was 11001100 the parity bit would be 0 since there are four ones in the byte.

If the byte were 01001100 the parity bit would be a 1 to make the total sum of 1's into an even number. Every time information is read from RAM the parity of the value read is checked. If an error is found it is immediately reported by generating a Non-Maskable Interrupt. The default action to take at this point is to display the dreaded "PARITY CHECK" error message, together with a code indicating the bank in which the error occurred.

Read Only Memory: Every PC contains at least two ROMs - the system BIOS ROM and the ROM containing Cassette BASIC. The system BIOS ROM is located at the high end of memory, in the F0000 segment. There are three locations in the system ROM which are worthy of notice: FFFF0, FFFF5 and FFFFE. FFFF0 is where the 8088 starts executing instructions when it is powered on. In the IBM PC this location holds a FAR JUMP instruction, which jumps to the start of the Power-On Self Test. Immediately following this jump instruction in memory (at FFFF5) is the date of the system ROM. This date is important occasionally, when determining the compatibility of certain hardware with the PC. At FFFFE is a single byte which identifies the type of PC. This can be used by software which is sensitive to the kind of PC in use. The currently defined codes are:

```
XT, PPC, 3270 PC
         FD = PCjr, PC JX
         FC = PC AT, 3270 PC AT
```

It is possible to build an I/O adapter which contains its own BIOS code. Such code can make the adapter much easier to interface to DOS, or can take advantage of special features of the adapter. Some of the adapters that have ROM BIOS code on them are: the Fixed Disk Adapter, the PC Network Adapter, and the Enhanced Graphics Adapter. In each case the ROM containing the BIOS extension is located in the memory map between C8000 and DFFFF. These optional ROM BIOS modules are discussed in the BIOS module.

This ability to provide additional ROM BIOS support to a PC, built directly on an adapter, is one of the more powerful innovations of the PC architecture. It allows the BIOS interface to be customized, or even replaced (as in the EGA), to suit the installed hardware. An excellent example is the Fixed Disk Adapter, which modifies the system BIOS so that it attempts to IPL from the first fixed disk after failing to IPL from the first diskette drive. A PC without a Fixed Disk Adapter will only try the first diskette drive.

4.6 VIDEO

The video support of the IBM PC is based on a memory-mapped approach to the screen. The two original display adapters for the PC -- the Monochrome Display / Printer Adapter (referred to here as the MDPA) and the Color Graphics Adapter (referred to here as the CGA) both use a simple scheme to map the screen buffer into PC memory. The more recent Enhanced

Graphics Adapter has to adopt a more elaborate method to be able to cope with the large amount of memory it uses.

The 128K region of the PC memory space from A0000 to BFFFF is allocated to video RAM. The EGA uses 64K of this space, starting at A0000. The MDPA uses 4K, starting at B0000, and the CGA uses 16K starting at B8000. The memory layout for the text modes is the simplest, so we will start with that.

All the text modes of the PC video systems use the same kind of memory layout. Starting at the top left corner of the screen, which is termed (0, 0), the positions on the screen are given row and column coordinates. On the MDPA and the CGA the row number ranges from 0 to 24, and the column from 0 to 79. The EGA supports considerably more rows, depending on the character size selected. Every character on the screen has an attribute byte, which determines the color of the character on the screen. The character itself is stored in memory followed by its attribute. Row 0 is the first to be stored, followed by row 1, and so on. Within each row the characters are stored in order. Thus the screen layout can be considered to be what might be obtained by placing row 1 at the end of row 0, then row 2 at the end of that, and so on.

The attribute bytes used in text modes are oriented towards the CGA. The highest bit controls whether the character will blink. The next three bits are the background color, with one bit for each of the Red, Green and Blue components of the color. This gives a total of eight possible background colors. The foreground color is determined by the low four bits of the attribute byte. The lowest three bits are again the Red, Green and Blue components, while the highest bit of the four is the Intensity bit, providing a choice of high or low intensity foreground colors. This gives a total of 16 possible foreground colors.

On the MDPA the blink and intensity bits are supported, and the remainder are operational to a very limited extent. Setting just the Blue bit of the foreground color, with the Red and Green bits clear, causes the MDPA to underline the character. Setting all the bits of the background colour causes the MDPA to display in reverse video. Clearing all of the RGB bits of the foreground colour cause the character to be invisible, unless in reverse video. Any other combination of RGB bits displays the character in living green on a dark background. This compatibility between CGA and MDPA attributes has led to many programs using the same attribute for both in an effort to simplify the programming task. This approach works well everywhere except when using an EGA to emulate an MDPA, where certain attributes can render the display unreadable.

Each graphics mode uses a different screen buffer layout, but there are similarities. All of the CGA graphics modes display 200 lines of pels, organized on an odd/even system - the odd numbered lines are held in the upper half of the screen memory, while the even numbered lines (including 0) are held in the lower half of memory. The reason for this complicated layout lies in the engineering of the CGA. In 320x200 resolution the CGA displays lines of 320 dots, termed picture elements, or pels. Each dot may be one of four colors. To indicate which of the four colors a given dot is, it is represented by two bits. Four pels are represented in a

byte of screen buffer, so the whole 320x200 require 16K of memory. The background color can be any of the 16 colors available on the CGA, because the background color setting is held in a register on the CGA card, called a palette register. The other three colors are fixed, except that a choice can be made between cyan/magenta/white and green/red/yellow.

The higher resolution mode of the CGA displays 640 pels on each row. Each pel can be either colored or black, so the CGA places eight pels in a byte of screen buffer, again using all of the 16K available. The palette register determines the foreground color, so it can be any of the 16 available.

The EGA uses a more sophisticated screen buffer system. It has four distinct "planes" of screen memory, with each plane mapped into the same memory addresses as far as the system is concerned. This might seem to indicate that the EGA must have the same value written into all of the planes whenever the buffer is addressed, but this is not so. The planes can be individually masked, so a given write can access any or all of the planes without affecting those masked off. Again, a specific 4-bit color value can be written into those locations the system attempts to set. This allows the writing of large amounts of a given color quite easy. The EGA is an extremely complex piece of hardware, and anyone wishing to program it directly is referred to the Technical Reference manual for details. The complexity of the EGA is increased by the sophistication of the support it offers for the older displays -- the Monochrome Display and the Color Display. To realize the full power of the EGA the Enhanced Color Display must be used. The EGA replaces the system board video BIOS interface, adding extra functions to support the host of features available on the EGA.

The EGA has sixteen palette registers, each addressed by a different combination of bits in the screen memory planes. Each palette register holds a six bit value, with two bits for each of the Red, Green and Blue components. Two bits gives the ability to use four different levels of each color, and a total of 64 different color settings. Only sixteen of these colors can be used at any time, but the choice of which sixteen is completely open.

4.7 KEYBOARD

The keyboard on the IBM PC is more than just a collection of buttons. The keyboard contains an Intel 8042 microprocessor, which scans for keys that are held down. When the 8042 finds a key down it transmits a "make" scan code to the system unit. This code informs the system unit that the key has been pressed down. After a preset delay the typematic function of the 8042 takes over, and starts transmitting repeated "make" codes of the last key pressed. This continues until the key is released, at which time the 8042 transmits a "break" scan code, indicating that the key has been released.

It may seem strange at first sight to transmit a code both for the pressing and the releasing of a key, but the reason is simple. It allows any key to be used as a shift key, without reprogramming the 8042 (something which is quite difficult, because its programming is fixed!). When the system unit receives a "make" code for a shift key it sets a flag indicating that the shift key is down. Any scan code received after that is treated as shifted, until the "break" code for the shift key is received. The game "Styx" (also known as "JX Labyrinth") uses this technique to treat F1 as a shift key, while the IBM PC Network SNA 3270 Emulation package treats F4 as a special purpose shift key.

The ROM BIOS normally handles all the conversion of scan codes into characters. Whenever the 8042 interrupts the system with a scan code interrupt 9 is invoked. The interrupt 9 service code processes the scan code. We will examine details of the interrupt 9 service routine in the BIOS module.

4.8 INTERRUPTS

The PC provides 8 interrupt levels, using the Intel 8259 interrupt controller. Two of the levels are sequestered for use on the system board, by the system timer (level 0 -- highest priority), and by the keyboard (level 1 -- second-highest priority). The other six levels are available on the system bus for use by adapters.

The interrupt controller is programmed for strictly prioritized interrupt handling. That is to say, an interrupt on level 3 will always be serviced before one on level 4, assuming that both occurred at the same time.

Each of the interrupt levels corresponds to an interrupt number. The PC has allocated interrupt numbers 8 to F to hardware interrupt levels 0 to 7 respectively. When a hardware interrupt occurs the 8088 will stop on completing the currently executing instruction. It saves the current flag register on the stack, then the current CS and IP values (as for a CALL), before loading the CS and IP from the interrupt vector.

The interrupt vector table is located at the lowest point in memory, address 00000. Each interrupt vector is four bytes long (two for CS and two for IP). Thus the first entry is at address 00000, the second at 00004, and so on. The full 256 interrupt vectors occupy addresses 00000-003FF.

As well as being used for hardware interrupts, the interrupt vector table can be used by software interrupts. A software interrupt is a special instruction which tells the 8088 to behave as though an interrupt were occurring. The software interrupt specifies what interrupt number to use. This mechanism allows the PC to treat some of the software interrupts as a standard means for programs to use common routines. Both BIOS and DOS use interrupts as their means of invocation. That is to say, the interrupt vector table is used as a table of global routines.

Using an interrupt vector for access has a number of advantages. It eliminates any memory dependency of one program on another. It allows the routines to be replaced, if necessary, either by ROM on a new adapter (as in the EGA), or by a piece of software.

4.9 I/O PORTS

The I/O ports in a PC are used to control and exchange information with all manner of peripheral devices. The I/O ports are addressed on the bus in the same manner as memory, but with fewer bits in the address being significant, as the 8088 will only address up to 65,536 ports. When an I/O port is addressed instead of a memory location the I/O Read or I/O Write control signal on the bus is active, as opposed to the Memory Read or Memory Write. The 8088 will read from a port using the IN instruction, and write to one using the OUT instruction.

I/O ports are often registers located on adapters. They may be used to report the status of an adapter (ready for input, busy, etc), to control the adapter (set speed, size, etc), or to transfer data -- a register may provide the input or output buffer to a device. Because an I/O port is performing such functions it is not unreasonable for the value read from a port to vary without the CPU altering it -- in contrast to ordinary memory, which is normally not changed except by the CPU.

4.10 DMA

Direct Memory Addressing (always termed DMA) is a fast and efficient way to transfer blocks of data between I/O ports and memory. It is always used by the fixed disk, and usually by diskette, for the transfer of sectors of information to and from the device port. DMA operates without involving the CPU in the transfer. This allows the CPU to continue with other processing during the transfer. The keyboard can be used, and the time of day updated, without the disk transfer being affected.

The PC has a DMA controller chip (Intel 8237), and a number of DMA control registers to support DMA. The controller supports four channels of DMA, each of which is separately programmable. Channel 0 is used by the RAM refresh mechanism, and so is unavailable for use by adapters. The other three channels are available, with the diskette controller using channel 2 and the fixed disk channel 3, when disk I/O is going on. Unlike hardware interrupt levels, it is possible to share DMA channels, providing that only one device is actually using the channel at a time.

The DMA controller does not support segmentation of memory. To get the full 20-bit address needed to address 8088 memory the DMA subsystem uses page registers. A channel's page register is a 4-bit register that supplies the top 4 bits of the 20-bit address. The remaining 16 bits are provided by the DMA channel's address register. Forming the 20-bit ad-

dress in this way restricts the use of large block sizes. In normal 8088 memory addressing it is possible to start a 64K segment on any paragraph (16-byte) boundary. The page register approach is restricted to the use of 64K boundaries. This is the cause of the infamous FORMAT problem -- if the FORMAT utility is loaded into memory in such a way as to make its buffer overlap a 64K boundary it will fail. Moving the load point will make it work again. The FORMAT program doesn't check for this condition.

When a DMA channel has been told which I/O port to use, where in memory to find or store the block of data, and how many bytes to transfer, the DMA transfer can be started. For each DMA cycle the controller will transfer one byte. After every DMA cycle the 8088 is given a chance to access memory. When the transfer is complete the status for the channel changes to free.

4.11 TIMERS AND THE SYSTEM CLOCK

The PC is driven by the system clock. Every operation inside the PC is synchronized to the clock. Memory accesses to RAM take exactly four clock cycles. I/O port accesses take five. An ADD instruction between two registers takes four clock cycles. The system clock in the PC is centered on a crystal. A crystal is a carefully cut piece of quartz enclosed in a metal case, with two leads coming out of it. The frequency of oscillation of a crystal is extremely precise -- your quartz watch uses a crystal to keep time. The PC and XT use a 14.3 MHz clock crystal, and divide its frequency by 3 to derive a system clock rate of 4.77 MHz. That means 4.77 million clock cycles per second. The AT uses a 12 MHz crystal, and divides it by 2 to get a system clock rate of 6 MHz, significantly faster. A higher clock rate means that the PC is operating faster, but there are limits. The PC is based on the 8088 processor, which until fairly recently was only designed to operate at speeds up to 5MHz. The 80286 processor used in the AT is specified to run at 6 MHz, but there are faster 80286s out now -- 8MHz and even 10 MHz are said to be readily available; there is even speculation on a 12 MHz version. The processor is not the only item in the PC that is speed dependent -- the RAM and ROM chips can only operate so fast. The PC uses 200ns RAM, which will operate quite comfortably at 4.77 MHz. Raise the speed to 8 MHz and you need 120ns memory or better. Faster RAM chips are available, but they get more expensive the faster you go. Many other components are speed critical -- the old Asynchronous Communications Adapter will not operate reliably in an AT because it cannot respond quickly enough.

The PC uses a single timer chip to provide a number of timer functions. The most critical of these is the refresh timer, which is used to generate the RAM refresh requests. Almost as important, the system timer, which runs the time-of-day clock, is used for timing for diskette access, and is central to any attempt to provide a form of multitasking for the 8088. The third timer/counter in the PC is used for sound generation in the built-in speaker. All three timer/counters are driven by a 1.19 MHz signal. The 1.19 MHz signal is derived from the system clock in a PC or

XT -- the 4.77 MHz is divided by four. The AT uses a separate crystal to produce the timer signal.

The system timer has been programmed on the PC to count from its maximum (65536) to zero before interrupting. This gives an effective interrupt frequency of 18.16 Hz, that is a little over 18 times per second. When it interrupts it invokes interrupt 8, which updates the time of day counters. The service routine for interrupt 8 invokes interrupt 1A, which is the point at which timer-driven routines should intercept the interrupt.

The refresh timer operates every 15 microseonds, with 256 refresh cycles required every 4 milliseconds. Interfering in any way with this channel of the timer chip is likely to destroy the currently running system, because RAM that is not being refreshed often enough is unreliable.

The timer which drives the speaker can be programmed to divide by any number from 1 to 65536. This allows the programmer to choose the frequency of the tone the speaker will generate. For example: dividing by 2705 will give a frequency of 440 Hz -- a concert A, while dividing by 4648 will give a frequency of 256 Hz -- a middle C. There is an I/O port address which can be set or cleared to switch the speaker on or off. If you turn it on you had better have a small program available to turn it off -- a constant tone quickly becomes deafening!

4.12 BUS

The bus of the PC is the interface between the system board and adapters. It consists of 62 lines in parallel across a number of circuit board edge connectors. The lines on the bus include 20 address lines, 8 data lines, 3 DMA request and acknowledge lines, 6 interrupt request lines, and a number of power, ground, and control lines. The address lines are separated from the data lines before being placed on the bus (they are multiplexed coming out of the 8088).

The bus in the PC is run in maximum mode, permitting the use of multiple processors in the PC. The most common is the use of an 8087 mathematics coprocessor. The bus can be controlled by the 8088, another processor, such as the 8087, or by the DMA controller.

4.13 BUILDING A CARD FOR THE PC

The most foolish thing that a hardware developer can do is to ignore all of the guidelines that IBM has published about the various resources that a card may use. Such an attitude is almost guaranteed to produce a product that will sooner or later become incompatible with some other product on the market. This has been observed on a number of occasions. IBM has made incompatible adapters too -- the worst was the incompat-

ibility of the original 5250 emulation card (for connecting a PC to a System/36) and the XT fixed disk controller.

The resources that a new card may wish to use are:

- Memory addresses (both RAM and ROM)
- I/O port addresses
- Software interrupt/s
- Hardware interrupt/s
- DMA channel/s

Each of these will be discussed in turn. They have been ordered roughly in order of simplest to hardest.

Memory addresses may not seem too difficult an area to address, with a one megabyte address space. And they aren't, because the PC memory map has been fairly well laid out, with explicit allocation of a lot of the address space to specific functions. Generally, manufacturers seem to manage to cope well with the allocation of addresses to their cards, partly because very few peripherals are memory mapped. Very few add-on cards contain ROM extensions, which is strange, because this is one of the best ways to mask the differences of new hardware. Many addresses are available for add-on ROM's, so this is unlikely to be a problem.

I/O port addresses are also documented in the Technical References. Again, there seem to be few clashes in this area, and most cards are designed to use any of a number of port addresses depending on the setting of jumpers on the card. This allows clashes to be resolved by moving the addresses used by one of the cards. This is a useful technique, but only if the software support for the device will support all of the possible choices.

Software interrupts are more limited. Particularly when the number of interrupts used or reserved by IBM is taken into account. There are only 31 interrupts marked "not used" out of interrupts 0 to FFh, with another 8 marked "reserved for user programs". A large number of the rest are marked reserved for various reasons. Several manufacturers have used interrupts that were marked reserved, often without any problems so far. One instance, however, is fairly well-known: the clock component of an AST multifunction card uses an interrupt that is also used by the IBM 3278/9 Emulation adapter -- these two cannot be used together.

Hardware interrupts are a much more serious problem. There are only 8 levels of hardware interrupt in the PC (15 in the AT), and two of these are used by the system to support the timer and the keyboard. A third is used by the diskette drive controller, and yet another for the fixed disk controller, if one is present. The remaining interrupt levels might, theoretically, be shared, but few cards support this facility (the IBM Cluster adapter is one of the few).

DMA channels are the scarcest resource of all. The PC has only four DMA channels (the AT has 7), of which one is required for dynamic RAM refresh, leaving only three. Of these three, one is used by the diskette controller when data transfer is in progress. Conflicts can arise very easily with cards which wish to use DMA.

Both DMA channel and hardware interrupt number should be configurable to help install the card along with other cards. The software interrupt number, and the I/O port addresses should also be capable of being changed, and the software should support all of the possible options. Most cards seem to manage to fit fairly well by providing two choices for each of the necessary items.

4.14 ARCHITECTURAL EXTENSIONS

The IBM PC architecture was not carved in stone in August 1981. As the scope of the product expands it has proven necessary to augment the architecture in many areas.

The first of these to be acknowledged was the need to make video memory layouts a part of the architecture. Because of the slow speed of the BIOS graphics interface many software vendors were forced to address the video memory directly. If IBM were to alter the layout of the graphics memory in, for example, the Enhanced Graphics Adapter, without providing the software with a means to distinguish between this and the previous hardware, much software would be rendered unusable. So IBM decreed that the layout of the video memory would remain fixed for each defined video mode. Since then, each new layout in video memory has had a distinctive video mode (so far the Enhanced Graphics Adapter has taken the number of defined modes to 12, using mode numbers from 0 to 16).

One of the most important changes to the PC architecture was the inclusion of the ability of adapter cards to change the machine by extending or supplanting the system BIOS ROMs. This was the advance that made it necessary to change the BIOS in old PCs when installing an expansion unit or the PC Cluster adapter. The new BIOS scans the ROM address space looking for additional BIOS ROMs on adapter cards during system reset. If a BIOS ROM is found its initialization code is executed, allowing it to install its interrupt handlers, possibly replacing system BIOS ones.

Possibly the two most significant recent changes to the PC architecture were the NETBIOS interface and the inclusion of the 80286. The 80286 came in the AT, proving that the PC architecture could extend beyond the 8088. The NETBIOS interface was introduced with the PC Network, providing a powerful Local Area Network programming interface. Already the NETBIOS interface has proven that it is independent of the PC Network hardware -- it is also available for the Token Ring network.

4.15 SUMMARY

You should now have an understanding and appreciation of the architecture of the PC. The elements that you should understand are:

- the difference between an architecture and an implementation
- the pieces that make up the PC architecture
- the concepts behind the important parts of the PC:
  - Memory
  - Video
  - Keyboard
  - Interrupts
  - I/O Ports
  - DMA
  - Timers and System Clock
  - System Bus
- the considerations involved in designing hardware and software for

It is only necessary that you grasp the larger aspects of how the PC works -- there is no need for you to be able to build one. You should appreciate, for example, what DMA does and what it is used for -- the exact timing of a DMA cycle and the hardware involved is not important to understanding the remainder of the course. From here we will be looking at the layers of software that lie between an application and the naked hardware -- so the exact workings of the hardware become less and less important.

The first layer of software over the hardware is the BIOS. That is the next section of the course. Proceed to it when you feel comfortable with the concepts of the hardware and the 8088 microprocessor.

5.0 8088 ASSEMBLER FUNDAMENTALS

An understanding of major 8088 Assembler programming techniques is essential as a prerequisite for subsequent sections of this course and for future PC technical courses. In this section we will divide key 8088 machine instructions into eight arbitrary groups, and will examine the groups one by one. By the end of this section the student should understand the following:

- The categories of 8088 instructions
- The operands used with instructions
- The usage of 8088 registers and flags
- How data are coded in assembler source code
- The advantages and limitations of programming in assembler vis a vis other programming languages

In addition the student should be able to follow a sample assembly language program.

In order to accomplish these objectives, topics will be covered as follow:

1. The origin of assemblers will be explained to show that Assembler, while some may think it crass and primitive today, was a godsend in its early years and improved programmer effectiveness tremendously.
2. Instruction types will be identified and categorized.
3. The format of assembly language instructions will be explained.
4. Individual instructions will be examined and examples of these instructions will be shown in order to give the student a full understanding of their operation.
5. The student will learn how data definitions are imbedded in source code.
6. We will briefly describe the use of other programming languages available on the PC family in order to show how Assembly language, while versatile and powerful, is not always the best solution to a programming requirement.
7. Finally in the 8088 Assembler lab, students will use the Resident Debug Tool (RDT) of the Professional Debug Facility, and will examine the step-by-step workings of an assembled program, using what they have learnt in the Assembler module.

5.1 THE ORIGIN OF ASSEMBLERS

On the earliest computers, the only way to write programs was to code each instruction in its machine-language format. Machine-language instructions on any computer appear to us as meaningless series of 1's and 0's, or octal or hexadecimal characters. To construct a single instruction could require a great deal of programming time -- each element of the instruction had to be analyzed to determine how particular bits of the instruction were used, and the whole instruction then had to be coded.

Not only was coding in machine language time-consuming and dehumanizing, but errors were unavoidable and debugging was almost impossible. If one bit in one instruction was coded wrong, how would the programmer locate the wrong bit?

Assemblers solved the problems inherent in bit-by-bit programming, by taking over the programmer's task of translating machine instructions into bits and bytes. Instead of the programmer having to calculate what bits needed to be set to code a particular MOV instruction, for example, the programmer typed "MOV AX,BX" and the assembler did the translation into machine code.

Consider how the following program, coded in 8088 assembly language and ready for the assembler, would have to be coded if machine language were the only way to program a PC today:

Assembly language	Machine code (binary)	Machine code (hex)
MOV AX,1	10111000	B8
	00000001	01
MOV BX,2	00000000	00
	10111011	BB
	00000010	02
SUB BX,AX	00000000	00
	00101001	29
	11000011	C3

Assemblers reduced the frequency of errors and the amount of time spent coding dramatically. Debugging time was reduced because the programmer only had to look for misuse of instructions, not mistranslation of the proper instructions into machine code.

Current assembler packages contain other facilities as well to make the programmer's life easier. They often allow programmers to set up their own macros so that they can use a single command to replace a number of machine instructions. Instead of having to code the same set of assembler mnemonics every time a function is needed, the programmer needs only the name of the macro which performs this function. This allows a higher level of programming than was available in the days of writing machine code directly.

Another enhancement of current assembler packages is the ability to use labels to refer to elements in the program's data or code. This means that, when programming, the programmer need not worry about the physical location of a data element or a portion of code in memory; the location can be referenced as a label and the assembly process will take care of establishing the proper address.

Terminology: Now that we know what an assembler does, what is the difference between the terms "Assembly language" and "Assembler"?

Assembly language: A language in which programmers use mnemonics to indicate individual machine instructions.

Assembler: The program which processes assembly language source code and assembles it into machine instructions for use as executable code.

Most people use the term 'Assembler' to mean 'Assembly language'.

5.2 TYPES OF INSTRUCTIONS ON THE 8088

We have divided the instruction types into eight categories. These categories have been chosen arbitrarily, with a view to making the learning process easier. Your objective as a student should be to understand the function of each instruction category, without memorizing the function of more than the most basic of individual instructions. The categories of instructions are:

1. Arithmetic
2. Move
3. Logical
4. Stack
5. Conditional
6. Control
7. 8087 (Math co-processor)
8. Input/Output

5.3 FORMAT OF INSTRUCTIONS

Before we look at individual instruction categories, students should understand the format of 8088 assembly language instructions. Each instruction consists of two components:

Opcode: The operation to be executed (one per instruction) Operands: The variable elements used to affect the meaning of the opcode (zero or more per instruction).

Some instructions in assembly language will include labels for reference to data or to code locations. However, in the machine language version

produced by the assembly process, these labels will be translated into numerical references.

5.4 OPCODES AND OPERANDS

Operands can be **source** operands or **destination** operands. A **source** operand is a variable element which affects the result of an operation, but which itself is not affected. For example, in the statement **"MOVE into BX the contents of AX"**, AX is the source operand, since something is coming from it and going into BX.

Likewise, the **destination** operand is the variable element which is affected by the operation. In **"MOVE into BX the contents of AX"**, BX is the destination operand since the result is stored there.

In some cases, one or both operands may not be required (they can be assumed, or they may not be used at all).

To complicate things further, the 8088 assembler requires the destination operand to be placed **before** the source operand. This means that when we want to "Move the contents of AX into BX", we actually code "MOV BX,AX", or, if you like, "MOVE into BX the contents of AX". If you get used to thinking in terms of moving **into** something **from** something else, this will begin to make sense.

The format of assembler instructions can therefore be summarized as follows:

```
OPCODE [Destination-Operand,] [Source-Operand].
```

5.5 VALID OPERAND TYPES

Operands can be coded as registers, locations in memory, or immediate values (absolute values specified in the source code). Destination operands cannot be immediate (you can't do anything **to** an absolute value) but can be memory or register values; source operands can be any of the three types. For both source and destination operands the valid types (register, memory or immediate) will depend on the particular instruction involved.

Example of Register operands:

```
ADD AX,BX
(add to AX the contents of BX)
```

Example of Memory operands:

```
MOV AX, [DI]
(move into AX the contents of the memory location pointed to by
ES:[DI] -- remember that ES is the default segment register used with
DI)
```

Example of immediate data:

```
SUB AX,12
(subtract from AX the immediate value 12)
```

5.6 THE EIGHT INSTRUCTION TYPES

5.6.1 Arithmetic Instructions

The 8088 is capable of performing some basic arithmetic such as we learned in primary school mathematics. We will review some important arithmetic instructions and give examples of their usage.

Note that in the examples below and in all ensuing Assembly language examples, any text preceded by a semicolon [;] is commentary, used to explain exactly what the instruction does. This is a standard practice in writing 8088 Assembly language programs. Because the assembler discards any comments after a semicolon when it assembles, the comments do not affect the object code in any way, and they make the source code more understandable.

ADD and SUB: ADD takes the contents of the source operand and adds them to the contents of the destination operand.

```
ADD AX, BX    ;adds to what's in AX the contents of BX.
```

The addition is done on the hexadecimal contents of the registers.

SUB takes the contents of the source and subtracts them from the destination.

```
SUB CX,2      ;subtracts from what's in CX the immediate value 2.
```

For both ADD and SUB, the source (the amount to be added or subtracted) can be memory, immediate value, or a register. The destination can be memory or a register.

MUL and DIV: MULtiply and DIVide assume the accumulator and its extension as the destination operand, and this cannot be overridden. The accumulator and its extension consist of:

- DX (high half) and AX (low half) for 16-bit operations

- AH (high half) and AL (low half) for 8-bit operations

Because the destination operand is assumed, it is not specified in the instruction. The source operand is specified as a register or memory location.

```
MUL:
        MUL BX  ;multiply AX by BX
                ;(product in DXAX)
        MUL BL  ;multiply AL by BL
                ;(product in AX)

DIV:
        DIV CX  ;divide DXAX by CX
                ;(quotient in AX)
                ;(remainder in DX)
        DIV BL  ;divide AX by BL
                ;(quotient in AL)
                ;(remainder in AH)
```

The important things to remember with MUL and DIV are:

1. They assume the accumulator and its extension to contain the destination operand.

2. Therefore, no destination operand is specified with the instruction.

Other arithmetic instructions: INC and DEC are used to add or subtract a value of 1 from the destination operand. "INC DX" is the same as "ADD DX,1". The advantages of using INC and DEC are that they require less space in the code than ADD and SUB, and that they execute more quickly than if you use ADD DX,1 or SUB DX,1.

NEG is used to negate the operand. "NEG AX" forms the twos complement of AX, which is equivalent to subtracting AX from 0.

Examples:

```
INC CX              ;increment count register by 1
                    ;same as "ADD CX,1"
DEC word ptr [BP]   ;decrement word value at SS:[BP] by 1
                    ;same as "SUB word ptr [BP],1"
NEG byte ptr [SI]   ;subtract byte value at DS:[SI] from 0
```

Note: "word ptr" and "byte ptr" are used in assembly language to indicate whether the memory element addressed is a word or byte length, respectively.

Shifts and rotates: Shifts and rotates are used to move the bits in a word or byte register around in sequence. We will look at three types of instructions within this category: logical shifts, arithmetic shifts, and rotates.

Logical Shift:

A logical shift pushes the bits in a register out of the register in the direction specified by the opcode. The bits which are moved out are discarded. The vacated positions are filled with zeros. This is best illustrated graphically:

SHR (Shift logical right):

```
                          [register]
              0's ---->    11011010 ---> bit bucket
```

For the above value, "SHR register, 3" would return:

```
                          [register]
                           00011011
```

where "3" is the number of positions to shift. '010' went into the bit bucket.

SHL (shift logical left) follows the same rules as SHR except that the direction of the shift is reversed.

One use for SHR/SHL is that it provides a quick way to multiply or divide a register's contents by a power of 2. For example:

```
                SHL     AL,4
```

If we assume AX contained 3 before the shift, or binary 00000011, the result of the operation will be binary 00110000, which is 30 hex, or 48 decimal, which is 3*(2¬4)=3*(16)=48.

Arithmetic Shift:

An arithmetic shift functions in the same way as a logical shift except that the vacated positions are filled with the sign bit instead of zero in a Shift Arithmetic Right (SAR).

The sign bit is the original high bit of the register. The sign bit is used to indicate whether a number is a positive integer, or a negative integer in twos complement form. If the sign bit is 1, the number is a negative integer in twos complement form.

The purpose of SAR is to ensure that, for operations using arithmetic numbers (numbers which can be positive or negative), the sign bit (and by extension the polarity of the number itself) is not lost.

There is no Shift Arithmetic Left on the 8088 -- you can't shift a sign bit into the low bit of a register, since the sign bit doesn't go there. The PC Macro Assembler assembles SAL (Shift Arithmetic Left) as SHL.

SAR (Shift Arithmetic Right)

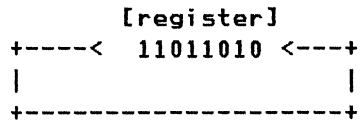
```
                          [register]
            sign bit ---->    11011010  ---> Bit bucket
```

For the above value, "SAR register,3" would give:

```
                        [register]
          sign bit ---->   11111011  ---> Bit bucket
```

Rotate:

A rotate is a logical shift except that the vacated positions, instead of being filled with zeros, are filled with the value of the bit that was most recently shifted out. In other words, a ROL (rotate left) looks like this:



As each bit is pushed out one end of the register, it takes its place at the otner end. For the above value, "ROL register, 3" would give:

```
                        [register]
                         11010110
```

The command ROR (Rotate Right) is similar to ROL except for the reversed direction.

Arithmetic summary: We have discussed the following instructions:

- ADD and SUB
- MUL and DIV
- INC and DEC (like ADD and SUB but source is an immediate value of 1)
- NEG (subtract destination from 0).
- SHR and SHL (shift logical right/left)
- SAR (shift arithmetic right)
- ROR and ROL (rotate right/left).

Note that for more complex mathematical operations, such as exponentiation, the 8087 co-processor is required (80287 with 80286-based PC's).

5.6.2 Move instructions

The purpose of all MOVE instructions is to move data from place to place in the computer. In fact, a MOVE is actually a COPY, since the source of the data is not usually altered by the operation. Data can be moved to and from the following:

- To register from register
- To memory from register
- To register from memory
- To register or memory from immediate data

- To memory from memory (for string moves only).

Bytes, words and strings can all be moved. The format of byte or word MOVE instructions is:

```
MOV     Destination, Source.
```

The format of string MOVE instructions is:

```
MOVSW  or MOVSB.
```

In this case the Source and Destination are assumed, as will be shown shortly.

MOV instruction: MOV allows byte or word moves between registers alone, between registers and memory, and from immediate data to registers and memory.

Between registers:

```
MOV     DS,AX   ;move into DS the value in AX
```

To memory from register:

```
MOV [BP],ES     ;move into word at SS:[BP] the value in ES
```

To register from memory:

```
MOV DL,[SI]     ;move into DL the value in memory at DS:[SI]
```

Note that both source and destination must be the same length. For instance:

```
MOV     AL,BX   ;invalid operation - BX can't fit in AL
MOV     AX,BL   ;invalid operation - BL can't fill AX
```

To register/memory from immediate data:

```
MOV     DX,5    ;move into DX the value 5

MOV     word ptr CS:[BX], 90H
                ;move into memory at CS:[BX]
                ;the word value 90H
```

With immediate data source operands, not all registers can be destination operands. Segment registers are invalid as destinations with immediate sources. For example:

```
MOV  DS,0       ;invalid instruction, will not assemble
```

The proper way to code such an instruction is:

```
MOV AX,0        ;set AX to 0
MOV DS,AX       ;move 0 to DS from AX
```

Finally, you can never move **anything** into CS or IP because this would in effect force the 8088 to jump to an undefined location.

```
MOV CS,...      ;invalid
MOV IP,...      ;invalid
```

There is no alternate way of coding the above, and there is no need for it.

String moves: String moves move a byte or word string from one memory location to another. These commands do not reference the memory locations directly, but through the source and destination indexes (SI and DI) and their associated segment registers (DS and ES respectively). To carry out a word string move, SI and DI are loaded and the command is issued:

```
MOV SI, offset SOURCE   ;this moves into SI the address offset of
                        ;the label SOURCE, which references a
                        ;word string in memory

MOV DI, offset DEST     ;this moves into DI the address offset of
                        ;the label DEST, which references a
                        ;location in memory where the word string
                        ;should go

MOVSW                   ;this copies the word value at DS:[SI] to
                        ;the location at ES:[DI].
```

The MOVSB command can be used to move byte strings instead of word strings. Both MOVSW and MOVSB adjust SI and DI to point to the word or byte following the one on which the string move was just performed.

It you want to move more than one word/byte at a time, and the words/bytes are adjacent in memory, you can use the REP prefix with MOVSW/MOVSB, as follows:

```
MOV CX,10               ;Count of 10 words
MOV SI, offset SOURCE
MOV DI, offset DEST
REP MOVSW               ;moves the word at DS:[SI] to ES:[DI];
                        ;adjusts SI and DI to point to the next
                        ;word; adds 1 to CX; compares CX to zero;
                        ;if CX>0 then execute the command again.
```

(REP loops the statement it is applied to until CX=0)

With the REP MOVSW command and a count in CX of FFFF, you can make a copy of a 128KB block of memory in a single instruction!

Summary of Move commands: Three similar mnemonics are used for moves:

- MOV (for word or byte values)
- MOVSB (for byte strings)
- MOVSW (for word strings)

Word- or byte-value moves can involve registers, memory and immediates.

String moves can use the REP function to copy up to 128K of data from one location to another.

5.6.3 Logical instructions

The purpose of logical instructions is to "mask" on or off selected bits in a byte or word value. Three of the four logical instructions compare the source and the destination, and set the bits in the destination operand according to their own individual criteria. The fourth logical instruction merely inverts every bit in the destination operand. The four logical instructions are:

- AND
- OR
- XOR
- NOT

In the following examples we will illustrate the logical operations through concrete examples. The 8088 logical operations are the same as in other processors, so those with a firm understanding of the instructions need not concentrate on this section.

In our examples we will use the weekly schedules of several people to show how logical operations can help us determine possible meetings between the people.

AND and OR: Two lovers, Annabel and Bill, like to meet at night under the moon. But Annabel plays bridge on Mondays, Thursdays, Saturdays and Holidays; Bill plays shuffleboard on Wednesdays, Thursdays and Sundays. If we want to determine what nights Annabel and Bill will be able to meet beneath their favorite willow tree, we must find those nights where both are not busy.

We have assigned a "1" to every night in which Annabel or Bill is available, and a "0" for any night in which they are not available. We will perform an AND to determine what nights they can meet. We have chosen an eight-day week (Monday through Sunday, plus a holiday) since the 8088 deals more readily with 8 bits than 7.

```
                M T W T F S S H (Holiday)
     Annabel    0 1 1 0 1 0 1 0
        Bill    1 1 0 0 1 1 0 1
                  &     &
      Result:   0 1 0 0 1 0 0 0
```

Annabel and Bill only meet on Tuesdays and Fridays.

The AND instruction sets any bit in the destination operand which is set in both operands before the operation took place. Only if both values

start out with 1 in a particular position will that position have a 1 after the AND. Note that the 8088 assembler format of AND is "AND Destination, Source", for example "AND AL,BL".

If we want to determine on what nights at least one of the two lovers will be able to go to the park and visit their favorite willow tree, we can use the OR instruction.

```
                Annabel OR Bill
                M T W T F S S H (Holiday)
      Annabel   0 1 1 0 1 0 1 0
         Bill   1 1 0 0 1 1 0 1
                / / /   / / / /
        Result: 1 1 1 0 1 1 1 1
```

At least one of them will be able to visit the willow tree every night but Thursday.

The OR instruction sets any bit in the destination operand which is set in either operand before the operation took place.

XOR and NOT: Gustav is an unruly type who likes to throw mud pies at Annabel or Bill in the park. However he is a coward and will only try to bother them when only one of them is there; and there's no point in his going there if neither is around. So Gustav does an XOR of Annabel and Bill to determine what nights only **ONE** of the two (not both, not neither) will be around.

```
                    Annabel XOR Bill

                    M T W T F S S H

          Annabel:  0 1 1 0 1 0 1 0
             Bill:  1 1 0 0 1 1 0 1
                    X   X     X X X

  Only one of 'em:  1 0 1 0 0 1 1 1
```

Gustav can sneak up one one of the lovers on three of eight nights.

The XOR instruction sets any bit in the destination which is set in one or the other of the operands, but not both. XOR is an exclusive OR.

Germaine is not as malicious as Gustav, but has an uncontrollable fear of Annabel. She likes to visit the park but only on nights when she knows Annabel won't be around. By performing a NOT on Annabel, Germaine can determine what nights she can visit the park without fear.

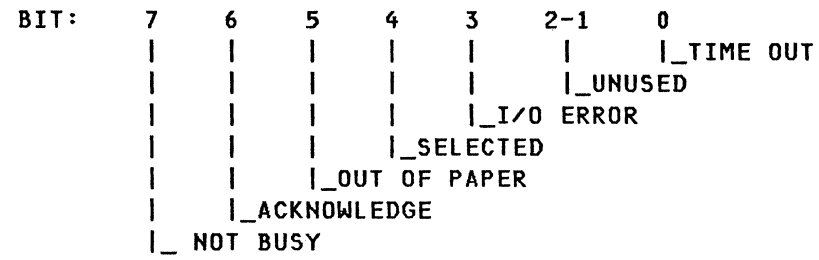
```
                    M T W T F S S H

          Annabel:  1 1 0 0 1 1 0 1

       Not Annabel: 0 0 1 1 0 0 1 0
```

Germaine can go on three of the eight nights, since Annabel goes on the other five.

The NOT instruction has only a destination operand. It clears every set bit, and sets every clear bit, of the destination operand.

Logical instruction summary: As we have seen, logical instructions are used to mask on or off selected bits of a value. This value can be byte- or word-length. Remember that in 8088 Assembly language the format is always OPCODE SOURCE-OPERAND, DESTINATION-OPERAND, so the "Annabel AND/OR/XOR Bill" should really read "AND/OR/XOR Annabel, Bill".

The purpose of bit masking depends on the program's requirements; one common usage is for passing several parameters to a routine within a single byte. For instance, in a Get Status call to the BIOS printer routine, the status is returned in AH with bits set as follows:



A routine invoking the Get Status function call could use logical instructions to isolate a particular bit in the returned status byte, and take appropriate action depending on which bits were set by the function.

5.6.4 Stack operations

The stack is used by the 8088 to keep track of routines which call other routines. When a program calls a subroutine, before the 8088 passes control it pushes the address of the calling routine onto the stack. Then, when the subroutine returns control, the 8088 pops the address of the calling routine back off the stack, and can thereby transfer control back to the original routine which issued the call.

Programmers can also use the stack, as long as they make sure they don't mess up the 8088's use of the stack and thereby cause control to be transferred to undefined locations. Programmers can store the contents of registers on the stack so that their values can be restored after a subroutine or software interrupt handler has modified their contents. Routines can also pass parameters to subroutines by pushing them onto the stack -- the subroutine just pops them off again. PUSH and POP are the two main stack instructions. For example:

```
        MOV BX,12       ;value 12 stored in BX
        PUSH BX         ;save value of BX on stack
        MOV AX,13       ;value 13 stored in AX
        SUB BX,AX       ;subtract AX from BX. Ooops! We lost the original
                        ;value of BX. But...
        POP BX          ;... BX is restored to 12 from the stack.
```

Two variants of the PUSH and POP instructions exist: PUSHF/POPF, which push or pop the flags register onto/from the stack, and PUSHA/POPA, which push or pop all general purpose registers. PUSHA and POPA work only on the 80286, and operate on AX, BX, CX, DX, BP, SP, DI and SI all in a single instruction.

All stack operations are word-based. Valid source operands can be any 16-bit register (you have to push AX if you want to save AL), or any word in memory. Immediates cannot be pushed or popped.

Note that you cannot POP CS or IP. This would cause a branch to an undefined location by setting the instruction address to an undefined value from the stack (however, the RET NEAR instruction in effect does a POP IP, and this is an often-used instruction).

Some students have trouble understanding how the stack is implemented in the 8088 and other Intel microprocessors. The easiest way to understand the stack is to think of a pile of plates. On the 8088, the stack is a pile of plates stored in memory. You can only put plates on at the top of the stack, or remove them from the top; every time you put a plate onto the stack, the pointer to the top of the stack (the STACK pointer, SP) changes to indicate the new top of stack. A PUSH places a plate on the stack; a POP removes a plate. In the 8088, each plate is a word-value.

Parameters can be passed to routines by pushing the parameters onto the stack. The called routine can then pop those values off and use them as required. For example, in the following routine we push the AGE parameter on the stack, and call a routine to determine if the AGE was odd or even. The called routine pops the parameter off the stack, performs the necessary calculation and pushes the result back onto the stack. When the calling routine regains control, it pops that value off and now knows whether the value was odd or even.

```
MAIN:                           ;this is the main part
        PUSH    AGE             ;AGE points to a word in memory
        CALL    ODDEVEN         ;ODDEVEN will determine whether it's even
        POP     AGE             ;On return, we pop the result back to mem.
        .....                   ;and the MAIN routine goes on.
ODDEVEN:                        ;This routine checks the value.
        POP     RETADDR         ;First store return address
        POP     AX              ;Then retrieve the value to change
        AND     AX,1            ;and AND it with 1 to check if it's odd
        PUSH    AX              ;It's pushed back on the stack
        PUSH    RETADDR         ;Restore return address so RET can work
        RET                     ;and control returns to MAIN.
```

Notice that we did not POP AX immediately on entering the routine. Why? Because when we invoked the CALL, the 8088 automatically pushed the address to return to onto the stack. If we had just POPped AX, changed it, and PUSHed it back, we would have PUSHed an invalid return onto the stack, thereby causing the program to return to who knows where.

To get around this we POPped the return address off the stack and into a memory location defined elsewhere as RETADDR. Because our last PUSH was of RETADDR, the original return address is kept intact.

Stack summary: The stack is used by the 8088 for program control purposes. Applications can also use it to save and restore the contents of registers and memory before an operation which might inadvertently alter those contents, and to pass parameters to a subroutine and retrieve those parameters after the subroutine has operated on them.

The main Stack instructions are PUSH (push onto stack) and POP (pop off stack). PUSHF and POPF are the equivalent instructions for the flags register; PUSHA/POPA on the 80286 pushes or pops all general purpose registers.

5.6.5 Flag instructions

The FLAG category of instructions is a very broad, lose category. We have included in it any instruction which modifies one or more of the flags. Some of these instructions belong in another category, but affect the flags on the side. For instance, arithmetic instructions will affect the arithmetic flags (sign flag, carry flag, auxilliary carry flag, overflow flag etc.).

The purpose of changing flags is that a program can do a conditional branch to another location depending on the setting of one or more flags. Since the setting of the flags is in turn dependent on prior operations, a branch can be made on the basis of the nature of the results the prior operation produced.

An easy flag instruction to understand is the Compare instruction, or CMP. In the example:

```
MOV AX,15
MOV BX,30
CMP AX,BX
```

we can see that by comparing AX and BX the result will be that BX is greater than AX. But how is this indicated in the flags?

The CMP instruction works by subtracting the source from the destination without returning the result -- only the flags are affected. If we do a SUB AX,BX with the same values, the sign and carry flags are set, and the result is stored in AX. With the CMP instruction, the registers would not be altered but the sign and carry flags would be set. The CMP instruction

can be used to determine if one value is greater than, equal to or less than another value.

The TEST instruction performs a logical AND of the two operands, affecting the flags without returning the result. The TEST instruction can be used to determine whether particular bits in a value are set or cleared. For example:

```
MOV  AL,  11001110B   ;these numbers are in binary
TEST AL,  00000001B   ;as is indicated by their terminal "B"
```

No result is returned, but the zero flag is set to indicate that the source operand does not have the low bit set.

Direct flag-altering instructions: Some instructions have been included in the 8088 exclusively to modify the flags. In some cases these modifications are done in order to fool another routine about the nature of a result (whether it had an overflow, or was negative, etc). In others they determine aspects of the environment under which the 8088 executes. Below is a description of several of these instructions:

STD (Set direction flag): The Direction Flag is used in string operations (MOVSB/MOVSW and others). If the direction flag is cleared, the string indexes SI and DI will increment after each operation so that string moves, for example, proceed from lowest byte/word in the string to highest byte/word. The STD instruction changes the direction so that string operations occur in highest-to-lowest order instead. The complementary instruction to STD is CLD, which clears the direction flag, restoring the direction to lowest-to-highest order.

STC (Set carry flag): The carry flag is normally altered through arithmetic instructions to indicate the nature of a result. STC allows you to set the carry flag without performing any operation. Its complement is CLC, clear carry flag. These two instructions are sometimes used by service routines which set the carry flag if an error occurred.

STI (Set interrupt flag): The interrupt flag is used to allow or disallow certain external hardware interrupts. By using the STI instruction, you allow a routine to be interrupted by external interrupts coming from the keyboard, for example. By using the CLI instruction (Clear Interrupt Flag), you prevent keyboard input from being processed until another STI instruction is issued. (The Non-Maskable Interrupt, INT 2, cannot be masked out.)

CMC (Complement carry flag): This switches the value in the carry flag. It is equivalent to performing a NOT on the carry flag.

SAHF (Store AH in flags): By placing a value into AH and issuing the instruction SAHF, you can alter the values of several flags at once. The following flags are altered by SAHF: Sign, Zero, Auxiliary Carry, Parity and Carry. The LAHF instruction loads into specific bits of AH the value in these five flags.

Other instructions: Other instructions modify the flags as well, particularly arithmetic instructions. In these cases the flags tell us something about the result of the operation. For instance, in the code excerpt:

```
MOV AX,1
SUB AX,2
```

The result of the operation is negative, so the carry flag is set. And in the multiplication:

```
MOV AX, FFFFH
MOV BX, AX
MUL BX
```

the result of the operation is two words long (the result is stored in DXAX) so the carry and overflow flags are set. In fact, by checking the overflow flag, we can tell whether the result is meaningful. Because FFFF times FFFF has a result longer than 16 bits, the overflow flag is set and we know the result is stored in DXAX.

Logical instructions also affect the flags. In the example:

```
MOV AL, 1010101B
XOR AL, 1010101B
```

the result is zero (there are no bits here which are set in only one operand) so the zero flag is set.

Interrupt instructions save the flags register on the stack, then clear the trace and interrupt flags before transferring control to the interrupt routine. Interrupt **routines** often alter other flags (and registers too) but the INT instruction limits itself to TF and IF.

Flags summary: Many instructions which fall primarily into other categories are also flag-altering instructions. These instructions alter one or more flags to indicate the nature of the result of their execution.

A handful of instructions are used directly to modify the flags without performing any other operation. These instructions are used to change the operating environment of the 8088 (shut off external interrupts, change the direction for string operations etc.) or to simulate the result of an arithmetic or logical operation in order to cause a conditional branch of control.

The flags are used as test values in conditional branches, which we will discuss below. Basically by testing the value of one or more flags we can determine whether to proceed with the next sequential instruction or whether to branch to another location.

5.6.6 Control

In a machine language that did not have control instructions, the computer would be useless. Control instructions are used to make decisions based on the results of an operation; they are used to skip over blocks of code which are not applicable in a particular situation; they are used to allow several routines to access a common function without having to code that function in duplicate. And for the programmer trying to understand a source code listing, the effective use of control instructions allows a more modular approach to programming and therefore clearer, more readable code.

An important aspect of coding controls in 8088 Assembly language is the fact that we don't need to hard-wire addresses of routines into our source code. This means that instead of using instructions like "Execute the routine at offset 06DE from the current segment", we can use instructions like "Execute the routine labeled 'THIS_ROUTINE'". The Assembler takes care of translating the labeled routine into an address so that the control-passing works properly.

One advantage of coding with labels rather than hard-wiring is that in the course of program debugging, addition or deletion of instructions changes the addresses of routines, so hard-wiring controls may cause branches to unexpected locations. Another is that names tend to be more meaningful to programmers than memory addresses. "CALL PRINT_CHARACTER" is a trifle more understandable than "CALL 7600:07F5".

The use of labels is illustrated below, followed by the object code which the assembly process would be produced. The clarity of the labeled code is contrasted with the obfuscation of the object code.

Source code with labels:

```
MAIN_ROUTINE:     CALL SUB_ROUTINE
                   .... ;other code here
SUB_ROUTINE:      ADD AX,BX
                  RET
```

Object code with addresses:

```
06DE:0FF3         CALL 10D3
                  ...
06DE:10D3         ADD AX,BX
06DE:10D5         RET
```

Three types of control instructions: Control instructions can be divided into JUMPS, CALLS and INTERRUPTS. We have already studied these instructions in the 8088 architecture module. We will review each type here.

JUMPS: Jumps transfer control to another location. The location can be SHORT, NEAR or FAR. As we learned in the 8088 module, a SHORT jump is within -128 to +127 bytes of the current instruction; a NEAR jump is within the current segment, and a FAR jump is anywhere in memory.

A JUMP corresponds to a GOTO in a higher-level language. The 8088 does not remember where a JUMP originated, so a programmer would have to figure out her/his own way to jump back if a return to the calling routine was necessary. In 8088 Assembly language the JUMP command is abbreviated to JMP, as in:

```
JMP NEXT_STEP    ;Jump to the location labeled 'NEXT_STEP'
```

CONDITIONAL JUMPS: We discussed flags in the previous section, without delving into why we would want to alter the flags or how we can base decisions on their contents. Conditional Jumps allow us to pass control to different routines depending on the values in the flags (or depending on the value in CX). Conditional Jumps are SHORT only -- you can never jump further than 128 bytes in either direction from the current instruction.

The important thing to understand about conditional jumps is not what flags are tested in individual jumps, but what characteristics of a result are tested. In other words, the jump "JB", or "Jump Below" means "Jump if the destination was smaller than the source". We shouldn't have to think about the flags at all. Just think of the mnemonic. Some common conditional jumps are identified in this example:

```
CMP AX,BX
JA      AX_IS_GREATER          ;JA=Jump if Above
JB      AX_IS_SMALLER          ;JB=Jump if Below
JE      AX_EQUALS_BX           ;JE=Jump if Equal (same as JZ)
JZ      AX_EQUALS_BX           ;JZ=Jump if Zero (Same as JE)
JCXZ    CX_REGISTER_IS_ZERO    ;JCXZ=Jump if CX=0
```

These jumps can all be used to branch control depending upon the result of an arithmetic operation.

The IBM Macro Assembler implements multiple conditional-jump mnemonics for certain Assembly-language instructions, in order to allow the programmer to think about the result of an operation rather than the values in the flags. Thus JE and JZ (Jump If Equal and Jump If Zero) have the same machine-language code (74H and displacement) because both test the Zero Flag and branch if it is set.

As with the JMP instruction, conditional jumps are one-way control transfers -- once the jump has occurred, the 8088 does not know how to return to the location from which the transfer originated.

Because conditional jumps are short only (-128 to +127 bytes from the current instruction), you cannot do a far or near jump. If you want to execute a large chunk of code for a conditional jump you will have to use a format such as the following:

```
        MOV AX,1
        CMP AX,2
        JB  AX_IS_LESS
        ... other code in here (less than 128 bytes though)
AX_IS_LESS:
        JMP AX_IS_LESS_FAR  ;AX_IS_LESS_FAR can be furthern than 128 bytes
                            ;from the JB statement
```

In other words, we do a short conditional jump to a location which then does a near or far jump. This indirectly allows a conditional jump to transfer control to a near or far location, not just a short.

CALLS: The difference between a jump and a call is the difference between moving to Swaziland for good, and taking a vacation there. If you move there for good, you forget your old address and don't know where you came from; you have not paid for your return airfare. If you take a vacation there, you probably bought your return ticket before you left on the trip. If a program jumps to another location, it doesn't know how to get back to where it came from; in a call, it knows it has an easy way back to its home town.

CALLS on the 8088 can be near or far -- control can be transferred within the current segment (NEAR) or anywhere in memory (FAR). The difference between a CALL and a JMP is that in a CALL the 8088 saves the address of the instruction following the CALL onto the stack. When the 8088 encounters a RET instruction which asks for control to be returned to the caller, the 8088 pops the address off the stack and control is returned.

We have already seen several examples of programs containing CALLS. Let's look at another:

```
        MAIN:
                MOV     AH, 75
                CALL    SHIFT_CHAR
                ...     ;other code here
        SHIFT_CHAR:
                SHL     AH, 1
                ADD     AH, BH
                RET                     ;return to caller.
```

In this example, rather than build the shift-character routine into the main portion of the program, we put it on its own, and use the CALL and RET instructions to pass to and from the SHIFT_CHAR routine. There are several reasons we might want to do this.

One is that we can chart out the main block of code in large steps, without showing every detail of operation, by isolating lengthy routines elsewhere in the code and calling them. Thus, a programmer can look at the main block of code and more readily understand what the program is doing.

Another is that we may be calling the SHIFT_CHAR routine from several different places. Granted, it's not a long routine and could easily be coded into every location without impacting code size or speed of execution. But what happens if we decide that the SHIFT_CHAR routine should

Shift Left 2 instead of 1? If we coded the routine as a routine CALLed by other routines, we would only have to recode the single SHIFT_CHAR routine. If we coded the routine in every location where it was needed, we would have to rewrite every piece of code where that routine was used.

The advantages of Calls over Jumps can be summarized as follows:

1. The same code can be called from many locations, which enables a consistent use of that code. For instance, if you write a routine to print a character on the screen, and call that code from anywhere in your program, you know the character will print the same way. If you write the routine right into every block of code which uses it, you may wind up with different versions of your print-character routine, causing inconsistency in your program.

2. Calls reduce redundant coding by allowing access to a single routine from multiple locations.

3. Programs are more modular with CALLS than with jumps. Modularity means that program changes are more easily carried out, and that programs can be more easily understood in their source form by programmers.

4. CALLed routines can return to their caller with the RET instruction. They don't need to know where to return to, the 8088 takes care of that.

SOFTWARE INTERRUPTS: Before we discuss the similarities between software interrupts and calls, we should explain the difference between hardware and software interrupts.

Hardware interrupts are caused by external I/O events. When a key is struck on the keyboard, an interrupt is issued from the 8259 Interrupt controller to the 8088, and the 8088 interrupts whatever it was doing to process the keystroke. Conceptually, a hardware interrupt is having your boss come up to you, interrupt your work, and demand a twenty-page report on your career plans by 4 o'clock. You stop playing Space Invaders and start to write the report.

Software interrupts have less to do with interruption and more to do with requesting services. A software interrupt is issued by a program, usually to access a system routine which is made available to all programs running in the system. For example, the BIOS on the PC provides a routine to read from the keyboard buffer. By issuing interrupt number 16H, an application can obtain the value of the first key in the keyboard queue.

From a conceptual point of view, a software interrupt is like a service provided to the public. You can write your twenty-page report on your career plans, stick it in an envelope, and mail it to your boss. The act of writing the report was the processing of the external interrupt. The postal system is the software interrupt service, and you can invoke this service by placing your report in the mail. The writing of the address is the passing of parameters to the interrupt service (so that it knows what to do with the package).

There is nothing to prevent a hardware interrupt-handler from invoking software interrupts. It is during your processing of your boss's request for a report that you invoke the mail system. Once you've mailed off the package, you can return to your desk and go back to Space Invaders. You don't care about how the mail system gets the report to your boss by four. That's their business.

INTERRUPTS vs. CALLS: Like a CALL, an interrupt (indicated by the Assembly language mnemonic INT) transfers control to a routine which can then return to the routine which invoked it. The 8088 implements this, as with the CALL, by pushing the address to return to onto the stack before control is transferred. However several key differences allow software interrupts to be more globally useful than CALLs.

1. The flags are saved on the stack automatically. The 8088 does this when it first processes the INT instruction. Since interrupt handlers alter the flags so frequently, the pushing and popping of the flags is automatic whenever an interrupt is invoked or returned from, respectively.

2. Encoding of the instruction is only 1 or 2 bytes. Interrupt 20H assembles into CD20H. A CALL FAR 01D0:0335 with a PUSHF would assemble into 9A3503D0019CH, or 6 bytes. Use of interrupt routines cuts down on the size of your code and on the time it takes the 8088 to execute it.

3. No memory address is required in the Interrupt instruction. To invoke the BIOS get-keyboard-input interrupt, you merely issue INT 16H. You don't have to worry about the actual address being called. This not only makes coding easier, but enables you to access the keyboard input routine even if its address has changed. (If a keyboard enhancer has been loaded, for example, it will change the interrupt address.)

Because the routine which invokes an interrupt needs only the interrupt number, it is easy for many different programs to access the same routines using this interrupt interface. For this reason DOS and BIOS are accessed by programs through the interrupt interface almost exclusively. The invoking routine does not need to worry about new releases of DOS or BIOS changing the addresses of functions, because the interrupt vector table will point to the proper location of the functions.

A simple example of using a software interrupt is:

```
        MOV AH, 0               ;AH=0 tells BIOS you want to
                                ;get a keystroke
        INT 16H                 ;INT 16H is the interrupt used for
                                ;application keyboard input
                                ;by BIOS
        CMP AL, 'a'             ;BIOS returns the keystroke in AL.
                                ;Check to see if it's an 'a'.
```

As with the CALL which has a RETurn instruction, INT has an Interrupt return instruction, the mnemonic of which is IRET. If you write a routine which is invoked through an interrupt, you must return to the calling

routine with an IRET so that the flags get restored, control gets returned to the right place, and the stack is properly set.

SUMMARY of CONTROL INSTRUCTIONS:: We have looked at three major types of controls:

```
JuMPs        -- Non-conditional
             -- Conditional
             -- 8088 doesn't know the way back to the caller

CALLs        -- Non-conditional
             -- 8088 can RETurn to the caller
             -- several routines within a program can access it

INTerrupts   -- Non-conditional
             -- 8088 can IRETurn to the invoker
             -- System-wide access to system routines: BIOS, DOS etc.
```

5.6.7 The 8087/80287 Math Co-Processor

We looked at arithmetic instructions as the first instruction type on the 8088. As we mentioned, many of these were instructions we could have understood even in primary school. They do not perform complex mathematical functions such as engineers, scientists, statisticians and other numerical manipulators might want to perform. For this reason Intel created the 8087/80287 math co-processors. The 8087 runs with the 8088; the 80287 with the 80286. These co-processors improve the accuracy and speed of mathematical calculations tremendously. However, simply plugging one of these into your PC will do you no good at all. You have to program explicitly for it.

Math co-processor mnemonics all start with an "F" as the first character. ADD, SUB, MUL, DIV and so on still are carried out on the 8088; to get the speed and accuracy of the co-processor, you would code FADD, FSUB, FMUL and FDIV. You would also have to use different locations for storing your operands; however the particular functioning of 8087 instructions is beyond the scope of this course.

Some language compilers and interpreters, and some end-user applications, require the 8087 in order to execute. Macro Assembler version 2.0 supports the 8087 instructions. Macro Assembler version 1.0 did not.

5.6.8 Input/output instructions

The 8088 can address 64K (65,536) ports of input/output. Very few of these are used on the PC. The 8088 allows access to these ports through IN and OUT instructions. IN takes a byte in from a port, and OUT sends a byte out to a port. In fact, at the lowest level, the IN and OUT instructions

are one of only two ways to send information to the world outside the main processor and the math co-processor. The other way, which is more restricted, is through the use of DMA.

In an IN or OUT instruction, AL is always the register used for the byte to be input or output. The port can be an immediate number from 0 to FFH, or a word value stored in DX. For example:

```
IN      AL, 61H         ;input a byte from port 61H
MOV     AL, 74H         ;Now AL=74H
MOV     DX,03DAH        ;set up DX to address port 3DAH
OUT     DX,AL           ;output byte in AL to port addressed
                        ;by DX (3DAH)
```

It is very rare for a programmer to use the IN and OUT instructions. Because the BIOS and DOS interrupt routines provide all the I/O support needed by most programmers, application programs use the interrupt interface. The BIOS routines themselves, however, use IN and OUT instructions copiously because this is how they communicate with I/O devices.

5.7 DATA ENCODING IN ASSEMBLER

We have already discussed the use of labels to make referencing of code or data elements easier. Now we will examine how data are stored within the source code in assembly language. There are four elements to a data definition in assembly language. Two of these are optional, and are indicated in parentheses:

```
[DATA_LABEL]  DEFINE-TYPE   [LENGTH]  VALUE
```

A real example would be:

```
MYAGE         DB                      24
```

The elements of this are:

```
LABEL       DEFINE AS BYTE          VALUE
```

Note that the blank spaces in this example are not significant to the assembler.

The LABEL can be used in the source code, so that direct address referencing is not used in the source code:

```
MOV AL, MYAGE
```

moves the value stored at the memory location labelled MYAGE into AL. The Assembly process will figure out what the actual memory address is. A label is optional in a data definition, however if a reference is to be

made the starting address of a data element, it must be labeled. For example, in the definition

```
MYADDRESS       DB      '29 Ranleigh Ave.'
                DB      'Toronto, ONT Canada'
```

there is no label for the second line, because it is a continuation of the first.

The Define-type tells the assembler what length to assign to the data value. Some possible types are:

```
BYTE                        (DB)
WORD          (2 bytes)     (DW)
DOUBLEWORD    (4 bytes)     (DD)
QUADWORD      (8 bytes)     (DQ)
```

The value itself can be specified in binary, decimal, hexadecimal, octal, ASCII characters, or a combination of the above. Octal, however, is rarely used because it is designed for 6-bit machines and the 8088 works only with values of 8 or 16 bits.

```
- Binary:       10100101B
- Decimal:      1938 or 1938D
- Hex:          20FAH
- ASCII:        'Gelato'
- Combination:  'Gelato',20FAH, 101B
```

The assembler normally assumes that values which are not explicitly declared as binary, hexadecimal or octal are decimal. However the default value type can be changed to any of these. As for ASCII values, they are always enclosed in single quotes.

The Length portion of a data definition allows you to define a longer variable quickly, provided it contains the same byte/word value in each position. The "n DUP" statement is used to indicate a repeat prefix of the value which follows (where n is a value between 1 and 64K). For example

```
LOTS    DB      8 DUP ('a')
```

is the same as

```
LOTS    DB      'aaaaaaaa'
```

In this example, little coding space is saved by using the DUP function. However, consider a program in which a disk I/O buffer is established to allow file access. At initialization of the program, the buffer will not contain anything, but we would like to reserve space for it. The buffer could be coded as follows:

```
FILE_BUFFER  DB 4096 DUP (?)
```

This sets aside 4096 bytes of space for the buffer. The (?) value says that it doesn't matter what value is placed into that buffer to start with, but the space will be reserved.

5.7.1 Beware of stray data definitions

In some types of assembler programs (.COM programs), the data is stored right in the code segment, whereas in others (.EXE programs) the data is stored in a separate data segment. From the processor's point of view, there is no difference between data and code in the code segment of a .COM file. If you define a data variable in the middle of a subroutine, the 8088 may try to execute it! Consider the following example:

```
        MOV     DX, 3
        SILLY   DB 0F4H
        INT     37H
```

This assembles as:

```
        MOV     DX, 3
        HLT                     <--- halts the processor!
        INT     37H
```

You can still access the data variable F4H, but if the 8088 runs into it, it will interpret the data as an instruction, and your CPU will halt. A way to code around this is to JuMP over any data definitions defined in source code, thus:

```
        MOV     DX, 3
        JMP     NEXT_STEP
        SILLY   DB 0F4H
NEXT_STEP:
        INT     37H
```

This is not a serious problem. It just pays to remember that to the 8088, instructions are just bits and bytes, and data is just bits and bytes, and it will treat one as the other if you tell it to.

To summarize the data definition format, we code data definitions in source code as:

```
        [DATA_LABEL]  DEFINE-TYPE   [LENGTH]  VALUE
```

The data label is optional, and is used to refer to the value inside source code. The Define-Type can be byte, word, doubleword, or quadword. The value itself can be written in Binary, Octal (obsolete), Decimal, Hexadecimal or in ASCII characters. And the optional LENGTH repeat prefix allows a value to be repeated any number of times.

5.8 OTHER PC PROGRAMMING LANGUAGES

Before we examine the value of other PC programming languages, we should understand why people still do use assembly language. Many application-oriented programmers regard assembly language as a dinosaurian tool which takes forever to code, forever to debug, and has to be rewritten for every new hardware release. All of this is true to a degree, but there are some very good reasons for using Assembly language for a variety of system-oriented programs.

The first consideration is that Assembly language runs very very fast. It runs faster than BASIC, "C", APL, Pascal, or anybody else -- faster than all of them put together! All these other programs eventually break down into machine language anyway, so if you want speed, why not start there? A compiled BASIC program uses a routine library to access its most common routines; every time one of these routines is called, a complex series of instructions is executed to check the validity of the call, the parameters passed, and so on. In Assembly language you only generate the instructions required to do the work.

Of course Assembler isn't any faster than BASIC if the programmer doesn't know what s/he is doing. But a good assembly-language programmer can write a program that runs faster on a PCjr than a BASIC program of similar function will run on a 3090.

The second reason for using Assembly language is that it is very compact. The sample program used in this course, which reads keystrokes and converts lower-case characters to uppercase before displaying them, has been coded in "C", BASIC, Assembler using BIOS and Assembler using DOS. The Assembly-language versions are a small fraction of the size of the BASIC version. For system-oriented programming, where many complex functions must be crammed into as few bytes as possible, Assembly language offers compactness of code, provided the programmer uses proper algorithms.

The third reason for using Assembly language is that it offers much more control over the 8088 and the PC system. A BASIC programmer may not care whether a value is being stored in a register or whether it's in memory, but if an Assembly-language programmer knows a value is going to be referenced fifty thousand times per second during a particular routine, s/he will be able to give that value a high-access-time location in storage -- a general register.

The worst reason for using Assembly language is that it's easy to debug. Actually it is harder to debug Assembly language than any other language except APL. However, if the programmers have done their job and put tons of lively comments in the source code, debugging becomes quite easy.

5.8.1 Other PC languages

Assembler may be fast and compact and offer lots of control, but it's not everyone's kettle of fish. Many programmers prefer more high-level programming languages because these languages don't expect the programmer to know the machine inside out. Another feature of high-level languages is that some languages come with interpreters -- tools which allow you to run programs directly, without compiling them. Interpretive programs are far slower than compiled ones, but during development of a program it's much easier to test a simple change slowly in an interpreter than to do it by recompiling, which may take ages.

Other PC languages are easier to understand than Assembly Language (except for APL). These languages are designed to address the needs of the programmer, not the architecture of the processor. And because these languages are less directly associated with the processor, the code becomes more transportable. An 8088 Assembly language program is useless on a VAX system, but a "C" program which runs on the PC can be recompiled with no major changes to run properly on the VAX system.

It may take dozens or hundreds of machine instructions to implement one high level instruction. High-level languages speed up the application development cycle by requiring minimal coding and producing powerful programs. The programs may be slower and bulkier than machine-language programs, but they won't have taken 17 years to write.

The five PC languages we will review are Pascal, BASIC, Fortran, APL, and "C".

5.8.2 Pascal

Pascal is a high-level, structured language, used heavily in Computer Science courses to teach students true structured programming. Programmers who use a GOTO statement in Pascal are subjected to heavy chastisement by their programming professors. Pascal is known for powerful data-structuring facilities and for modular programming units, which allow extremely large programs to be written in small units. The Pascal compiler on the PC provides good detection of syntax errors during compilation -- unlike a compiler like most "C" compilers, which will give you hundreds of syntax errors if you forget the first bracket in a program.

5.8.3 BASIC

BASIC is another high-level language, although it is not structured in its interpreter format, and only barely in its compiled format. It is fairly easy to learn, easy to get addicted to, and difficult to use in

an effective way. BASIC on the PC is available in compiled and interpretive versions. The interpretive versions are very slow, but are good if you want to write short, low-priority routines. For example, you can write a ten-line program in BASIC to print out diskette labels, use the program once, and not feel bad about throwing out the program. Compilers for BASIC on the PC allow the code to run rather faster, but anyone serious enough to want to compile their programs would be better off programming in a more powerful structured language such as "C" or Pascal.

5.8.4 FORTRAN

Fortran stands for Formula Translator. The Fortran language is very old -- dating back to the mid-fifties, when not much else was available. It is still used extensively, particularly in engineering, scientific and mathematical environments. On the PC, IBM Professional Fortran and IBM Fortran version 2.0 both use the 8087/80287, providing fast execution of math functions. Fortran is a medium-level language without the bells and whistles of Pascal but with more punch. Fortran is not well-liked by programmers of other languages, but it has its following, and its uses.

5.8.5 APL

APL stands for A Programming Language. This isn't nearly as cryptic as the language itself. APL does not use alphabetic mnemonics for instructions, it uses funny-looking characters which pack a lot of function into a single space on the screen. APL must have been designed for the days when you bought video displays by the square centimeter. Because it is so highly symbolic, it is also highly undecipherable, and the best way to debug an APL program is to write a new one that works properly. Yet the instructions are very powerful, and though APL on the PC is only available in an interpretive version, it does work relatively fast and is great for math. APL on the PC requires the 8087 co-processor.

5.8.6 "C"

"C" stands for "C". It's the successor to "B", which probably sprang out of "A". "C" is a low- to medium-level language, designed in Bell Labs and about the only language used on UNIX-based systems. "C" source code is designed to be extremely transportable; it is structured without being obsessed about it. "C" produces compact and efficient object code, and is heavily used by system hackers. Many operating systems and device drivers are being written in "C" these days.

5.9 CONCLUSION

During this module we saw how Assembly language reduced the headaches of programmers who previously had to code bit by bit in machine code; we learned the eight categories of 8088 instructions -- Arithmetic, Move, Logical, Stack, Flag, Control, 8087 and Input/Output -- and we examined specific examples of each instruction category. We learned the format of instructions -- "OPCODE [Destination], [Source]", and looked at several examples of instructions which follow this format. We also learned the format of data encoded in Assembly language; finally, we examined the reasons some programmers today still prefer to use Assembly language, and we skimmed over the features and drawbacks of five other programming languages used on the PC.

Students should now have accomplished the objectives outlined at the beginning of this module. We will repeat them here; think them over as you read, and make sure you review any objective you haven't reached, by rereading the section involved or asking an experienced workstation specialist for help.

By the end of this section the student should understand the following:

- The categories of 8088 instructions
- The operands used with instructions
- The usage of 8088 registers and flags
- How data are coded in assembler source code
- The advantages and limitations of programming in assembler vis a vis other programming languages

In addition the student should be able to follow the assembly code of the sample program used in the Assembly Language lab.

If you go on to later topics without understanding the essentials of what we covered in this lecture, you may become lost as references are made to materials covered here. Finally, the sample assembly language program is contained in Appendix C of this guide. Browse through it and make sure you understand its major components, and recognize some of the instructions.

When you have achieved a strong comfort level with 8088 Assembler Fundamentals, you should proceed to the module on the PC BIOS.

6.0 BIOS

The BIOS (Basic Input/Output System) of the IBM PC handles low-level I/O to devices such as the keyboard, diskettes, printers and the like. It allows programmers to perform I/O to and from these devices at a much higher level: the program requests a service from the BIOS, and the BIOS takes care of all the physical procedures which must take place in order to fulfill the request.

The BIOS, in software terms, is what gives the IBM PC its individuality over PC compatibles, just as the PC architecture gives the hardware its individuality over other personal computer architectures. By individuality we do not necessarily mean incompatibility, but in some cases features existing in the IBM BIOS will not exist in compatible BIOS's. Through an understanding of the BIOS you will be able to judge the differences between IBM Personal Computers and compatibles, and hopefully to some extent between IBM PC's and incompatible but functionally equivalent computers as well.

6.1 OBJECTIVES

By the end of this module, you should understand the following:

- The purpose and structure of BIOS
- The assembly language interface to BIOS routines
- What the six key BIOS routines are
- The importance of using the BIOS interface rather than direct hardware addressing
- How direct hardware addressing works, and why it's dangerous.

In order to accomplish these objectives, we will examine the following aspects of BIOS:

1. BIOS as compared to the first I/O control systems
2. The purpose and structure of BIOS
3. BIOS's power-on diagnostics
4. The assembly language interface
5. Several individual BIOS routines
6. How and why programs bypass BIOS

7. How DOS and other PC operating systems use BIOS

8. BIOS support for add-on modules to the BIOS.

In the classroom version of this course, a lab will be conducted to show differences between using the BIOS interface and using direct hardware control.

6.2 FIRST I/O CONTROL SYSTEMS

Long before the Personal Computer, in the days when a machine with a PCjr's computing power would have filled a department store and weighed sixty gigatons, every program written on a computer had complete control of the machine. The program ran alone in the computer, unassisted by any operating system, I/O support system, or anything else. This meant that each program had to have its own hardware-level I/O support routines. Imagine a routine as simple as our keyboard-to-display routine which converts keystrokes to their uppercase equivalent and prints them on the display. This routine, rather than involving a dozen or two dozen instructions from the programmer's point of view, would involve every IN, OUT and other instruction our program invokes through interrupts. On the first computers, in other words, a tremendous amount of effort was expended on coding and recoding the same old thing -- low-level I/O.

Then someone had a bright idea, and decided to write the routines once and for all, and allow everyone to use them. They provided a mechanism for accessing the routines from a program, and called the routines an I/O Control System. Don't worry about who wrote it, when, or exactly what it did. Suffice it to say that the first I/O control systems allowed many different programs to use the same I/O routines, and that for a programmer, I/O coding became a much more elevated and simplified process. Furthermore, if a hardware change occurred -- a new device was installed, for example, or a shop decided to switch from a mag card reader to a 3380 disk subsystem -- as long as the I/O control system was revised to allow the old interface to access the new device, there were no problems of hardware dependence.

6.3 THE PURPOSE OF THE BIOS

The BIOS on the PC is similar to the first I/O Control Systems in that it takes care of the lowest level of I/O support for the most common PC devices, thereby allowing programs to make use of the available I/O without expending a great deal of programming effort. In addition, the BIOS was designed to allow programs which followed the BIOS interface to be compatible with future BIOS releases (which would support new hardware).

Let's summarize the philosophy behind the BIOS:

- Allows an easier interface to supported hardware than direct hardware coding
- Allows a consistent interface -- if a program follows the BIOS routines, and IBM changes the hardware, the new BIOS shipped with the new hardware will still respect the existing interface and old programs will still work
- Eliminates hardware-dependent code
- Provides system services for applications and operating systems
- Performs hardware diagnostics at power-on.

6.4 STRUCTURE OF BIOS

BIOS is stored in a read-only memory chip on the PC or AT system board. From an addressing point of view it is identical to any other area of memory, except that it cannot be modified. It is located at segment F000H, from offset E000 or thereabouts to offset FFFF. BIOS is the first thing which the PC executes after power is switched on.

BIOS routines are accessed via software interrupts; hardware interrupts cause BIOS routines to be invoked as well, so that when an external interrupt occurs, BIOS can process it.

6.5 POWER-ON DIAGNOSTICS

When the power switch is turned on, the CS register of the 8088 is initialized to all 1's in binary, or FFFF hex. The IP register is initialized to all 0's in binary, or 0000 hex. This causes the instruction at FFFF:0000 (which could also be viewed as F000:FFF0, or the very end of the BIOS) to be executed. The instruction which is stored there does a far jump to the RESET routines which perform power-on diagnostics for the PC.

The power-on diagnostics are designed to perform the following:

- Verify the proper functioning of system hardware
- Initialize hardware to its start-up state for use by I/O routines
- Display any diagnosed errors on the screen
- Test memory to see if there are any optional BIOS modules installed, such as the PC Network NETBIOS or the Fixed Disk BIOS
- Load the bootstrap routine to bring up the operating system.

We will examine each of these in more detail.

6.6 VERIFY FUNCTIONING OF SYSTEM HARDWARE

The power-on diagnostics check every component of the PC system board, and several attached I/O devices, to ensure their proper functioning. Memory is checked by writing values to it and reading the same values back to make sure the data is properly recorded and returned. The 8088 itself is run through a series of tests (ironically, it is testing itself) in which individual registers and instructions are tried out, and erroneous results cause a branch to an error routine. If the 8088 works properly, the associated microprocessors are diagnosed as well -- the DMA controller, the Interrupt controller, the math co-processor, etc.

Once these aspects of the system are deemed operational, the power-on diagnostics check certain I/O devices including the display, keyboard and diskette drive. The blinking cursor and the brief instant at power-on in which the cursor shoots forward several columns before returning to the home position are an indication that the display tests have executed. The first spinning of the diskette drive on power-up, before the bootstrap-loading spin, is a check of the drive motor. Once an I/O device is deemed to be operational, any initialization routines required are carried out, so that BIOS routines can perform I/O to the device properly.

Add-on BIOS modules, which are loaded at a later stage in the power-on diagnostics, perform their own diagnostics as well. We will review these add-on modules later on.

Diagnostic error messages are displayed on the console. They are not designed to be highly enjoyable reading, and some of the 4-digit error codes are undocumented. However they are useful in isolating problems when the problems prevent the loading of the Diagnostics diskette. By using the hardware maintenance manual you can determine whether an error is memory-related, expansion-unit-related, keyboard-related and so on, and to an extent you can isolate the problem. But many users have confessed to being less than thrilled with the user-hostility inherent in messages like '3015 ERROR. (RESUME = "F1" KEY)'

In cases where the error message cannot be printed -- tests did not even proceed far enough to be able to display a message -- the sound generator is used to indicate the nature of the problem. The length and number of beeps determines whether the failure is processor-related, power-supply, or one of several other hardware failures.

6.7 OPTIONAL BIOS MODULES

Once the BIOS has done its own diagnostics and has initialized the hardware, and before it loads the bootstrap from diskette or disk, it scans

memory to see if any other BIOS modules are stored there. Anyone can write their own BIOS module (it's not easy, but it can be done). BIOS scans memory from segments C800 to F400, in 2K blocks. (In fact, the segments it scans will depend upon the PC type -- PC, PC XT, PC AT etc. Consult the appropriate technical reference manual for exact addresses.) The BIOS looks at the first two bytes of each block; if the first two bytes of the block are 55AAH, and if the block passes a number of other tests, control is passed to the instruction following the 55AAH identifier. Normally, the optional BIOS will do its own tests and initializing of its hardware; it may also alter the environment the System-Board BIOS set up before control was passed. For example, the Fixed-Disk BIOS resets interrupt vectors set by the System-Board BIOS so that if the diskette drive is empty on IPL, the system can be loaded from the hard disk rather than branching to Cassette BASIC. Once the optional BIOS has finished its own initializing, it returns control to the System-Board BIOS, which continues to scan up to F400 in search of other add-on BIOS modules, and finally loads the bootstrap routine.

6.8 LOADING THE BOOTSTRAP

Once the hardware has been initialized the BIOS attempts to load the operating system. An INT 19H is issued which invokes the BIOS bootstrap routine. The bootstrap does not know in advance what operating system it will find; in fact it never knows or cares. BIOS first reads track 0, sector 1 from the first diskette drive (drive A: in DOS). If it is able to load something from there, control is transferred to whatever it loads. We will review the DOS loading procedure in the DOS module. However, if the BIOS finds no diskette present in the diskette drive (or if no diskette drive is installed), it will jump to the BASIC stored in ROM, by issuing INT 18H, the Cassette BASIC interrupt.

The Fixed-Disk BIOS replaces this bootstrap module with its own, by intercepting INT 19H, the bootstrap loader. Like the original bootstrap routine, it starts by looking for a boot record on diskette. But it adds a step between failed diskette bootstrap and the jump to BASIC: it attempts to load from the first fixed disk. If it finds a valid bootstrap record, control is passed there; if not, the Fixed-Disk BIOS jumps to BASIC. This allows the operating system, whatever it is, to be loaded from the fixed disk.

6.9 POWER-ON DIAGNOSTICS: SUMMARY

The Power-On Diagnostics perform testing and initialization of system hardware, validation of optional BIOS modules, and bootstrap loading. Once the operating system is loaded, applications may use the BIOS interface to access the I/O devices (provided the operating system has not altered the BIOS interrupt vectors to point elsewhere).

6.10 ASSEMBLY LANGUAGE INTERFACE

To access BIOS routines an interrupt mechanism is used. Each I/O device has one or more assigned interrupts. The usual procedure to follow is to load the AH register with a value indicating the type of function to be executed, to load other registers with whatever other parameters are needed, and to issue the appropriate interrupt. It is not always necessary to load AH with a function value -- some I/O interrupt routines have only one option (such as PrintScreen -- INT 5) and don't look at the register contents. Otherwise AH must be loaded.

An example of an Assembly-Language interface to the BIOS is to print a character to the first parallel printer port (LPT1).

```
MOV     AH,0    ;function call 0 to Print Character
MOV     AL,'A'  ;character to print is 'A'
MOV     DX,0    ;DX contains the printer port:
                ;    0=LPT1
                ;    1=LPT2
                ;    2=LPT3
INT     17H     ;BIOS printer interrupt
```

By placing a different value in AH the function desired can be changed. However you must ensure that the function you select is valid, or the BIOS will return control to you without doing anything.

When a valid function call is issued to BIOS via the interrupt mechanism, BIOS will return information to you: either the status of an operation, if it was an output operation, or the data input from an input operation. For Printer I/O, the Get Status function call returns a byte value into AH, the various bits of which indicate the success or failure of a number of aspects of the printing process. And the value returned from INT 16H, which requests a keystroke from BIOS's keyboard buffer, is stored in the accumulator: AL contains the ASCII value of the keystroke, or a zero if no ASCII value corresponds to the key; AH contains the pseudo-scan code of the key. We will review scan codes later in this module.

Applications written in languages other than assembler use the BIOS indirectly -- from a programming point of view, the programmer need only specify the function to accomplish, such as printing a character, and the compiler takes care of constructing a machine language routine which sets up the registers properly and issues the interrupt. Alternately the compiler may construct a routine to invoke a DOS function, which will in turn invoke the BIOS.

6.11 I/O SUPPORTED BY BIOS

We will review six I/O areas which are supported by BIOS. The areas are:

- Keyboard
- Video
- Diskette
- Disk
- Printer
- Serial

Keyboard, Video, Diskette, Printer and Serial I/O are the five major I/O types supported by the System BIOS. Disk BIOS is an optional module, but because hard disks are so common on PCs, and because of the similarities between Diskette and Disk I/O, we will treat it as though it were supported directly by the System BIOS.

6.12 KEYBOARD

The keyboard is an input-only device. The BIOS keyboard routines allow buffered keyboard input, which means several characters can be stored in a buffered queue before being processed by the operating system or application. This is what allows us to continue typing up to fifteen keystrokes even when an application is busy doing something else.

Two I/O routines, and two interrupt interfaces, co-operate within the BIOS to provide keystrokes to applications.

The high-level routine is responsible for providing input to applications, on request. This routine is accessed by applications through INT 16H. When an application issues INT 16H, BIOS reads a value from the keyboard buffer and returns that value to the application. If no value is present (if the keyboard buffer is empty), BIOS waits until a keystroke is received. The high-level routine updates the pointer to the beginning of the keyboard buffer, to point to the character after the most recently read character. The value is a word-length value, containing a keyboard scan code in the low byte and an ASCII value in the high byte. We will discuss the contents of this word in a moment.

The low-level routine is responsible for receiving input from the keyboard and placing it in the keyboard buffer. This routine is a hardware-invoked interrupt (INT 9H) and occurs independently of any application requests for keyboard input. Each time a key on the keyboard is pressed or released (and also each time the keyboard sends a typematic repeat of a key which is being held down), an INT 9 is issued by the 8259 interrupt controller. (Remember -- the interrupt level of the keyboard is 1. The 8259 adds 8 to this to obtain software INT 9.) At this point the 8088 interrupts whatever work it was doing, and processes the I/O event.

When INT 9 is invoked, its BIOS routine processes the input in one of four ways:

- normal key, possibly shifted
- shift key

- toggle key

- special purpose key, requiring special actions

A normal key make code is converted into a character using one of a number of translation tables built into the BIOS, depending on the current shift status. The shift status includes the status of the two normal shift keys, the Caps Lock key, the Num Lock key, the Ctrl shift key, and the Alt key. The converted character is placed into the keystroke buffer, ready for the program/keyboard interface to collect it.

A normal key break code is generally discarded.

A shift key make code is a signal to set the appropriate shift flag, indicating that shift status. A shift key break code is a signal to clear the appropriate flag. No other processing is done, and no entry is made in the keystroke buffer.

A toggle key differs from a shift key in one respect. It can be set by the make code, but it is not cleared by the break code. Instead, the next make code clears the flag. Thus the state of the flag associated with a toggle key changes with each make code for the key encountered. The Ins key, Caps Lock, Num Lock, and Scroll Lock, are all toggle keys. Again, no entry is made in the keystroke buffer.

There are special processing keys. The first is the Del key; when the Del key make code is received the interrupts service routine checks the Alt and Ctrl shift flags. If both are set then a processor reset is initiated by invoking INT 19H. This is the Ctrl-Alt-Del reset. The second special processing key is the Num Lock key; when a Num Lock make code is received the Ctrl shift flag is checked. If it is set then the keyboard routine spins, waiting for the next make scan code. This is Ctrl-Num Lock pause -- it pauses the machine by making it totally busy waiting for the next key. The third special processing key is the Scroll Lock key; when its make code is received the Ctrl shift flag is checked; if it is set then INT 1BH is invoked. The fourth special processing key is the PrtSc key; when a make code for this key is received, and the shift flag is set, INT 5 is invoked. The PC AT introduces a fifth special processing key - the SysReq key. When a make or break code for this key is received interrupt 15 is invoked with special parameters. The AT also provides a special interface for code wanting to intercept every scan code; details of this can be found in the AT Technical Reference manual. None of the special processing keys place entries in the keystroke buffer when they cause their special functions.

6.13 SCAN CODES

The microprocessor in the keyboard does not tell BIOS the ASCII value of a key, and this allows keys such as the function keys, cursor keys and shift keys to be defined. It also allows keyboard enhancement programs

to redefine the values of keystrokes. The keyboard itself merely sends a value to the keyboard input routine each time a keyboard action occurs.

Each key on the keyboard is assigned a scan code, from 1 to 83. The make code consists of the scan code of the key involved. The break code consists of the sum of the scan code + 80H.

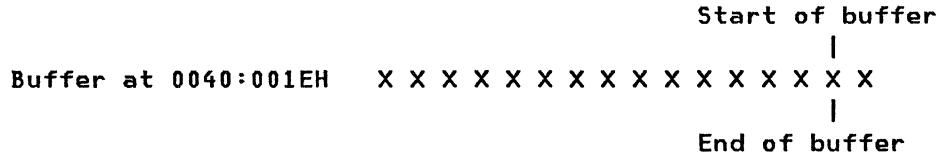
If a make scan code is not a shift code, and if it does not have an associated ASCII value, the look-up into the keyboard table in BIOS will return a value of -1 (FFH). If such a value is returned, the BIOS will place a 0 value as the low byte of the keystroke, and the high byte will consist of a pseudo-scan code. This pseudo-scan code is a number indicating the key combination struck. Keys like F1-F10 will have their normal scan code assigned as their pseudo-scan code; however if one of these is pressed while a shift key is active, the pseudo-scan code will be different. This assignment of pseudo-scan codes allows software programs receiving keyboard input from INT 16H to determine when such keystroke combinations as SHIFT-F3, CTRL-F7, ALT-4, ALT-F and so on are struck. The pseudo-scan codes for keystroke combinations are listed in the technical reference in the System BIOS section.

6.14 KEYBOARD BUFFER

INT 9 can only place a character into the keyboard buffer if there are free spaces in the buffer. If there are no free spaces, INT 9 causes a beep and does not process the keystroke. If there is available space, INT 9 places the keystroke into the buffer at the first available two-byte slot. It then updates the pointer to the next position in the queue.

If the buffer is full, no more characters can be placed in it until some are read out by INT 16H.

To understand the keyboard buffer it is important to understand the concept of a circular queue. In a circular queue, each time a value is placed at the end of the queue the end-of-buffer pointer is incremented; each time a value is removed from the beginning of the queue the start-of-buffer pointer is incremented. The queue is circular in that when one of the pointers exceeds the highest address in the queue, it simply wraps back around to the lowest value. The BIOS keyboard buffer is located at segment 0040 from offset 001EH to 03DH. Let's examine a scenario where several keys are pressed, then read by INT 16H, in order to see how the buffer pointers wrap around. We will ignore the low byte of each keystroke, which contains the scan code, since the alphanumeric character is enough to understand the process.



When we begin the example, our keyboard buffer is empty -- all keys that have been input from the keyboard have been read by an application. Let's assume that the pointers both point to the second-last position in the buffer. Now a key is struck -- an "H":

```
                                          Start of buffer
                                                 |
  Buffer at 0040:001EH   X X X X X X X X X X X X X H X
                                                   |
                                            End of buffer
```

The End-of-buffer pointer is incremented whenever a character is received. Now what happens when INT 16H reads the character in?

```
                                            Start of buffer
                                                   |
  Buffer at 0040:001EH   X X X X X X X X X X X X X H X
                                                   |
                                            End of buffer
```

H is read in, and the start-of-buffer pointer is incremented. Since the start and end are the same, there are no keys left to be read in the buffer. The fact that the "H" is still in the buffer is inconsequential -- it has already been read, and will be overwritten after 16 more keystrokes.

Now comes the circular aspect of the queue. Both pointers point to the end of the queue, so what happens when "e" is pressed?

```
                                            Start of buffer
                                                   |
  Buffer at 0040:001EH   X X X X X X X X X X X X X H e
                         |
                End of buffer
```

The end-of-buffer does not increment if it is already at the highest address of the buffer; instead it goes back to the lowest address. The start-of-buffer pointer will do the same once it has read in the "e". An easy way to think of this wrapping is to think of an odometer turning over from 9999 to 0000. If there are only four digits on the odometer, it will wrap around to 0 when it reaches 9999. When a pointer in the keyboard buffer reaches the last buffer position, it wraps back to the first position.

6.15 KEYBOARD SUMMARY

Keyboard input is received by BIOS INT 9 and placed in the keyboard buffer. Software programs can read in this input via INT 16H. INT 9 performs conversion from scan codes to ASCII values, or to pseudo-scan codes where no ASCII value exists for the key-combination received. By assigning hardware input to one interrupt and software input to a second,

buffering allows programs to continue executing as soon as the lowest level of keyboard input is processed, without the program's having to actually read the value in from the buffer. Finally, the buffer is a circular queue, whose head pointer is managed by INT 16H, and whose tail pointer is managed by INT 9.

6.16 VIDEO

BIOS provides support for video output to the monochrome and color graphics adapters, which in turn output to their respective displays. BIOS has some support for input from the display -- but this is not really input, it is only reading in values which were previously output to the display.

BIOS supports both text and graphics modes on the color graphics adapter, while it only allows text mode on the monochrome adapter. Text mode allows the display of all base and extended ASCII characters, including the graphics characters such as lines, corners, crosses etc. Graphics mode allows all-points-addressable display of images. The video interface is accessed via INT 10H.

In addition BIOS supports the redefinition of the appearance of extended ASCII characters (characters with ASCII value 128 to 255) when displayed in text mode with the color graphics adapter. By creating a table which contains these character redefinitions, and pointing to that table with interrupt vector 1FH, graphics characters can be defined which are not available in a normal IBM-extended ASCII character set.

6.17 BIOS VIDEO OUTPUT ROUTINES

The following output routines are available for text-mode output:

- Print a character and attribute
- Set cursor location or type (ie. what cursor looks like)
- Set mode -- text, low, medium or high resolution graphics
- Set active display page (text only)
- Scroll display page (text only)

In addition, these simple input routines are supported:

- Read a character and attribute from the display
- Read position of cursor

- Read current mode of display

- Read number of current video page

Video pages are available with the color graphics adapter only. The color graphics adapter contains 16K of RAM -- this amount of memory is required in the graphics modes to address all pixels on the screen. In text, only 4,000 bytes are required to display a single screen of text (25 lines by 80 columns=2,000 character positions; the screen requires two bytes per position, one for the character to display, one for the foreground/background attributes of the character). Since 4,000 bytes fit into 16K four times, the BIOS supports the use of four video pages. Applications can write data to any of the four pages, without necessarily displaying the page they write to. Applications can also switch between any of the four pages. This allows programs to write to an undisplayed page, then switch to display that page and have all changes to the page appear immediately.

Video pages are not supported on the Monochrome adapter, which only has 4K of video storage, and room for only one page.

6.18 PRINTING TO DISPLAY

An example of a BIOS video routine follows. Remember that the attribute character is used to indicate what foreground and background are used to display the character.

```
PRINT_TO_DISPLAY:

        MOV     AH, 9           ;function: write character and attribute
                                ;at current position
        MOV     AL, 'A'         ;character to write is 'A'
        MOV     BX, 07          ;attribute is black bg, white fg
        MOV     CX, 10          ;CX contains count of times to display it --
                                ;here we display it ten times
        INT     10H             ;call BIOS Video I/O interface
```

The character 'A' will be printed in 10 consecutive positions on the display, in white on a black background.

6.19 GRAPHICS CHARACTER REDEFINITION

With the color graphics adapter, programs can redefine the appearance of the upper 128 ASCII characters. Eight bytes are required to redefine each character. Since each character in text mode on the CGA consists of 8 by 8 pixels, the redefinition uses 8 bits (or one byte) times 8 bytes to accomplish the redefinition of a character.

Once the characters have been redefined, the beginning of the table must be pointed to by placing the table's address in the vector table at location for INT 1FH. This means that INT 1FH does not point to executable code, like other interrupt vectors usually do -- it points to data. You would never issue INT 1FH, because it would kill the system by trying to execute data.

Example: redefine character 128 to be the thorn character

```
CHAR_128 DB      00001011B          ;       1 11
         DB      00111101B          ;     1111 1
         DB      11010001B          ;   11 1   1
         DB      00010010B          ;      1  1
         DB      00011100B          ;      111
         DB      00010000B          ;      1
         DB      00010000B          ;      1
         DB      00100000B          ;    11
```

By defining every bit to be either 1 or 0, you define every pixel in the 8-by-8-pixel matrix to be either foreground or background. In the above example the 1's create an image somewhat like the thorn character used in Old Anglo Saxon writing. By placing the segment and offset of CHAR_128 in the vector table at the location for INT 1FH, the thorn character will display on the CGA in text mode every time ASCII code 128 is printed on the display.

6.20 DIRECT SCREEN ADDRESSING

We have already discussed video memory mapping in the PC architecture module. As we said, each of the display adapters has RAM built into it, which has reserved segment addresses which any application can address. The monochrome screen is mapped to segment B000H; the Color screen is mapped to segment B800H. By writing to these memory locations directly, characters and attributes can be displayed on the screen without using the BIOS or DOS interfaces. Whereas a BIOS call's registers must be set to display the character 'A' at the topmost, leftmost position by specifying the character, attribute, position, page number and number of times to print the character, in direct screen addressing a program just writes to location B800:0000. We will examine direct screen addressing in more detail in the section on bypassing BIOS routines, and during the BIOS lab.

6.21 GRAPHICS

BIOS supports APA graphics for the Color Graphics Adapter, whereby through BIOS function calls individual pixels can be addressed in a variety of resolutions and palettes. The details of BIOS graphics support can be obtained from the technical reference manual. Direct Screen Addressing is used by most programs to accomplish graphics-mode video output on the

CGA. By writing to locations B800:0000 to B800:3FFF, programs can access the full 16K used for displaying graphics. The CGA graphics modes have already been discussed in the PC Architecture module.

6.22 VIDEO BIOS SUMMARY

The BIOS routines for video I/O support a wide range of output functions, as well as some elementary input functions which allow programs to find out what characters, attributes, modes or pages are currently being displayed.

Direct Screen Addressing is an alternative way of displaying information on the screen. However, although at first glance it may seem like an easier interface than BIOS, it is in fact much more difficult for a program to manage than using the BIOS, because a direct-screen-addressing routine must itself take care of always knowing what position to print the next character at, and must manage scrolling information up and down on the screen.

In addition to the ease of use the BIOS provides over direct screen addressing, adherence to the BIOS video interface will ensure that programs are compatible with future hardware and with software packages which do not support direct screen addressing. For example, TopView does not support windowing or background operation of any task which does direct screen addressing.

6.23 DISKETTE

BIOS handles only the lowest-level I/O to and from diskette. BIOS diskette I/O is block-based rather than byte-based; this means that a block of memory is used to transfer data between the physical medium and the requesting system or application program. Data transfer with a high-speed device like a diskette through two-byte registers is simply impractical.

BIOS deals with diskette I/O in tracks and sectors, the physical units of data recording on diskette. It does not know anything about files or directories. File- and directory-based I/O routines are generally provided by the operating system. File-based I/O is far safer and easier to use than direct usage of BIOS diskette I/O, and any application programs which use BIOS diskette I/O directly risk damaging of valuable data, and risk incompatibility with other programs.

BIOS supports three types of diskette activity:

- Reading of physical units (sectors) from diskette into a block of memory
- Writing of a block of memory to physical units (sectors)

- Formatting of diskettes.

Students of this course have a requirement to know anything more about diskette I/O than the above, because application programs should never attempt diskette I/O through any interface other than the operating system interface. The BIOS diskette interface is provided for the use of operating systems, and for standalone programs which run without an operating system.

6.24 FIXED DISK

In terms of the support provided for fixed disk by BIOS vis a vis the support for diskette, there are very few conceptual differences. The Interrupt and parameter passing are different, but the same principals apply: BIOS handles only the lowest level of fixed disk I/O, and does so on a memory-block basis, transferring data from disk to memory or vise versa, not through registers. As with Diskette BIOS, Disk BIOS is designed for use by operating systems only and should not be used by programs running under an operating system.

6.25 PRINTER

BIOS supports output of characters to a line printer on a byte basis. Individual characters are output by placing them in a register, setting AH to the value for the Print function call and issuing INT 17H. In addition to character output, status can be obtained from a printer to determine whether a previous print operation was successful or whether an error has occurred. BIOS supports three parallel ports for a total of three parallel printers.

The PrintScreen function is accessed via INT 5. This routine prints the contents of any text screen onto the first line printer. INT 5 is invoked by the keyboard interrupt INT 9 whenever the Shift-PrtSc key combination is detected. INT 5 takes each character from the screen and prints it using INT 17H.

The INT 17H interface is shown in this example:

```
        MOV     AH,0        ;Print character function
        MOV     AL,'K'      ;Character to print is 'K'
        MOV     DX,2        ;Printer is LPT3 (0=LPT1, 1=LPT2)
        INT     17H         ;BIOS Printer interrupt
                            ;status returned in AH
        PUSH    AX          ;save value on stack
        TEST    AH,1        ;if low bit=1 then there was an error
        JE  PRINT_ERROR     ;so jump to our error handler
        POP     AX          ;restore stack and continue processing
```

Notice that the printer interface automatically returns a status byte in AH after any output operation. In addition, a BIOS call exists to explicitly request the printer status of any of the line printers without an output being performed. The status covers Time-Out, Out-Of-Paper, Printer-Busy, and other I/O errors.

6.26 SERIAL PRINTERS

Serial printers are printers which use the serial interface rather than the parallel. Serial printers are not supported as printers by the BIOS printer routine. However they are supported through DOS, which will redirect output from a parallel to a serial device by intercepting INT 17H, checking to see which printer a byte is being printed to, and sending that byte to a serial port if the printer number is assigned to a serial device.

6.27 SERIAL I/O

BIOS supports the use of two serial ports, known in DOS as COM1 (or AUX) and COM2. BIOS supports transmission speeds of up to 9600 baud. The three functions supported by BIOS are initialization of a serial port, receiving of a single byte and sending of a single byte. The serial I/O routine for output is invoked by BIOS INT 14H. Hardware interrupt levels 3 and 4 are used by serial ports (3 for COM1, 4 for COM2) for input, thereby giving INT 0BH and INT 0CH as input interrupts which are invoked when a byte is received by the adapter. BIOS itself does not provide any buffering of characters; this is the responsibility of the application or operating system using the serial interface.

6.28 HOW AND WHY PROGRAMS BYPASS BIOS

If the BIOS routines provide a high-level, easy-to-use interface to primary I/O devices on the PC, and ensure compatibility to that interface for future hardware products, then why on earth would anyone want to use any other interface?

There are several reasons why certain programs choose to bypass the BIOS routines and control hardware directly through their own I/O routines:

- These programs require speed of execution, and therefore speed of I/O, not the ease-of-use features of an I/O interface
- They have abnormal I/O requirements

- They use certain copy-protection mechanisms which look at diskette storage or hardware directly to make sure the system or diskette being used to run the program is licensed to do so.

The principal reason for bypassing BIOS is to increase the speed and sophistication of I/O.

6.29 DIRECT SCREEN ADDRESSING

As we discussed in the VIDEO section, programs can display text (and graphics on the CGA) by using the BIOS interface, or by writing directly to video memory at segment B000 or B800. The reason some applications write directly to the screen (which is another way of saying Direct Screen Addressing) is that the I/O occurs much faster. Since, in the end, BIOS is writing to the screen itself, why not have the application write to the screen directly, and do it a lot faster?

There are several good reasons why this kind of program behavior should be avoided. In terms of development effort, writing a routine which does direct screen addressing can be far more difficult than writing one which uses the documented BIOS interfaces. For another, such routines risk incompatibility with their operating environment. For example, TopView, which intercepts all BIOS video calls in order to take its own windowing routines into account, can do nothing to stop direct-screen-addressing applications from doing their video I/O directly to screen memory, and therefore these applications cannot be windowed. A third reason for avoiding direct screen addressing is that it is incompatible between monitors. If a routine uses INT 10H to display characters, the text will display on the active monitor whether that monitor is color or monochrome; if the routine does direct screen addressing to the monochrome adapter's memory location, it will still print to the monochrome display even if the color display is active. And if future display adapters are announced with new memory locations for their video memory, all the direct-screen-addressing code will have to be rewritten.

6.30 BYPASSING KEYBOARD ROUTINES

By writing custom keyboard routines which bypass the BIOS routines, a number of interesting effects can be produced. For one, routines can be programmed to use different keystroke combinations. In TopView, for instance, when no mouse is present, the ALT key can be pressed, then released, without pressing any other key, to bring up the TopView menu. TopView traps the hit-and-release of the ALT key by processing hardware INT 9H's from the keyboard before the BIOS routine does. A routine which intercepts hardware INT 9H judges each keystroke by its scan code, not its ASCII value. This means that such a routine can detect when a key like the ALT or CTRL key was hit and released.

By writing custom keyboard routines you can also reassign the values of individual keys. DOS national language support allows different ASCII values to result from the same scan-codes by loading the appropriate keyboard driver (electronically, US and French keyboards, for instance, function identically). Another benefit of writing custom keyboard routines is that the length of the keyboard buffer can be increased. DisplayWrite 3 increases the length of the keyboard buffer to 80 characters, so that while DisplayWrite 3 is processing the character you typed to paginate a document, you can type up to 80 characters of text at the same time (although you won't see the text until the document has finished paginating).

Another possible way of altering the keyboard routines is to intercept INT 16H and cause keyboard input to applications to come from a file instead of from the keyboard buffer. This amounts to sending artificial keyboard input to applications. DOS performs a version of keyboard simulation through its redirection of standard input -- you can use a file to send input to a program instead of the keyboard. However DOS does not intercept INT 16H in redirecting I/O, it intercepts the DOS function call used for obtaining keyboard input.

Because the keyboard buffer is stored in memory, some applications read this buffer directly, and update the head pointer to it themselves. This is probably done to increase speed of execution of those programs, but causes compatibility problems when INT 16H is revectored. For instance, if you write a routine which redirects INT 16H input requests to its own source of alternate input, and you feed those requests with data from a file, programs which read the keyboard buffer directly won't hear a whisper from your INT 16H routine.

6.31 BYPASSING OTHER BIOS ROUTINES

It is rare for an application to bypass BIOS' printer routines. There is little benefit in writing one's own low-level printer routines. The only instances where the printer routines are bypassed are for spooling purposes or device-sharing purposes. If you install a spooling program on your PC, your application can do all its printing through INT 17H, but instead of the print going out to the printer, it is stored in memory or in a file which is subsequently printed by the spooler, while you and your application do other work. We will see how this kind of multitasking works in the DOS Extensions module. Another reason for rewriting printer I/O has to do with the PC Network. When a remote user on a network is using a PC on another printer, the PC Network program intercepts INT 17H calls and sends the data across the network to the remote print server instead of printing out at the local station.

Serial I/O routines are rarely modified either, except in network environments where a serial device is shared across a network.

With disk and diskette BIOS very few applications even descend to the BIOS level: most use the operating system file services. However some appli-

cations use copy-protect mechanisms which perform direct hardware I/O to disk or diskette. For instance, certain spreadsheet packages alter the speed of rotation of a diskette drive, and try to read in a track at the reduced speed. If the track does not read properly, then the diskette is presumed not to be an original and the program will refuse to load.

Using the BIOS routines directly for disk or diskette output, or going below these and using the physical I/O interface, is a good way to ruin the contents of a disk or diskette. For this reason and for simplicity's sake, few people ever access secondary storage through any interface other than the operating system.

6.32 HOW DOS USES BIOS

In addition to the BIOS interface, programs running under DOS can use a DOS interface to accomplish I/O with the keyboard, display, printers, serial devices, disk and diskette. In the case of disk/diskette I/O, the DOS and BIOS interfaces are worlds apart; the other I/O routines supported by DOS are only small-scale enhancements to the BIOS routines.

The DOS interface adds the following I/O capabilities to BIOS:

String I/O for console: Whereas the BIOS routines support the keyboard input and video output of single characters only, certain DOS keyboard and video function calls support input or output of entire strings of characters. This makes it much easier for a program printing a long message to do so -- instead of printing every byte of the message, it just tells DOS to print the whole message.

File I/O for secondary storage: This allows programs to deal with secondary storage in logical, rather than physical, units: files and directories rather than sectors and tracks. DOS file support will be thoroughly discussed in the DOS module.

Redirection of standard input and standard output: The standard input device is the keyboard; the standard output device is the active display. DOS allows redirection of input or output, so that instead of receiving keystrokes from the keyboard, an application can receive them from a file, and instead of displaying output on the display, an application will print it to a file, or to a printer. The application itself has no idea that the redirection is taking place; this is a DOS function.

To illustrate the similarity in programming terms between the DOS and BIOS interfaces, observe how a character is printed to the printer with each interface:

```
BIOS: INT 17H
(AH)=0:function to print character to printer
(AL)=character to print
(DX)=printer to use (0,1,2 for LPT1,2,3)

DOS: INT 21H (same INT 21H for most DOS functions)
(AH)=5:function to print character to standard printer
(DL)=character
```

Notice that DOS assumes that output is to go to the standard printer. Also, eventually DOS will use the BIOS INT 17H to do the actual printing.

The difference in video interface between DOS and BIOS is also worth looking at:

```
BIOS: INT 10H
(AH)=9:function to write character/attribute to screen
(AL)=character
(BL)=attribute

DOS: INT 21H
(AH)=9:function to print string to standard output
(Other video output functions are also available...)
(DS:[DX])=Pointer to a character string (string must end in '$')
```

Notice that DOS can print a whole string of characters, whereas the BIOS must proceed one character at a time (the EGA BIOS extension and the PC AT BIOS both contain string-output to the screen, however the PC AT version is not supported and is reputed to contain bugs). Also, in the BIOS call the character will display on the screen no matter what redirection of output is taking place, whereas with the DOS function, if output is being redirected to a file, the string will print to the file instead of to the screen.

The four important aspects of the relationship between DOS I/O routines and BIOS I/O routines are:

- DOS and BIOS routines are sometimes similar in function
- DOS enhances some functions by reducing the amount of programmer coding necessary to produce output or receive input, and by allowing redirection of standard input and output
- DOS allows programs to be less device-dependent than BIOS
- Eventually in the process of the I/O handling, the DOS routine will probably invoke its BIOS counterpart.

6.33 BIOS AND OTHER OPERATING SYSTEMS

Although PC DOS makes heavy use of the BIOS to implement low-level I/O, other operating systems do not use BIOS as heavily. In fact, by revectoring hardware interrupts to point to locations in RAM, an operating system can control all I/O directly without ever using the BIOS except at power-on time, before the operating system has been loaded.

XENIX does not use the BIOS at all; it replaces all BIOS I/O routines. This is because XENIX uses the 80286 in protect mode (as opposed to real mode where the 80286 operates as an 8088). The code in the BIOS is not compatible with the 80286's protect mode so the BIOS won't run under XENIX. In addition, the interrupt mechanism is substantially different on the 80286 in Protect mode. Reasons for this will become apparent in the 80286 architecture lecture. The resultant drawback is that with new hardware releases, new device drivers must be written; and with the current support of XENIX, it is up to the end-user to write these device drivers.

The programmers of PCIX figured they could do a better job than the BIOS as well, so they wrote their own I/O control subsystem. We do not know the rationale they used in deciding to write their own system. This is proprietary information.

6.34 OPTIONAL BIOS MODULES

Our last BIOS topic deals with components of BIOS which are not included in the original BIOS contained on the PC system board. These optional BIOS modules allow the devices they support to have the same kind of BIOS interface as keyboard, display and so on, without requiring the ROM chip containing the BIOS to be changed. Detailed information on how these BIOS modules are activated is contained in the section on power-on diagnostics. The intent of this section is not to explain the theoretical functioning of optional BIOS modules but to look at three actual examples of BIOS modules: Fixed Disk BIOS, PC Network NETBIOS, and Enhanced Graphics Adapter BIOS.

6.35 FIXED DISK BIOS

The original PC did not support a fixed disk. With the announcement of the PC XT and the PC Expansion Unit, it became necessary to provide some kind of fixed disk BIOS support. The Fixed Disk BIOS is stored on the fixed disk adapter, in a ROM chip addressed by the system unit as starting at segment C800H. This BIOS module not only provides I/O handling for fixed disk but intercepts all diskette interrupts to make sure they shouldn't be directed to the fixed disk instead; it also replaces the

original bootstrap BIOS to make sure that if the boot record isn't found in Drive A: the fixed disk is scanned for it.

6.36 NETBIOS

The NETBIOS for the PC Network Adapter is contained on the adapter card. It provides network diagnostics during power-on, and network I/O support through an interrupt interface similar to the interface of other BIOS routines. In fact this BIOS module is considerably more sophisticated than most, and an entire course could be devoted to its study. The NETBIOS is stored at segment CC00H.

Programs written to the NETBIOS interface (which includes all programs written to run under the PC Network Program, since this uses the NETBIOS interface) will also run on the IBM PC Token Ring adapter, when the Token-Ring NETBIOS program is loaded on the PC or PC's involved. This program is loaded from diskette or disk like any other DOS program, and acts as a device driver to allow all NETBIOS function calls to be redirected through the Token-Ring network.

The Token-Ring NETBIOS program illustrates above all other BIOS modules the importance of coding to the BIOS interface. On the Token-Ring, without even using a ROM chip for the network BIOS, programs written for the NETBIOS interface can run without modification, and with increased performance, once the Token-Ring NETBIOS program is loaded. Programs written to interface directly with the PC Network hardware will not work at all on the Token-Ring.

6.37 ENHANCED GRAPHICS ADAPTER

The BIOS on the Enhanced Graphics Adapter replaces the Video BIOS by intercepting INT 10H. The EGA BIOS provides a more sophisticated way of displaying information in both text and graphics modes. As mentioned above, it provides string output to the display, whereas the system-board BIOS only provides single-character or same-character-repeat output. It also provides support for the new Enhanced Display, and for the new display modes available on the monochrome and colour graphics displays. Finally, when used with the Enhanced Display, it provides Color graphics or Monochrome display emulation.

6.38 ADD-ON BIOS MODULE SUMMARY

IBM intentionally saved memory locations from C8000H up to F4000H for add-on BIOS modules, to allow for support of additional or modified hardware without requiring reinstallation of the system-board BIOS. The

Fixed Disk BIOS, NETBIOS and EGA BIOS are good examples of add-on modules which provide support for new I/O devices. And there are still many 2K blocks of memory available for additional BIOS modules.

6.39 FINALLY...

This concludes our discussion of the IBM PC BIOS. During this module we examined the purpose and structure of BIOS, by comparing it to the first I/O control systems; we examined individual components of BIOS, including the power-on diagnostics and the assembly-language interface.

We also looked at individual BIOS routines, and noted their characteristics and their assembly language interface. Some of the routines we examined were:

- Keyboard
- Video
- Disk/Diskette
- Printer
- Serial

We looked at the DOS interface to BIOS -- how DOS enhances BIOS functions with its own, and how DOS eventually gets around to using the BIOS routine in its own way. We touched on redirection of standard input and output as it relates to DOS vs. BIOS methods of performing I/O.

Finally, we examined several add-on BIOS modules briefly: The fixed disk BIOS, the PC Network NETBIOS, and the Enhanced Graphics Adapter BIOS.

At this point, you should have reached the objectives of this module. You should understand:

- The purpose and structure of BIOS
- The assembly language interface to BIOS routines
- What the six key BIOS routines are
- The importance of using the BIOS interface rather than direct hardware addressing
- How direct hardware addressing works, and why it's dangerous.

In addition you should be able to perform the BIOS lab outlined in the appendix.

A lab is conducted for students in the class version of this course. The script for this lab is contained in Appendix D, along with the source code used. If you are studying this text on your own, we suggest you try the lab exercise to strengthen your knowledge of PC BIOS fundamentals.

When you have achieved a strong comfort level with the BIOS topic, you should proceed to the module on PC DOS.

7.0 DOS CONCEPTS AND FACILITIES

So far we have moved quite a long way up the chain of hardware and software components which combine to form the total PC operating environment. We have more or less wrapped up the hardware, and in the BIOS lecture introduced the first software component, although the BIOS is not entirely "soft" -- it is rather difficult to change since it is written on an unwritable ROM chip on the system board.

PC operating systems generally form the final layer between the hardware and the user's applications. DOS is the primary operating system used on IBM PC's (XENIX and PCIX are more elaborate operating systems but are not as widely used). Although DOS can be subdivided into layers of its own, in a larger sense it is a single, complex layer. In this module we will show how DOS fits into PC internals, and will also examine the layers, or components, of DOS itself.

7.1 OBJECTIVES

By the time you have completed this module you should be able to:

- explain what IBM PC DOS is and what it does
- detail the IPL process in a PC
- explain the process of reading a command
- describe the disk format used by DOS
- describe the file management facilities available under DOS
- understand memory allocation under DOS
- understand and use the DOS program interface
- differentiate the types of program DOS supports
- describe the support that DOS provides for programs
- explain DOS error handling
- describe device management in DOS

7.2 WHAT IS DOS?

The IBM PC Disk Operating System (almost always referred to as PC DOS, but in this document simply as DOS) is the most widely used operating system for the IBM PC. It provides a powerful set of functions in an easy-to-use format, and consumes very little memory.

An IBM PC needs an operating system to make it simple to load and execute programs. Without an operating system every piece of software would have to include its own operating environment. This would make it quite difficult to share information between software packages, because they would probably use different disk formats. It would make it difficult to switch between software packages, for they would probably use different operating environments, with different sets of I/O functions. Of course, if they used the same disk formats, and the same, standard, operating environment, it would be easy. That is what DOS is: an operating environment for programs, providing a set of standards to allow programs to work together. DOS also happens to make writing programs much easier, for the programmer can concentrate on the application, rather than the support environment.

7.3 WHAT DOES DOS DO?

DOS is an operating system. What does it do? In a large mainframe the operating system manages all the resources, sharing them amongst many users, and making sure that one user does not corrupt another user's resources, either intentionally or accidentally. DOS is the same, except that DOS users are programs, and because the PC is a relatively simple device it is only possible to protect programs from one another's accidental corruption. The PC provides no security against malicious behavior.

DOS manages resources. These include all the I/O peripherals, with an emphasis on disk, but also including the screen, keyboard, printer, and sundry other devices. Memory is a resource, and so, in a way, is the user! DOS manages command interpreting for the user, to allow the user to enter a command, have the appropriate program executed, and have control returned to the point where another command may be entered.

DOS provides services. The simplest of these have to do with the management of resources like files. More complex services support the installation of new device drivers, the ability for one program to cause the loading and execution of another, and the ability of the user to tailor the DOS system to conform to the user's requirements.

DOS provides error handling. All manner of errors occur in a PC, not all of them due to the user. DOS handles each of them in an appropriate generalized way -- a diskette drive without a diskette in it brings the familiar "Not ready error reading drive A - Abort, Retry, Ignore?". This

error handling may seem crude at first, but consider how **you** might handle a general error condition like this.

To get a feel for what DOS does, and how it does it, let us consider first how DOS loads itself, and then some of the more complex considerations associated with a DOS program.

7.4 IPL OF A PC

When the ROM BIOS completes its power-on diagnostics it checks the disk drives for a boot record. The boot record is located in sector 1 of track zero, head zero of a diskette, or the first sector of a hard disk partition. If a valid boot record is found on a diskette in drive A:, or on hard disk C:, it is read into memory, and given control. The boot record may simply issue a message describing the disk as a nonsystem disk, or it may attempt to bootstrap the system, depending on the format of the disk volume.

If it attempts to bootstrap the system it looks at the first two entries in the root directory, and verifies that they are IBMBIO.COM and IBMDOS.COM, in that order. If this is the case, the boot record reads them both into memory and passes control to IBMBIO. Once that is done the boot record is no longer required and the memory it occupied becomes free.

IBMBIO.COM is the low-level DOS interface to the ROM BIOS routines. It initializes the DOS interrupts, then reads CONFIG.SYS (if present) to determine what optional device drivers to load, and how large to make certain system tables (like the file handle table (FILES=), the disk buffer table (BUFFERS=), and the block device table (LASTDRIVE=)). Once all the device drivers have been read in and initialized IBMBIO gives control to IBMDOS.COM.

IBMDOS.COM initializes its internal data areas and the function call handling mechanism -- all DOS function calls are handled in IBMDOS. DOS service routines are located in IBMDOS too. When IBMDOS completes its initialization it loads COMMAND.COM (unless another command interpreter has been specified in CONFIG.SYS using SHELL=), and passes control to COMMAND.COM initialization.

COMMAND.COM has three separate pieces: a resident portion, an initialization portion, and the transient portion. The resident portion of COMMAND.COM is loaded immediately above IBMDOS.COM (with the exception of the boot record all of the previous pieces of code are loaded into the lowest available memory locations). The resident portion of COMMAND.COM handles program loading, termination and errors. The Critical Error and Control-Break vectors point into here. The initialization portion of COMMAND.COM establishes the interrupt vectors for the resident portion, and processes AUTOEXEC.BAT (if present). When the end of AUTOEXEC.BAT is reached the initialization portion passes control to the resident portion which loads the transient portion.

The transient portion of COMMAND.COM is the command interpreter. It reads in a user command, interprets it, and instructs the resident portion to load a program, if that is required. All internal commands (COPY, DIR, etc) are contained in the transient portion. The interpreting of BAT files is done by the transient portion. The transient portion of COMMAND.COM is loaded as high as possible in free memory. If it is overlaid by a program during the loading or execution of that program the resident portion will load a new copy to interpret the next command. If the transient portion is not overlaid the resident portion simply passes control to it when the program terminates. The resident portion determines whether the transient portion was overlaid by running a checksum on the memory that the transient portion occupied.

7.5 ENTERING A COMMAND

The reading of user commands is handled by the transient portion of COMMAND.COM. The transient portion begins by displaying the command prompt. The command prompt can be customized by the user using the DOS command "PROMPT". The transient portion then uses a DOS function call, number 0AH, to request a line of text from the keyboard. When this line is returned it is parsed for a command. The first string (sequence of nonseparator characters not containing a separator, where a separator is defined as any of blank, tab, null, comma, or slash) is assumed to be either an internal command, or the name of an external command. The internal commands are listed in the DOS Reference manual. An external command may refer to a COM, an EXE, or a BAT file. The transient portion looks first in its own data area to see if the command is an internal command. If not it looks in the current directory of the current disk volume to find the external command. It will look for a file with any of the three extensions mentioned above. If no such file is not found there, the sequence of directories listed in the PATH is searched until a matching command file is found. If the end of the PATH is reached without a match the transient portion reports the familiar "Bad command or filename", and reissues the command prompt.

If the command is a COM or EXE file then the remainder of the command line is parsed in an attempt to build two unopened FCBs for the program. The transient portion then passes instructions to the resident portion of COMMAND to load the specified program. Unless the program has been specifically created to be loaded high (a parameter in the linking process) the resident portion looks for the lowest block of free memory large enough to hold the program, and creates a Program Segment Prefix at the beginning of it. This Program Segment Prefix contains information which the program can use to access parameters which were passed to it, and other information which we will soon discuss. COMMAND then loads the program, performing any relocation required (if the file is an EXE file). The resident portion passes control to the program at its specified entry point.

If the command turns out to be a BAT file then the transient portion opens the BAT file and parses the first line as a command line. When that

terminates, it opens the file again and parses the next line, and so on, until the end of the file is reached. Each line of the BAT file is read from disk in its turn, which means that the file must remain accessible to the transient portion whenever the next line is to be read. This can be a problem with BAT files on diskette. Because the transient portion has only one place in which it can store the name of the currently executing BAT file, and a counter to determine what the next line of the file is, nested BAT files are not supported. If the command interpreter discovers a command in a BAT file to be a BAT file, it simply execute that BAT file, without going back to the other BAT file. This technique is sometimes known as "chaining". When the end of a BAT file is reached the command interpreter returns to the keyboard for the next command.

7.6 DISK FORMAT

One of the most important resources that DOS handles is disk storage. Diskettes and hard disks are handled similarly, the only difference being that hard disks can be partitioned. A partition on a hard disk can be considered a disk volume, as can a diskette. Every volume has the same format. DOS numbers every sector on the disk, starting with track 0, head 0, sector 1, and proceeding through all the sectors on this track, then all the sectors on the same track for the next head, and so on. The point of this scheme is to minimize first head movement, and then head switching. (Note that heads and tracks are numbered from zero, while sectors are numbered starting at one -- there are only historical reasons to justify this strangeness.)

Disk space is allocated on a first-available basis, choosing free space as close to the outer edge of the disk as possible. DOS attempts to keep data, directories, and programs as close to the outside of the disk as possible. This has two advantages, one being that it minimizes seeking, resulting in better performance, the other being that data recording is more reliable on the outer tracks, because they are longer.

DOS allocates space in groups of sectors called clusters. A given disk volume has a fixed cluster size, typically one sector on diskette and four or eight sectors on a fixed disk. A cluster is a group of sectors which are sequential in sector number.

To track the allocation of space DOS uses a File Allocation Table, known as a FAT. Multiple copies of the FAT are kept, to provide redundancy should one become damaged. Normally two copies of the FAT are kept, in separate sectors.

The FAT maps the allocation of clusters. The directory entry for a file points to the first cluster of the file. The FAT entry for that cluster contains the cluster number for the next cluster of the file. Each cluster points to the next, except for the last one, which contains an end of file marker. Unallocated clusters have a FAT entry of zero, since cluster zero cannot be used (it holds the boot record). Bad sectors are marked in the FAT to prevent their being allocated. The entries in the

FAT may be either 12 or 16 bits, depending on the size of the disk (disks of 10 Mb or less use 12-bit FAT entries). The end of file FAT entries are in the range FFF8 to FFFF (FF8 to FFF for 12-bit FATs). The reserved FAT entries are in the range FFF0 to FFF7 (FF0 to FF7 in 12-bit FATs). Bad sectors are marked reserved with FFF7 (FF7 in 12-bit FATs).

A disk volume has a boot record in the first sector. Following this is the first copy of the FAT, then the second. After the FAT comes the root directory, which is of fixed size, with the size depending on the size of the disk volume. Among the directory entries can be entries for subdirectories, which are in fact files containing directory entries. Subdirectories are variable in size, expanding as necessary to hold additional directory entries. The root directory can only hold a limited number of entries.

A directory entry is 32 bytes in length; its format is specified in the DOS Technical Reference manual. A directory entry holds the name of the file, its attributes, the date and time it was last changed, the size of the file in bytes, and the cluster number of the first cluster in the file.

The attribute of a file identifies the kind of file it is. The attribute is a single byte, with bits indicating read-only, hidden, system, volume label, subdirectory, and archived attributes. Many of these can be combined.

The readonly bit prevents the file from being opened for write, being deleted, or being truncated to zero by a create.

The hidden bit conceals the file from normal directory searches. You can hide your files or directories in this way, using a tool such as the Disk Repair program of the Professional Debug Facility.

The system bit does the same as the hidden bit, but is intended for system files only. Files like IBMBIO.COM and IBMDOS.COM are hidden from directory searches in this way.

The volume label bit is only valid for the volume label entry in the root directory. The volume label is a name identifying the disk or diskette.

The subdirectory bit indicates that the file is a subdirectory, that is, that it is a file containing directory entries.

The archived bit marks a file as having been backed up -- it is switched off every time a file is changed. The DOS BACKUP command refers to the archive bit when asked to do a partial backup, that is, a backup of all files which have changed since the last backup.

Files in a DOS system can be fully named by specifying their disk volume, path name, and filename. The disk volume name is a single letter, usually followed with a colon to identify it. The path name is the sequence of subdirectory names, starting from the root directory, that has to be followed to find the file; the subdirectory names are usually separated by backslashes to indicate where one ends and the next begins. The

filename is in two parts, a name of up to eight characters, followed by an extension of up to three characters; the two parts are usually separated by a period. This full path- and filename can be abbreviated to include only a portion of the path plus the file, or indeed only the filename itself. DOS has the concept of a default, or current, disk volume. Any reference to a file that does not specify the disk volume on which the file resides is assumed to refer to the current disk volume. For each disk volume there is a current directory. This may be any directory on the volume. Any reference to a file on that volume (either explicitly, or implicitly by way of the current disk volume) is assumed to refer to the current directory, unless the pathname contains a backslash, which indicates a full or partial pathname. You should already be familiary with the format of DOS directories and paths if you have met the prerequisites for this course. If not, there are several tutorial packages available for learning DOS directory structuring.

Files can be created only in existing directories. An attempt to specify a pathname which contains a nonexistent directory name will fail. Similarly, an attempt to remove a directory that still contains files will fail.

DOS provides a number of function calls to allow the allocation, manipulation, and deletion of files. None of these calls requires a detailed knowledge of the disk format, just an understanding of how DOS regards a file. To DOS a file is a sequence of bytes. Although consecutive sectors of a file may be scattered all over a disk volume DOS masks this from the program. To a program the file appears contiguous.

A program may request anything from 1 to 65535 bytes at a time from the file, and DOS will fetch that amount into the specified buffer, then advance the read/write pointer past that piece of the file. If the file does not have that much information left before the end of the file then DOS will fetch what remains, place the read/write pointer at the end of the file, and inform the program how much was returned. This simple approach allows a program to fetch large buffers of information for processing in memory, or to read the file record by record, or even character by character. One of the effects of this mechanism is to provide what might be regarded as automatic blocking and deblocking of the file; the program is completely ignorant of the sectors of the physical disk.

The program may alter the read/write pointer to anywhere in the file, even beyond the end of the file, if writing. This allows completely random access to the file contents. DOS provides no structure to the file, but the facilities provided make it quite easy to build quite complex access methods, including indexed sequential, relative record, key sequenced, and all manner of indices. DOS does not provide insert or delete functions for adding data to the middle of a file or removing data from the middle -- these must be handled by a program.

If a program asks DOS to write zero bytes to a file the length of the file will be changed, but no information is written. This is really only useful when a program wishes to shorten a file that it has opened for read/write. All the information past the current position of the read/write pointer is discarded.

7.7 MEMORY ALLOCATION

DOS manages the allocation of memory. When a single program is loaded this is a simple matter. All free memory is allocated to the program when it is loaded, and freed when it terminates. Complexities creep in when considering resident programs, programs which load other programs, and programs which use dynamic memory allocation (dynamic memory allocation has nothing to do with DMA -- Direct Memory Access). DOS was built with these complexities in mind, and so handles them appropriately.

Memory in a DOS system is handled in chunks. Each chunk is prefixed with a memory control block, which indicates the size of the chunk, whether it is free or allocated, and to what program the chunk is allocated. Memory is allocated in paragraphs, meaning that only a segment value is necessary to address the chunk. When a request to allocate storage is processed, DOS finds a chunk of memory large enough to satisfy the request and carves off the requested amount, splitting the chunk into two pieces, one allocated, and one free. If there is insufficient storage to satisfy the request available, DOS will tell the requestor how much is available -- the size of the largest free block.

When a program is loaded all free memory is allocated to the program. Before it can issue any requests requiring the allocation of storage it must free some. Typically, a program will free all storage that it is not using.

Free storage is required if a program wishes to use dynamic memory allocation, or if it wishes to load and execute another program. Dynamic memory allocation is convenient in a program which can use large buffers, because the buffers can be made as large as the free memory.

A resident program is one which remains resident in memory after control is returned to DOS. A resident program typically occupies a small amount of memory and performs some specialized task. An example is a "hotkey" program, which remains resident in memory and can be invoked at any time by a special keystroke sequence. Such a program monitors the keyboard and interrupts whatever program is running when the "hotkey" is recognized. The hotkey program may provide support lacking in normal programs, such as printer setup, or keyboard macros.

A program makes itself resident by using a "terminate and remain resident" call to DOS. This instructs DOS not to free all of the memory that was allocated to the program, but rather to leave the part that contains the resident code allocated. Any program that DOS loads after this will not be loaded on top of the resident program. Control is passed back to DOS, usually bringing up the command interpreter again.

7.8 PROGRAM INTERFACE

A program may request a number of services from DOS. These services fall into several broad categories:

- file, disk, and directory management: a program need not worry and the details of getting data to and from disk - DOS will handle it

- memory management: DOS handles the allocation of memory. A program not using DOS to handle memory allocation is likely to crash the system

- program management: program loading, termination, and checking of return codes

- miscellaneous resource management: things like the date and time, country data, network setup, and the state of many system flags. One of the more important resource management features is the getting and setting interrupt vectors.

Each of the functions that DOS can perform for a program is documented in considerable detail in the DOS Technical Reference manual, including the restrictions, requirements, and possible error conditions of the call.

A function is requested by placing the number of the function into the AH register and issuing interrupt 21h. Parameters to the function are passed in other registers. DOS 3.1 offers 82 distinct function calls, many of which are obsolete, and present only for programs which were written for an earlier version of DOS. Services can also be requested from DOS by way of certain other software interrupts (like interrupt 20h), but all of these have been made obsolete by more powerful function calls (interrupt 20h has been rendered obsolete by function call 4Ch). A programmer should only use the new function calls, as this allows a program to take advantage of the advances since DOS 2.0.

7.9 FILE, DISK, AND DIRECTORY MANAGEMENT

DOS provides function calls to manipulate all aspects of the file system. All of these are documented in the DOS Technical Reference manual. A program can create, open, read, write, close, delete, and rename files. It can change the current disk volume, and get information about any disk volume. It can create, search, and remove directories, as well as getting or changing the current directory.

7.10 MEMORY MANAGEMENT

DOS provides functions to allocate and free memory, and to change the size of allocated blocks. These functions must be used to avoid corrupting the memory control blocks. Corrupted memory control blocks lead to a system crash.

7.11 PROGRAM MANAGEMENT

DOS allows a program to load and execute other programs by using the appropriate function calls. Other function calls are provided to allow the program to return control to DOS, with or without a portion of the program remaining resident. The program can pass a completion or return code to DOS at termination time, indicating successful or unsuccessful execution. Where a program has loaded and executed another program it can retrieve the completion code of that program.

7.12 MISCELLANEOUS RESOURCE MANAGEMENT

DOS controls a number of resources other than the file system, memory, and programs; function calls are provided to access these. The system date and time, the country information, the DOS version number, and the Control-Break checking flag are amongst the available resources. The network resources, like network names and the network printer setup, are equally important. Interrupt vectors can be read and set using DOS function calls -- this allows programs to redirect an interrupt vector to point to an alternate interrupt routine, or to figure out where an interrupt vector is currently pointing to. Extended error codes can be retrieved, complete with advice on the action to take. All of these facilities are available to a program, and should be used where appropriate, rather than accessing the hardware directly.

7.13 TYPES OF PROGRAM

DOS supports two kinds of executable program files, known by their file extensions: EXE and COM. These two types of programs have complementary attributes. An EXE program is essentially unlimited in size, but takes a while to load. A COM program loads quickly, but is limited to 64K in size. The reason for these differences lies in the segmented architecture of the 8088.

Once a Program Segment Prefix for a COM file is built, the file can be loaded into memory and run immediately. (Hold on! The PSP gets covered in a moment.) All that DOS has to do to load a COM file is to set all of

the segment registers to the segment holding the program, set IP to 100h, and SP to FFFFh, then push a zero word. No relocation is required.

An EXE file has a special header built into it. This header specifies a number of things, including the location of the stack segment, the start of the executable program, and a number of relocation entries. The relocation entries specify where the program has mentioned segment values. The program loader has to alter each segment reference to the correct value before starting to execute the program. This, however, allows the program to contain a large number of segments, which means that it can be larger than 64K in size.

Typically, COM files are used for small utility programs, where their speed of loading is an advantage. They are also used for resident programs, where the fixed layout is easier to handle than the EXE format. Extra care is required to build a COM file, because segment fixups must be avoided -- effectively the program consists of a single segment.

Typically, EXE files are used for large programs. The output of most compilers is converted into an EXE file. An EXE program is easier to build in that multiple segments can be used, allowing the programmer (or compiler writer) to take advantage of the 8088 separation of code, data and stack.

When a COM file is to be loaded, the program loader (part of the resident portion of COMMAND) allocates the memory the program will be loaded into. It builds the PSP for the COM file in the first 256 bytes of the memory, and loads the COM file immediately after it. All of the segment registers are pointed to the start of the memory, where the PSP is, and IP is pointed just past the end of the PSP, where the first instruction of the COM file must be located.

When an EXE file is to be loaded, the program loader allocates the memory for the program and builds a PSP in the first 256 bytes. The EXE file is loaded after the PSP, and then processed for relocation. The relocation process actually modifies all the locations in the program that refer to absolute segment addresses. When that process is complete the SS register is pointed to the program's stack segment (listed in the EXE header), the DS and ES registers are pointed at the PSP, and the CS register is pointed at the segment address of the program's defined entry point (also listed in the EXE header). IP is set to the offset address of the entry point, and the execution of the program begins.

The two types of program are otherwise equivalent -- both can make use of DOS and BIOS calls freely.

7.14 PROGRAM LOADING

The loading of programs by DOS is a little more complicated than described above. The first thing the program loader has to do is find a free chunk of memory sufficiently large to hold the program. If this is not found

then an error message is issued. Once a free piece of memory is found, the loader builds a Program Segment Prefix (PSP) at the low end of it.

The PSP is a 256-byte control block containing a lot of information about the way in which the program was invoked. The parts of interest to the programmer are:

offset	contents
2C	segment address of the environment for this program (The environment is a string area in memory which describes some of the operating characteristics of the program. The environment will be discussed in a moment.)
80	length of the parameter list
81-FF	parameters on the command line after the program name

There is a lot more information in the PSP, but a programmer using DOS 2+ function calls is unlikely to use it. All the details can be found in the DOS Technical Reference manual.

Having built the PSP, the loader fetches the program from disk into memory. If the program is a COM file, the segment registers, SP, and IP, are all set as described above, and control is passed to the program. If the program is an EXE file, all its relocation information must be processed first. This process is covered in detail in the DOS Technical Reference manual. The relocation information covers only segment relocation -- all else is handled during the linking of the program.

A loaded program is passed all of its parent's open files, and a copy of its parent's environment. The parent of a program is the code that requested the program be loaded -- most programs have the command interpreter as their parent. The fact that a program is passed a copy of the parent's open files allows a parent to redirect the I/O of a loaded program. The parent opens the files it wishes the program to use, then requests that the program be loaded. The program is loaded, uses the open files, then returns to the parent. Normally the only files which are redirected are the standard I/O handles, because these are the only ones a program is likely to use without opening - this is how the command interpreter redirects I/O.

The environment is a list of strings (up to 32K total) which contain text entered using the SET command, plus a copy of the current PATH, and the specification of the command interpreter. At a minimum the environment contains a string (eg. "COMSPEC = c:\command.com") consisting of the specification of where the command interpreter was loaded from.

7.15 PROGRAM LINKING

The DOS program linker converts the output of compilers and assemblers (object code) into an EXE file. In this process it takes one or more OBJ

files and resolves all external references. One or more libraries of object modules can be scanned to find required code, if requested.

The output of a compiler or an assembler is not executable -- it is called an object file (with an extension of OBJ), and is specifically intended for processing by the program linker. Even a program consisting of a single OBJ file must be processed by the linker before it can be executed. The linker takes in OBJ files and produces EXE files. A program which is intended to be an EXE file can be run after linking. A program which is intended to be a COM file must be processed by the EXE2BIN utility before it is ready - the EXE2BIN process removes the EXE file header, and checks that the program does not violate any of the strictures placed on COM files. EXE2BIN stands for "EXE to BINary".

The link process deals mainly with the combination of segments and the resolution of external references. An OBJ file contains pieces of one or more segments, each with a name, a combine-type, a class, and possibly a group name. The relocation entries describe every point in the file that refers to a segment relative address. Additionally, it may contain a table of external and public references, listing all the locations in the file that refer to or define external entities.

The linker combines all the pieces of segment of the same name, concatenating them or overlaying them, depending on the combine-type of the segment. It then combines all the segments of a group, and then all the groups of a class. Once this is done it processes all the offset relocation entries, adjusting offsets to allow for the combination of segments -- something that was at offset 4 may end up at offset 5A4, if a piece of segment 5A0 in length is included before it.

The linker attempts to resolve external references from the OBJ files it is told to process. An external reference may be the call of a procedure in one OBJ file by a procedure in another OBJ file, or it may be a reference to a data element in one OBJ file by a procedure in another OBJ file. If an external reference cannot be satisfied within the OBJ files the linker is processing, it may look in a library, if any libraries have been made available to the linker. A library is a collection of object modules which the linker can search. Libraries are frequently used by high-level languages to hold common routines, like I/O service routines.

7.16 ERROR HANDLING

Errors will occur, and DOS has to handle them. The kinds of errors DOS handles range from the familiar "Bad command or filename" to the upsetting "Read error on disk A: - Abort, Retry or Ignore?". DOS handles these errors in a way intended to protect the user and his/her data.

Different errors are detected in different parts of DOS. There are three categories of errors:

- command interpreter errors

- function call errors

- critical errors

Command interpreter errors are errors detected by the command interpreter while it is attempting to execute a user command, either directly from the command line, or from a BAT file. These errors are generally not serious, normally just a slip of the fingers. The errors include misspelled commands ("Bad command or filename"), too little free memory to execute the requested command ("Insufficient memory"), and BAT file errors.

Function call errors are detected by IBMDOS. They generally occur when a program makes a function call to DOS that cannot be successfully completed. These errors are indicated by a return with the carry flag set. Normally these errors are the concern of the programmer, and are only reflected to the user when some action is required to fix the problem.

Critical errors are detected by DOS during I/O processing and always result in the invocation of the Critical Error handler. This handler is on interrupt 24h. Unless the critical error handler is changed by a user program it displays the "Abort, Retry or Ignore" message. A program can change the Critical Error handler to perform whatever processing is desired. The information necessary to do so is contained in the DOS Technical Reference manual.

One circumstance is handled by DOS as an error condition, and yet is usually not indicative of an error -- the user pressing Control-Break. DOS detects this condition by checking the next character to be read from the keyboard buffer, but without actually reading it (this is why the Control-Break will only be detected when no other character is in the keyboard buffer unread). DOS can check every time it gets control -- provided the BREAK flag is set on -- or it may check only when doing I/O using the console driver (screen and keyboard) -- provided the BREAK flag is set off. The entry of Control-Break causes DOS to invoke interrupt 23h. DOS supplies a default handler for the interrupt, which aborts any currently executing program and queries whether to abort an executing BAT file. A program can replace the handler if desired, to provide appropriate processing.

7.17 DEVICE MANAGEMENT

One of the functions of IBMBIO.COM is to load any device drivers specified in CONFIG.SYS, and then to initialize the default device drivers. Each device driver in a DOS system is of one of two types: character or block. A character device is a simple one, handling a single character at a time -- a typical example is the keyboard. A block device driver handles a block of data at a time -- the most common example is a disk driver. DOS provides drivers for the standard IBM keyboard (plus national language versions), the standard IBM screens, IBM disk drives, and general purpose asynchronous and printer drivers. Two special purpose drivers are sup-

plied with DOS -- ANSI.SYS, which is an ANSI standard screen and keyboard driver, and VDISK.SYS, a RAM disk simulation driver. Drivers for any other hardware must be supplied by someone else. The complete specification of the requirements of a device driver can be found in the DOS Technical Reference manual.

IBMBIO loads each device driver and passes control to its initialization point. Character device drivers specify their own names, and IBMBIO records these to enable it to pass control to them when they are requested. If a character device driver specifies the name of a default device driver then the default driver will not be installed. There are a number of default character device drivers built into DOS:

```
CON: -- the screen and keyboard (console) driver
PRN:, LPT1:, LPT2:, LPT3: -- printer device drivers
AUX:, COM1:, COM2: -- serial port drivers
NUL: -- the bit bucket
```

Block device drivers are identified by single letters, and are given their identifiers by the order in which they are loaded. The first block device driver to be loaded is usually the diskette drive device driver, which supports two (or up to four) devices, and so gets letters A and B (C and D as well if four diskette drives are present, but this is not usual). If a hard disk is present its driver is usually loaded next, and so it gets letter C. Assuming that there is only one hard disk present, a VDISK driver which is loaded next would get letter D, and so on. This is the allocation system for block device letters. It is not possible for a block device driver specified in CONFIG.SYS to replace the default disk drivers.

7.18 SUMMARY

This concludes the main DOS module. Because many programs and services are available to programs which are nevertheless not an intrinsic part of DOS, we have devoted a separate module to these DOS extensions. At this point, you should understand the following points about DOS:

- what IBM PC DOS is and what it does
- the IPL process in a PC and the components of DOS:
  - IBMBIO
  - IBMDOS
  - COMMAND
    - resident portion
    - initialization portion
    - transient portion / command interpreter
- the process of reading a command
  - the kinds of command: internal, external and batch

    - redirection of I/O

- the disk format used by DOS

    - the Boot record
    - the File Allocation Table (FAT)
    - the directory structure

- the file management facilities available under DOS

- memory allocation under DOS

- the DOS program interface

    - DOS function calls
    - resource management

- the types of program DOS supports

    - COM programs
    - EXE programs
    - BAT batch files

- the support that DOS provides for programs

    - program linking
    - program loading
        - the Program Segment Prefix (PSP)
        - relocation of EXE files
        - the environment

- DOS error handling

- device management in DOS

    - device drivers
    - block devices
    - character devices

The DOS module is the largest module in the course in terms of class time allocated to it. In addition, a DOS lab is conducted with class students to familiarize them with DOS functions through hands-on experience. If you are studying this book on your own we urge you to use the DOS lab to improve your DOS knowledge and reinforce the materials presented here. The DOS lab is provided in Appendix E.

Once you have completed the DOS Lab, you can proceed to the DOS Extensions module. Alternatively, you may pursue the DOS Extensions module first, and then conduct the DOS Lab.

8.1 OBJECTIVES

In this module we will discuss some of the the ways in which DOS can be enhanced and extended. There are tasks normally associated with an operating system which DOS does not do, and there are others which DOS does perform that could be done in other ways. At the end of this module, you should understand the following:

- The kinds of extensions that exist and why
- What multitasking is and how it can be implemented
- How we can extend DOS support of devices through device drivers
- How we can enhance DOS with other types of resident extensions

8.2 AGENDA

In order to accomplish these objectives, we will proceed as follows:

1. Define the areas where DOS can be enhanced and extended.
2. Look at the available IBM multitasking extensions.
3. Learn how DOS arranges and uses device drivers, and how we can replace or add new device drivers.
4. Look more closely at the construction and function of device drivers.
5. Look at other types of utilities which enhance the operating system.

8.3 WHAT KIND OF EXTENSIONS DO WE HAVE AND WHY?

Up to this point in the course, we have looked at different levels of application and end-user support. We have seen that we can control the PC on different levels. Using the BIOS and hardware ports directly, we are able to do almost anything with the PC, but it may take us a long time to do the coding. DOS and some high level languages make it easy to control the PC in a relatively short time. Why would we want to extend DOS?

The answer is naturally that DOS does not support all the functions we want. Examples of that could be:

- MULTITASKING - We want to use the keyboard and display for editing a note while the PC is busy sending a file via a modem to another computer. The processor in the PC has the capacity to do both things at once, but DOS does not support it.

- DEVICE SUPPORT - DOS contains support for the different devices it is announced to support. If we want to add some other type of equipment to our PC, a drawing tablet or a mouse for example, and we also want our programs to use normal DOS function calls to talk to this device, then we have to write a device support program or driver.

- OTHER ENHANCEMENTS - When we are not satisfied with the support within DOS for a device or application, then we may want to replace that function with our own code. A common example of this are the so-called keyboard enhancers -- programs which replace the normal keyboard handling routines in DOS and allow a string of commands to be invoked with a single key.

We will now take a look at the current IBM multitasking offerings.

8.4 MULTITASKING EXTENSIONS

A multitasking environment is one in which several programs can co-reside in the system, in which software or hardware mechanism let each program share the system's processing and I/O resources.

The complexity of such a system can range from a simple print spool program running at the same time as we edit a text file with an editing application under DOS, to a system with many applications running in parallel, with a scheduling program to control resources and priorities between programs and a dispatcher to control which program should execute at a given time.

We will begin our examination of multitasking by examining a print spool program which is actually part of DOS -- the PRINT command.

8.4.1 PRINT

The DOS PRINT command loads itself into storage as a resident program the first time we enter the command "PRINT". It intercepts the timer tick interrupt which is issued 18.2 times per second, and in this way it is called at each timer interrupt. After each time it is invoked, the PRINT program will decide whether it wants to use part of the current time slice itself (the time slice is the processing interval between the most recent and the next timer ticks) or whether it will return control to the program which was interrupted by the timer. (This program could even have been the PRINT program itself if it was interrupted while printing something).

The percentage of processor time during which the PRINT program is in control can be defined by the user when the program is started. The PRINT program will naturally not use any of its allotted time slice if the print queue is empty.

There is only one way to communicate with the PRINT program, and that is via the PRINT command. Every time the PRINT command is used, it will check to see whether it has already been loaded into storage, and if it has it will not install itself again, but will simply pass information to the installed version, retrieve any required information back, and on exit will return this information to the user.

8.4.2 TopView

With TopView you can run several applications at once on a single PC, and simultaneously view portions of the display of each of these applications. TopView accomplishes application multitasking by using a time-slicing technique, and performs multiple-application display with a windowing system.

Multitasking: The time-slicing technique operates by intercepting the timer-tick interrupt. Each time the timer issues a timer tick (18.2 times per second), TopView gains control. It checks a list it maintains called the dispatch list or dispatch queue, which contains information about all the currently loaded tasks. Based upon the status of each of these tasks and upon an undocumented priority algorithm, it will select either to begin a new task, or to resume execution of the task which was interrupted by the timer tick.

Each time TopView gains control at a timer tick, it must save the environment of the executing task. To do this it allocates several storage areas for each loaded task, and in one of these storage areas it automatically saves the contents of all the processor registers and flags as they were when the task was interrupted. It also saves the interrupt vectors for any interrupts the task has supposedly changed. For example, if an application has changed the execution address of INT 75H, TopView will save the vector pointing to that address. TopView will save other environmental components as well, as documented in the TopView manuals.

When TopView decides to give control to a different task, it will reset the interrupt vectors for that task so that they are vectored as they were when the task was last interrupted. TopView restores the contents of the registers and re-establishes the operating environment of the program; it then allows the task to continue running wherever it left off.

This time-slicing is complicated by the concept of I/O-driven multitasking. If a hardware input event occurs, it is important to dispatch as soon as possible the task for which the I/O event was destined. Similarly, if an application requests input which is not available, there is no point in wasting processor cycles executing the application's wait for input. Therefore, when an input event occurs (for a background application, it

could come from a serial port, for example), TopView gains control, saves the environment of the current task, and loads the task for which the input was destined. If an application requests input which is not available (for example, a background application requests keyboard input, or the foreground application requests keyboard input when the user is not typing), TopView stops that task and dispatches another task. It will not allow the stopped task to resume execution until the requested input arrives.

This dual method of multitasking is called "Time-slicing with natural program breaks". The result of this technique is that I/O-bound programs will not hog the processor while waiting for the next I/O operation, and compute-bound programs will be able to soak up all processor capacity during I/O waits. Such a multitasking method ensures maximum utilization of the computer. TopView does not allow you to select the number of time slices to give to an individual application; however, you can choose to suspend an application by using the Suspend command. A suspended program does not receive time slices until you resume it.

Windowing: TopView accomplishes windowing by intercepting DOS and BIOS video function calls. If you have chosen to view several applications at once, you could, for example, have the upper right portion of one application displayed in a window on the lower left; the bottom half of another application's logical display could be shown on the upper half of the physical display. When an application issues a DOS or BIOS video call, TopView intercepts this call. It analyses the type of request, and updates the window for that application as required. If a character is to be displayed in an area of an application's display which is not mapped onto its current window, TopView will remember that the character was displayed there, but will not display anything. If an application requests that its screen be cleared, only the portion of the physical screen contained in that application's window will be cleared. In addition, TopView will remember that non-displayed portions of the screen have been cleared, but these portions will not be displayed unless the size of the window is changed.

We mentioned in several previous modules the importance of not using direct screen addressing in an application, and now we can understand one reason why. TopView intercepts BIOS and DOS video calls, decides on what area of the physical screen to map the call, and then does direct screen addressing itself to effectuate the call, provided the call results in an update of the physical screen. If an application were to write directly to the screen, it might overwrite the contents of other windows. For this reason, programs which do write directly to the screen are not permitted to run in the background, and when they run in the foreground they are given control of the entire screen (that is, no other windows are displayed).

Programs which want to write directly to the video buffer when running outside of TopView may also do direct writing within TopView and still be regarded as well-behaved (that is, still be able to run in the background and to be windowed). TopView supports two special BIOS video functions which allow windowing of direct-video-addressing applications:

- Get Video Buffer (INT 10H, AH=0FEH)

  This enables the program to get the address of an alternate video buffer. If the program is running under TopView, the address of an alternate buffer is returned, usually a segment in user RAM. If the program is running under normal DOS, the physical video buffer address is returned.

  If the program finds the normal physical buffer address returned, it may just write to the buffer directly. If the address is different from the normal buffer address, this means TopView is loaded, and when the program wants to change the display, it writes to the buffer address returned by TopView.

- Update Video Display (INT 10H, AH=0FFH)

  This call is used to tell TopView what parts of the memory screen buffer have been updated, so that TopView can map the changes to the physical screen.

Application Program Interface: In addition to performing windowing and multitasking functions for most DOS-based applications, TopView supports a special interface for programs written specifically for TopView. This interface allows TopView-specific programs to display multiple windows for a single task, to perform task management functions like forcing another task into the background, and to use TopView's full-screen-input functions. We will identify several types of routines below.

- Window management

  An application can request that its window be moved or changed in size. It can request that an additional window be created for that application (only TopView-specific applications can have multiple windows). It can request that TopView display a specific type of input window, and return the input to the user.

- Full-screen input

  Application programmers can use the Window Design Aid, provided with the TopView Programmer's Toolkit, to design windows, including windows of full-screen-input fields. When the programmer designs a full-screen-input window, the length and appearance of each field, as well as the valid ways of entering information into that field, are designed with the Window Design Aid, and a file is created to store the window. The application need only provide the name of the appropriate window to TopView, and set up the necessary request, and TopView will retrieve all the requested input from the user and return it to the application. This not only reduces programmer time spent on console I/O routines, but allows for a TopView-like interface so that programs can blend more easily into the TopView environment.

- Task management

Programs can request that they be made the foreground task; they can ask to be suspended. They can declare a section of their code as critical, so that TopView will not time-slice it out; they can force another task to execute a specific routine. They can initiate other tasks -- one application can actually have multiple tasks working for it.

The TopView Programmer's Toolkit manual provides detailed information on supported TopView API (Application Program Interface) functions.

8.4.3 PC Network Program

The PC Network Program allows multiple computers hooked together on a PC Network or PC Token-Ring Network to share resources and exchange information. The PC Network Program can be considered a DOS extension insofar as it allows you to assign resources on a server computer as local resources -- in other words, you can access information on someone else's C: drive by calling that drive your K: drive and reading from it just as you would in stand-alone DOS. You can also name a print server's printer to be your LPT1, LPT2 or LPT3 so that you can print data on someone else's printer.

If we consider the PC Network Program as a DOS extension we must think of it in terms of how it intercepts certain DOS function calls -- in particular printer- and disk-related calls -- to redirect printer, file and directory I/O from the local computer to the server who owns the resource being addressed. In addition to intercepting these types of calls, the PC Network performs a multitasking of its own so that applications can run at the same time as resources are being shared between computers. Like the DOS PRINT command and TopView, the PC Network program uses the timer tick to interrupt applications in order to allow processing of network jobs such as printing and disk access.

Several ITSC Red Guides have already been published on the PC Network Program, so if you want additional information you may obtain one of these. The important thing to remember in terms of the PC Network's place in DOS extensions is that the PC Network Program works together with DOS by providing logical DOS device support for remote resources. As such, one might consider the PC Network Program to be a sophisticated device driver for a wealth of different devices.

8.4.4 3270/PC Control Program

The 3270/PC Control Program performs multitasking and windowing in a similar way to that employed by TopView. Some of the differences, however, are worth noting.

The Control Program is not running under DOS. It is loaded by DOS, and upon obtaining control it hides itself at the top of physical storage. It then changes the "Top of Memory" indicator in low storage and re-IPL's DOS, which loads into the now reduced storage area. To be able to control the PC and DOS it then replaces the necessary interrupt vectors which point to DOS with vectors pointing into the Control Program code.

Since we are in fact running DOS under the Control Program, we are perhaps stretching our credibility in calling it a real DOS extension.

The 3270/PC is also using other screens and screen adapters than those supported by DOS. This means that when DOS writes to the screen buffer addresses, the CP has to map this data out to the physical screen in its own way.

The API (Application Program Interface) for the 3270/PC is also very powerful, but also quite different from the TopView API.

8.5 DEVICE DRIVERS

8.5.1 What is a DOS device and a device driver?

Exactly what is a DOS device? A device can be thought of as a place to and from which bytes travel. In most cases this place has a physical counterpart -- a hardware peripheral of some kind -- that is actually generating or using the bytes that are coming from or going to the device. A display or a printer are examples of physical devices which receive bytes, a keyboard is a device that emits bytes. An example of a non-physical device is the DOS NUL device. It functions as a "bit-bucket" and just eats anything coming into it.

Two types of devices exist: character and block. Character devices send or receive bytes one at a time. Block devices are generally disk drives or disk drive emulators, and they send or receive a block of bytes at a time. A block typically consists of a physical sector of the disk drive.

Device drivers are programs created to control particular devices. They have a fixed, specific format and are loaded by DOS during IPL. They give application programs a consistent interface to different devices through DOS.

DOS has some default device drivers which are initialized during IPL, and DOS also gives us the capability to add our own device drivers or to replace existing ones. DOS contains default drivers for support of the NUL, CON, AUX (COM1), PRN (LPT1), CLOCK$, LPT2, LPT3, and COM2 character devices plus two to four (A, B and eventually C and D) block devices.

If we write our own device driver and get DOS to load it for us, how do we then use it?

The answer is that you treat it exactly like other DOS devices: CON, LPT1, COM1, A, B, C, etc. If the device driver is written properly, you should be able to issue commands to the new device, or open it for input and output, just like any other DOS device. The device driver should make the input and output from the device transparent to the user.

8.5.2 Installation of the device drivers

The Device Chain: When DOS loads the device drivers during IPL, they are connected into a one-way linked list. Each node in the list is an 18-byte data structure called a device header. The first four bytes of each device header comprise a 32-bit pointer (segment:offset) that points to the next node in the list.

The first node is embedded in IBMDOS.COM and corresponds to the NUL device, a "bit-bucket" device that absorbs everything sent to it and emits nothing. The last node is identified by a value of 0FFFFH in the offset portion of the 32-bit pointer to the next node. You can see what the chain looks like in Figure 1 on page 119.

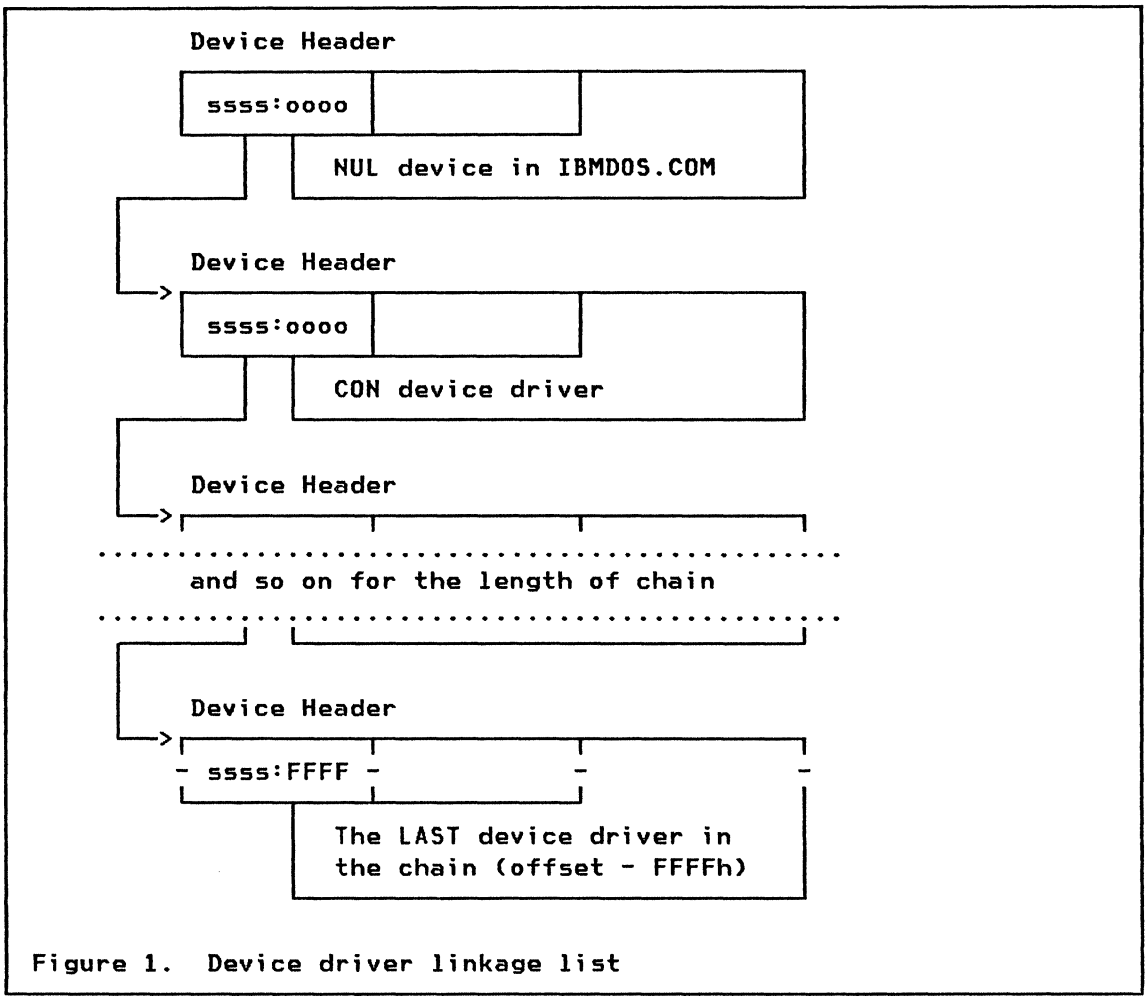


Figure 1. Device driver linkage list

The device header has five parts as shown in Figure 2:

Next Header Ptr		Attribute word	Strategy Routine Offset	Interrupt Routine Offset	8 character Device Name
Offset	Segment				

Figure 2. Device header

1. Chaining pointer to the next device driver header in the chain.
2. Attribute word
3. Offset pointer to the strategy routine entry point (the segment part of the pointer is taken from the chaining pointer in the previous device header).

4. Offset pointer to the interrupt routine entry point (the segment part of the pointer is taken from the chaining pointer in the previous device header).

5. Eight character field for the device name.

The fields will be explained in more detail shortly.

Besides NUL, which is always present, DOS requires a minimum of four devices in the system-initialization code. These devices are those that follow NUL in Figure 1 on page 119: CON, AUX, PRN, and CLOCK$. The device names are unimportant, but the function each performs must correspond to the following sequence: standard I/O device, auxiliary I/O device, listing device, and the realtime clock.

After these four devices have been installed, additional resident devices are installed. These resident devices are used to initialize DOS. Figure 3 shows what the device header chain looks like after DOS has installed all resident device drivers. Note that the address in the Next Header field for each device becomes the starting address for the device that follows in the device header chain.

	Device Header				
At memory address	Next Header	Attr	Strategy Entry pt.	Interrup Entry pt.	Device Name
0133:0248	0082:03D5	8004	1680	1686	NUL
0082:03D5	0082:0231	8013	00B6	00F8	CON
0082:0231	0082:02A6	8000	00B6	00FE	AUX
0082:02A6	0082:0345	8800	00B6	010D	PRN
0082:0345	0082:0416	8008	00B6	0122	CLOCK$
0082:0416	0082:0110	0800	00B6	0128	4 (disks)
0082:0110	0082:0307	8000	00B6	00FE	COM1
0082:0307	0082:0B56	8800	00B6	010D	LPT1
0082:0B56	0082:0B80	8800	00B6	0113	LPT2
0082:0B80	0082:0BA6	8800	00B6	0119	LPT3
0082:0BA6	0082:FFFF	8000	00B6	0104	COM2

Figure 3. Device header chain with the IBM default device drivers

When installation is complete, an attempt is made to open CONFIG.SYS on the IPL drive. If CONFIG.SYS exists, it is read into a buffer and the contents are parsed into commands.

The command line for adding a device is as follows:

DEVICE=(d:) (path) filename.ext (parameters)

where

(d:)	is the name of the drive where the device driver program is located.
(path)	is the path to where the device driver program is located.
filename.ext	is the name and extension of the device driver program.
(parameters)	information to the init part of the device driver program (everything after the equal-sign is passed to the program).

The items within parentheses are optional and include drive and path specifiers as well as a parameter field following the file name.

When a new device is installed, it is added to the linked list at the link immediately following the NUL device. This addition is done in two operations:

- the Next Header pointer in the NUL device header is moved into the Next Header field of the device header to be installed.
- the Next Header field in the NUL device header is replaced with the segment:offset address of the newly installed device header and driver.

New drivers are thus added to the linked list at the root (NUL), pushing previously-installed drivers further down the list. You can se the result in Figure 4 on page 122 after installation of the device driver for device MYPRN

At memory address	Device Header: Next Header	Attr	Strategy Entry pt.	Interrup Entry pt.	Device Name
0133:0248	129F:0000	8004	1680	1686	NUL
129F:0000	0082:03D5	C000	0030	0180	MYPRN
0082:03D5	0082:0231	8013	00B6	00F8	CON
0082:0231	0082:02A6	8000	00B6	00FE	AUX
0082:02A6	0082:0345	8800	00B6	010D	PRN
0082:0345	0082:0416	8008	00B6	0122	CLOCK$
0082:0416	0082:0110	0800	00B6	0128	4 (disks)
0082:0110	0082:0307	8000	00B6	00FE	COM1
0082:0307	0082:0B56	8800	00B6	010D	LPT1
0082:0B56	0082:0B80	8800	00B6	0113	LPT2
0082:0B80	0082:0BA6	8800	00B6	0119	LPT3
0082:0BA6	0082:FFFF	8000	00B6	0104	COM2

Figure 4. Device header chain with one installed device

Once the CONFIG.SYS file has been completely parsed, all of the initial standard device handles are closed and then reopened so that user-installed device drivers can preempt the default drivers for CON, AUX, and PRN. When a device handle is opened, the first device in the linked list that matches its name is used to satisfy the request. This allows duplicate device names to be included in the CONFIG.SYS file for CON, AUX, and PRN.

Because installed devices are inserted immediately after the head of the chain, a user-installed CON, AUX, or PRN device driver will be found before the DOS-supplied default drivers for these devices. Note that the position of the NUL device prevents another NUL being loaded before it. NUL is thus the only device that cannot be preempted.

8.5.3 Communicating with Device Drives

You use a device driver in order to convey commands to a "smart" physical device, to make a "dumb" hardware device look "smart" to the system, or to pass non-system information to the device. Communication can take two forms: byte stream communication or control channel communication.

In byte stream communication, the driver must recognize some kind of escape character in the byte stream and use the characters that follow the

escape as control characters. For example, byte stream communication is used when positioning the cursor with the ANSI.SYS device driver.

Control channel communication requires that the driver recognize a hardware control channel. In control channel communication, you:

- Build the message.
- Build the device driver name.
- Create the handle (file control block).
- Send a message to the handle.
- Close the handle.

Control channel communication is used, for example, when telling a printer to indent five spaces after each line feed.

8.5.4 The inside of a device driver

Now we will look a little closer at the device driver program. It contains four different parts:

- an 18-byte Device header
- a Strategy routine
- an Interrupt routine
- normally an Initialization routine

Device header: The format of a device header memory image is given in Figure 2 on page 119. The name and attribute words are user-selectable and reflect the function of the device. The attribute word is a bit map that indicates to DOS whether or not the driver has certain special properties. Other entries are determined by driver placement in memory and by the system configuration. The strategy and interrupt fields are offsets into the segment begun by the first byte of the header.

As mentioned earlier, devices are either character type or block type, depending on the way the driver deals with data. Figure 5 on page 124shows definitions of bit fields in the attribute word of a device header.

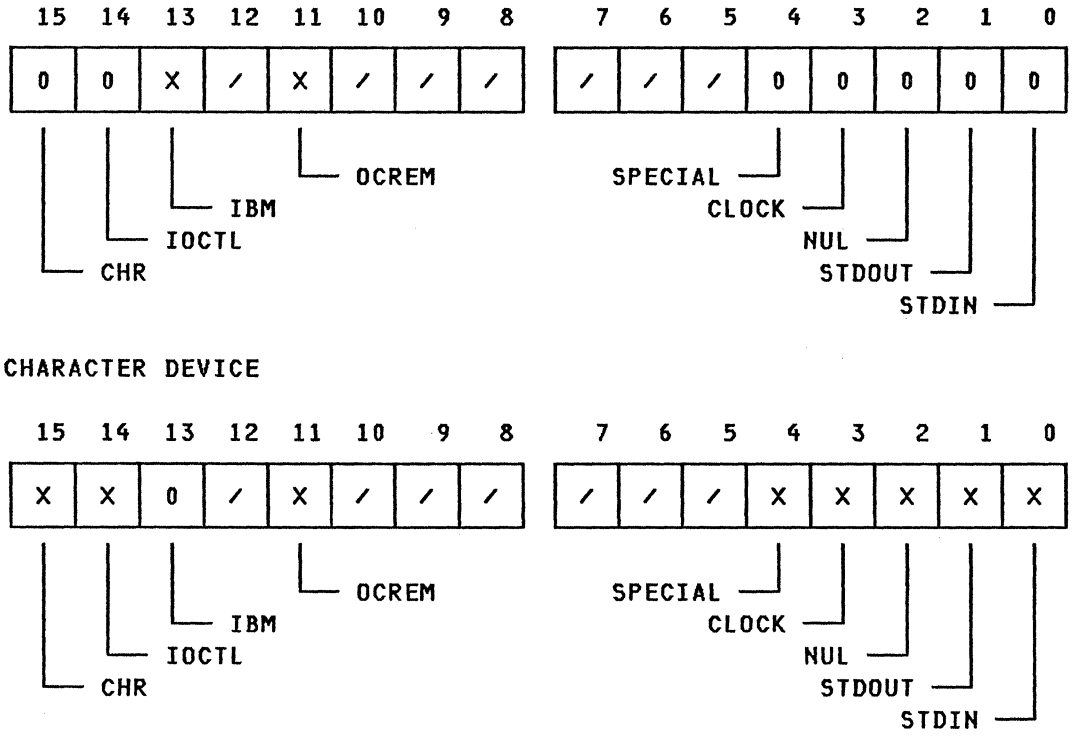


Figure 5. Bit fields of the attribute word

Bits 0-3 of the device's attribute word are flags that indicate to DOS whether the device is one of three devices treated specially by DOS. The NUL device has a unique device driver embedded in IBMDOS and cannot be altered. The NUL bit is a flag that tells DOS that this driver is to be used for the NUL device. For a device that is to be used as the standard I/O device, the pair of bits STDOUT and STDIN are set. Similarly, the CLOCK bit is set on any device used as the clock device.

In the case of character devices, the name field of the device header is an array of eight ASCII characters, making up a legal file name. Instead of names, block devices are given unit numbers. For block devices, the first byte of the name field contains a count of the units supported by the driver, expressed in binary form. The rest of the name field is not used.

Strategy and Interrupt routines: The interface between DOS and devices was designed with multitasking in mind. The intent was to allow each device in a multitasking environment to maintain its own queue of requests for its services. The DOS executive would periodically scan its queue of pending requests for completions and wake up waiting processes. A device request would call the strategy entry point, which would simply queue

up the request in the form of a request header and return. Upon completion, a device would receive an interrupt and its service routine would post results in the corresponding request header. Before returning, the device would check again for waiting requests.

This capability has not been used in DOS versions 2.0, 2.1, or 3.0, none of which handle multitasking. In a theoretical multitasking version of DOS, requests for device services would almost certainly be handled this way. In current DOS releases, the strategy entry point is used to pass a 32-bit pointer to a request block that is not queued. Next, an immediate call is made to the interrupt entry point. Here the request is serviced, and results are passed back in the request block.

The strategy and interrupt entry points do not have an explicit segment in the device header. The segment address is given by the link pointer of the preceding device header. The segment portion of the address for the strategy and interrupt entry points is the same as the segment address of the header.

Initialization routine: When DOS is installing the device driver program, the program is called at the strategy entry point with a function code of 0 (INIT) in the request header. The function 0 is the initialization call. The device driver program should perform any initializing of the device at this point and return information in the request header about the ending address of the loaded code and/or buffer areas.

Since the init call occurs only once during device driver load, the initialization routine is normally stored at the high end of the device driver routine and is discarded by setting the ending address at the first byte in the initialization routine.

If the device is a block device, some more information must be returned:

- The number of units this device driver will support. This number determines the logical names that the devices will have. If we have one diskette drive and one fixed disk, A, B and C are used. If our installed block device driver supports three devices, then these devices will be named D, E and F.

- A pointer to a BIOS parameter block (BPB) pointer array. BPB's contain information about the characteristics of the supported device, e.g sector size, number of sectors. Chapter 2 in the DOS Technical Reference Manual contains more information on the subject.

8.5.5 The Request Header

The initialization call is only one of many function calls from DOS to the device driver program. DOS approaches the device driver twice to handle one request. The device strategy routine receives the first request from DOS and saves a pointer to the Request Header. The strategy routine enqueues the request, complete with parameters for the interrupt routine.

When the strategy routine returns to DOS, DOS immediately places a second request to the interrupt routine which actually performs the request and returns its status to DOS.

The area used for communication between DOS and the two device driver routines is variable in length and headed by a 13-byte request header (Figure 6) followed by data appropriate to the requested function.

Field	Length
Length in bytes of the request header plus any data at the end of the request header	BYTE
Unit code. Has no meaning for character devices.	BYTE
Command code	BYTE
Status word	WORD
Area reserved for DOS	8-BYTE
Data appropriate to the request	Variable

Figure 6. Request Header

8.5.6 Device driver functions

The command code byte in the request header defines the function and there are now sixteen different functions defined in DOS versions 3.00 and 3.10.

Following are descriptions of the different functions available:

INIT (Function 0): The INIT function called once during device driver installation:

- Defines the number of units for DOS.
- Sets up each device.
- Sets the ending address of the driver.
- Sets the address for the pointer to the BIOS parameter block.
- Sets the status word.

The initialization function sets up the device driver within DOS. It is possible that the driver cannot set up the device. If this is the case, it normally aborts without using memory.

Media Check (Function 1): The Media Check function:

- Reads the media descriptor byte.
- Tests the media
- Sets the status word.
- Sets the media return code.

Build BIOS Parameter Block (Function 2): The Build BIOS Parameter Block function.

- Checks the device to find the descriptor byte.
- Finds the matching BIOS parameter block.
- Sets the direct pointer to the BIOS parameter block.

This function causes the device driver to read the boot sector of the media for the media description.

Input or Output (Functions 3, 4, 8, 9, 12): The INPUT or OUTPUT function:

- Reads the sector byte count.
- Performs the requested function.
- Sets the actual count of sectors or bytes.
- Sets the status word.

Nondestructive Input No Wait (Function 5): The Nondestructive Input No Wait function:

- Provides a one-character look-ahead (character devices only).
- Sets the status word.
- Does not alter the input buffer.

Input or Output Status (Functions 6,10): The Input or Output Status function:

- Determines the status of the input or output device.
- Sets the status word.

There is no data associated with this function.

Input or Output Flush (Functions 7,11): The Input or Output Flush function:

- Clears the input or output buffer.
- Sets the status word.

There is no data for this function.

Open or Close (Functions 13,14): The Open or Close function is for setting a device before starting a task.

Removable Media (Function 15): The Removable Media function sets the busy bit (9) of the status word to 1 if the media is non-removable.

8.5.7 Example of IBM loadable device drivers

VDISK.SYS: The VDISK.SYS is a block device driver that comes with DOS. This device driver simulates a disk drive by using a portion of the computer's memory as the storage medium.

- It is very fast.
- It can be installed in several copies.
- On the IBM PC/AT memory above the 1 Megabyte boundary can be used.
- The size of the simulated disk is defined at installation time.
- The source listing is available on the DOS Supplemental Programs diskette. It is a good example of how to write a block device driver.

ANSI.SYS: The ANSI.SYS is a device driver that comes with DOS. It interprets special character sequences (ESCape codes), using them to change the behavior of the keyboard and display.

ANSI.SYS allows you to:

- reassign the keyboard key definitions. You can reassign a key to issue other ASCII values, including character strings.
- control the position of the cursor on the screen.
- set attributes that determine how your display operates.
  - Set Graphics Rendition (SGR) codes allow you to change the foreground and background color on your color display. It also allow characters to be underlined (on IBM Mono Dispaly), blinking, reversed video, or invisible.
  - Set Mode (SM) codes allow you to set screen width and type.
  - Reset Mode (RM) codes are nearly the same as the SM codes (in fact only one code is different)

8.6 OTHER RESIDENT EXTENSIONS

There are also a lot of other types of extensions for DOS that could make the handling of the system easier. Sometimes is not DOS support good enough, and then someone writes a program that intercept the DOS function and replaces it. We will look at a few examples of that.

One thing that is common for this type of program is that they stay resident when loaded. They usually redirect some interrupt vector into their own code, and wait for the event (keyboard interrupt for example).

8.6.1 Keyboard enhancers

This type of programs enhances the way DOS handles the keyboard. The normal way to edit the commandline in DOS is quite limited. We may want to be able to recall previous commands from some kind of buffer and we would like it to stay on the command line so we can change it without parts of it disappearing when we move the cursor.

There is also possible to have the program that wait for some key combination and emits a string of characters to DOS when it recognizes those keys.

8.6.2 Popup utilities

This kind of utilities, when loaded, normally waits for a some key combination that will wake it up. When the popup-keys are found The program saves part of the screenimage and displays a window for communication with the user.

These program is not multitasking in the normal sense. They are coresiding with the normal DOS application running, but it is the user who decides (via the popup/popdown key) which one is running.

A common type of function for these program is calendar, telephone directory etc.

8.7 CONCLUSION

This concludes our module on DOS extensions. At this point you should understand the following:

- That DOS does not contain all the functions we need. It is just a base for further improvements and additions.
- What a multitasking extension gives us, and how it is implemented on top of DOS.
- Why DOS does not support any thinkable devices and how we can help out by constructing our own device driver.
- That there is a lot of different ways to enhance functions that are already present in DOS, but could be made easier to use or more efficient.

For further studies on the subject the DOS Technical Reference Manual and the TopView Toolkit Manual are warmly recommended.

9.1 OBJECTIVES

The power of the Intel 80286 microprocessor used in the IBM PC/AT extends far beyond the ability to run PC DOS programs at a faster rate than the PC. The 80286 processor can run in two different modes -- real and protected. Real mode is compatible with the 8088 used in the rest of the PC models and is also the mode PC DOS uses on the IBM PC/AT.

Because of the complexity and sophistication of the 80286, you will find this section more difficult than most other sections in the PC Internals Fundamentals course. It will only begin to make complete sense after two or possibly more readings; at the outset, you will find yourself grasping only individual details. In attempting to teach the 80286 architecture one encounters a classic chicken-and-egg problem: there is no logical place to begin, because in order to understand each component of the architecture you must know the other components too. Hopefully this module will at least acquaint you with some of the important features of the 80286 over its predecessors.

In this module we will look at the processor when it is running in protected mode, and at the end you should understand the following:

- The differences between real and protected mode
- The concept of global and local virtual address spaces
- The protection types available and how they function
- How task switching and state transitions are handled
- The concept of privileged Input/Output operations
- How Interrupts and exceptions are handled

9.2 AGENDA

In order to accomplish these objectives, we will proceed as follows:

1. The concept of segmented addressing and the registers of the 80286 in real mode will be compared with the new way in which they are used and expanded in protected mode.
2. The virtual addressing scheme will be explained and we will look into the use of global versus local address spaces.

3. The protection built into the addressing scheme will be expanded with the concept of privilege levels.

4. The different methods of task switching and transfer of control will be explained, as will the concept of using gate descriptors.

5. The way in which interrupts and exceptions are handled in real address mode will be compared with how they are handled and expanded in protected mode.

6. The protection of memory-mapped and port-based Input/Output operations will be discussed.

9.3 INTRODUCTION TO THE 80286 PROCESSOR

The 80286 processor has many features in common with the 8088 and 8086 processors. Examples are byte-addressed memory, I/O interface hardware, interrupt vectoring and support for multiple processors and processor extensions. The addressing modes and the basic instruction set are also the same.

The 80286 processor can function in two modes of operation. In one of these modes (Real addressing mode) only the base architecture is available to programs, whereas in the other mode (Protected addressing mode) a number of very powerful advanced features have been added, including support for virtual memory, multitasking and a sophisticated protection mechanism.

9.3.1 Memory management

The memory architecture of the 80286 in Protected mode has been significantly enhanced and expanded. The physical address space has been increased from 1 megabyte to 16 megabytes, while the virtual address space (i.e., the address space visible to a program) has been increased from 1 megabyte to 1 gigabyte (1 gigabyte=2¬30, or 1,024 megabytes). Moreover, separate virtual address spaces may be provided for each task in a multitasking system.

The 80286 supports a segmented memory architecture. It also fully integrates memory segmentation into a comprehensive protection scheme. This protection scheme includes hardware-enforced length- and type-checking to protect segments from inadvertent misuse.

Mechanisms are included in the 80286 to allow an efficient implementation of a virtual memory system. All instructions that could refer to a virtual segment that is not in real memory at the time the instruction is executed are fully recoverable. Such a situation would cause what is called a Segment-Not-Present fault, and would invoke an operating-system

routine to take care of the interrupt by loading the requested segment into real storage from an external device, then by restarting the faulted instruction.

9.3.2 Task management

The 80286 is designed to support multitasking systems. The architecture provides direct support for the concept of a task. For example, a special segment type, the Task State Segment, is a hardware-recognized and hardware-manipulated structure which contains information on the current state of all tasks in the system.

Very efficient task switching can be invoked with a single instruction. Separate logical address spaces may be provided for each task in the system. Finally, mechanisms exist to support inter-task communication, synchronization, memory sharing and task scheduling.

9.3.3 Protection mechanisms

The 80286 protection mechanisms are based on the notion of a "hierarchy of trust". Four privilege levels are distinguished, ranging from Level 0 (the most trusted) to Level 3 (the least trusted). Level 0 is usually reserved for the operating system kernel. The four levels may be visualized as concentric rings, with the most privileged level in the center.

At any one time, a task executes at one of the four levels. Moreover, all data segments and code segments are also assigned to privilege levels. A task executing at one level cannot access data at a more privileged level, nor can it call a procedure at a less privileged level (i.e. trust a less privileged procedure to do work for it). Thus, both access to data and transfer of control are restricted in appropriate ways.

The 80286 also make a complete separation of logical address spaces for different tasks possible. This part of the protection scheme is a natural part of the 80286 memory management architecture.

Privilege levels and separation of logical address spaces enables us to organize software systems so that each task is protected from damage by other tasks and so that privileged procedures are protected from lower-level procedures.

The processor interprets the protection parameters and automatically performs all the checking necessary to implement protection.

9.4 COMPARING THE REGISTERS IN REAL AND PROTECTED MODE

The register structure of the 80286 in protected mode looks very much the same as in real address mode. All the general registers (AX, BX, CX, DX, BP, SP, SI and DI) as shown in Figure 7 on page 134 are exactly the same.

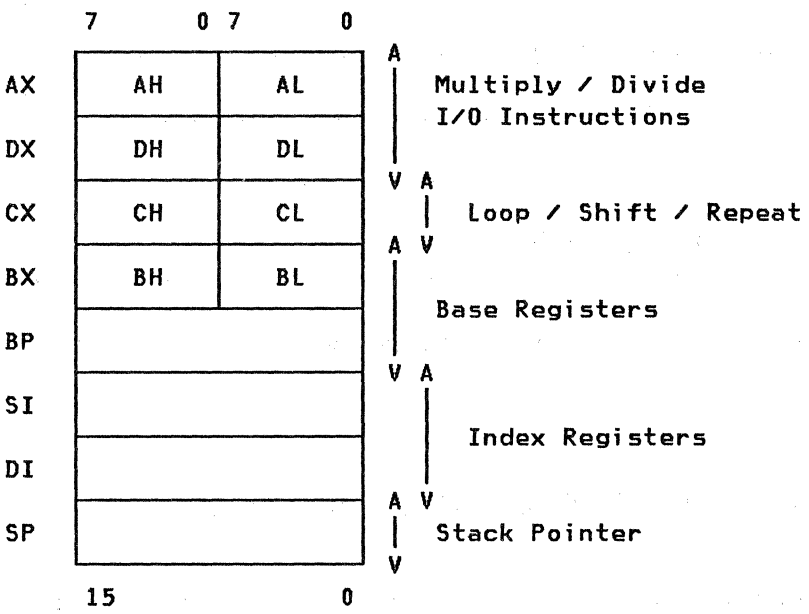


Figure 7. 80286 General Registers.

The segment registers (CS, DS, ES and SS) are still present, but they are called segment selectors because they no longer contain the base address. They are now used as index pointers into a memory table, where the real base address information is stored.

As you can see in Figure 8 on page 135 the segment selectors contain two parts. The 16-bit part we can reach from programs; in addition there is a 48-bit program invisible part. The first part contains the selector, and the second part contains the physical address information that the processor has picked up from the memory table.

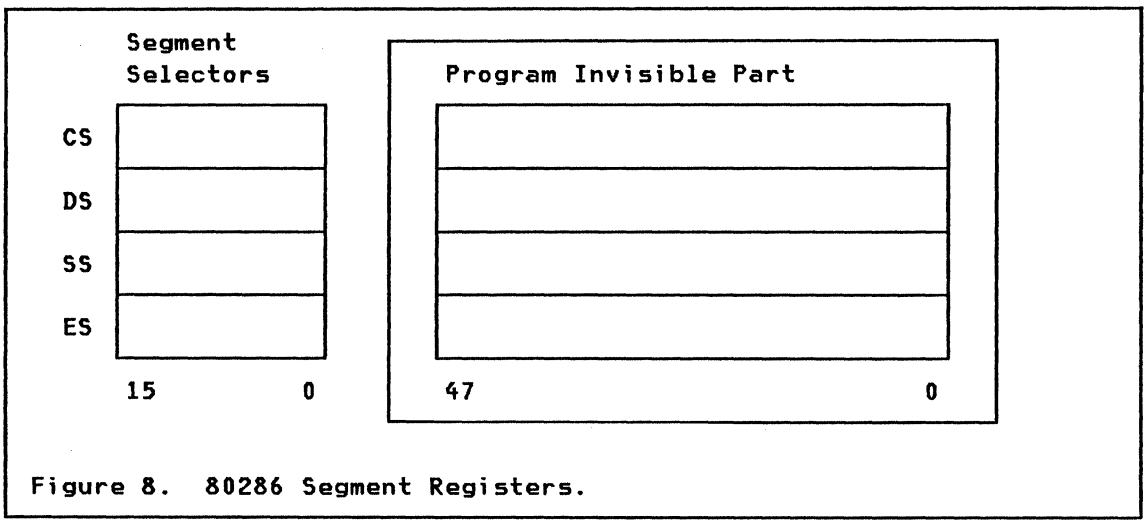


Figure 8. 80286 Segment Registers.

There are some new registers in the 80286, but we will return to them later.

9.5 DIFFERENCES IN THE MEMORY ADDRESSING SCHEME

Let us refresh our memory on how we used the segment register in real address mode to point directly into real memory, as shown in Figure 9 on page 136. The processor added four binary zeroes to the segment value, and that gave us the possibility to start the segment on any 16-byte boundary within the 1-megabyte physical address space.

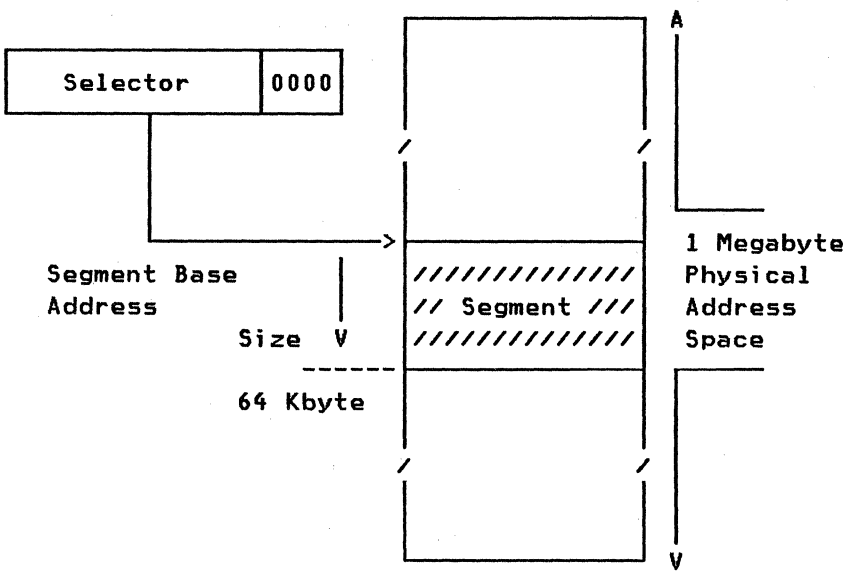


Figure 9. Segment Selector Interpretation (Real Address Mode)

Figure 10 on page 137 shows how the 80286 in protected mode instead uses 14 bits of the selector to indicate which segment we want to access. The 14 bits enables us to point to 16K different virtual segments, which could be from 1 byte to 64K in size. Note that these 14 bits are not an address: they are a segment number. By allowing 16K different segments of 64K maximum size each, we are able to address a total of 1 Gigabyte of virtual storage, and this is also the view an application programmer should use. How these segments are mapped into the 16 Mbytes of real storage should normally be of no concern for the application programmer, but we will look at this more closely in a moment.

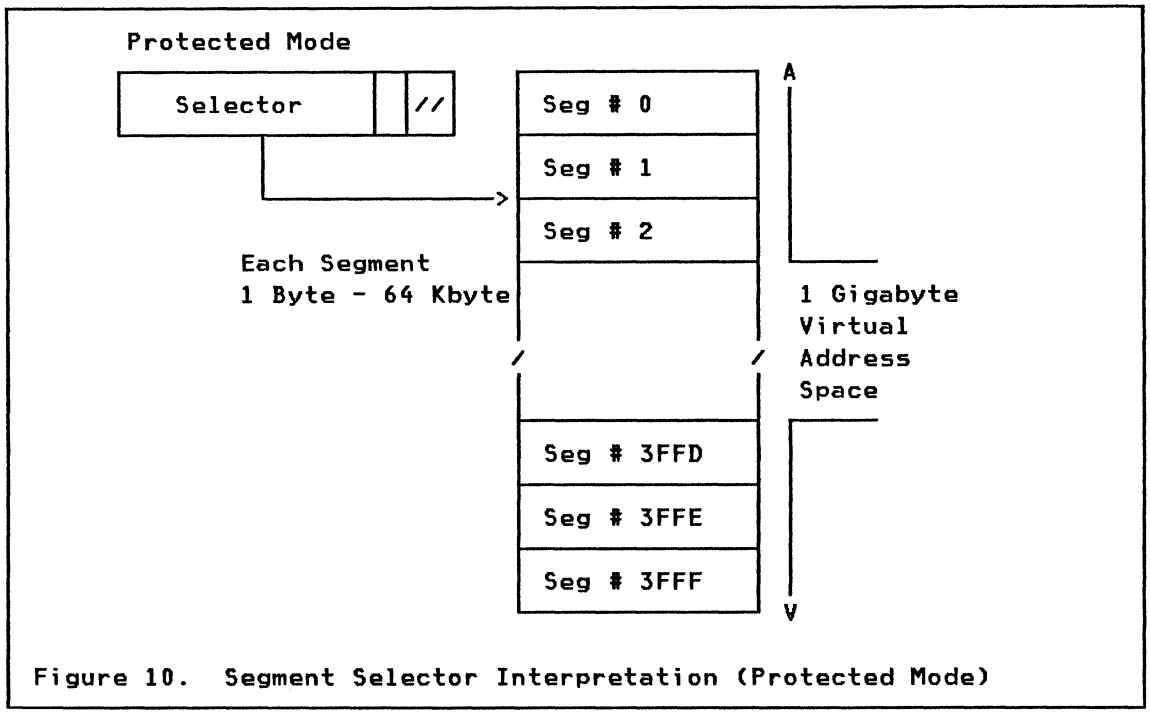


Figure 10. Segment Selector Interpretation (Protected Mode)

If we look once more at the segment selector in Figure 11 on page 138, we see that in real address mode the 16+4 bits give us the physical segment base address.

In protected mode we divide the selector into three parts. The high-order 13 bits are used as an index into one of two active memory tables. These tables are called the Global Descriptor Table (GDT) and the Local Descriptor Table (LDT). The table entries are called Descriptors and contain the information the processor needs to map the segment into real storage. Each table may contain up to 8K descriptor entries and the TI (Table Indicator) flag tells the processor which one to use.

The low-order two bits are called the Requested Privilege Level (RPL) and we will discuss privilege levels in more detail later.

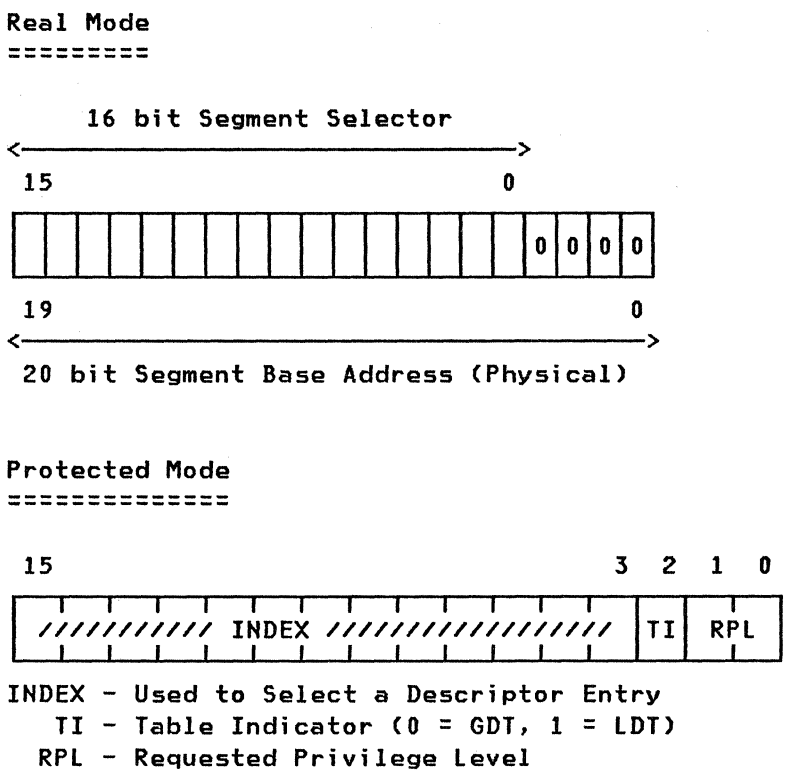


Figure 11. Segment Selectors (Real and Protected mode)

When we addressed real storage in real address mode we needed a pointer that consisted of two parts -- the segment register value plus an offset.

When we want to address real storage in protected mode the pointer also consists of two parts -- the selector and an offset. You can see this pointer at the lower left corner of Figure 12 on page 139. In fact, to the programmer, there is no difference. The selector points to a descriptor entry in a descriptor table. The descriptor entry contains the physical base address of the segment, which is added to the offset by the physical address adder to give us the real 24 bit storage address.

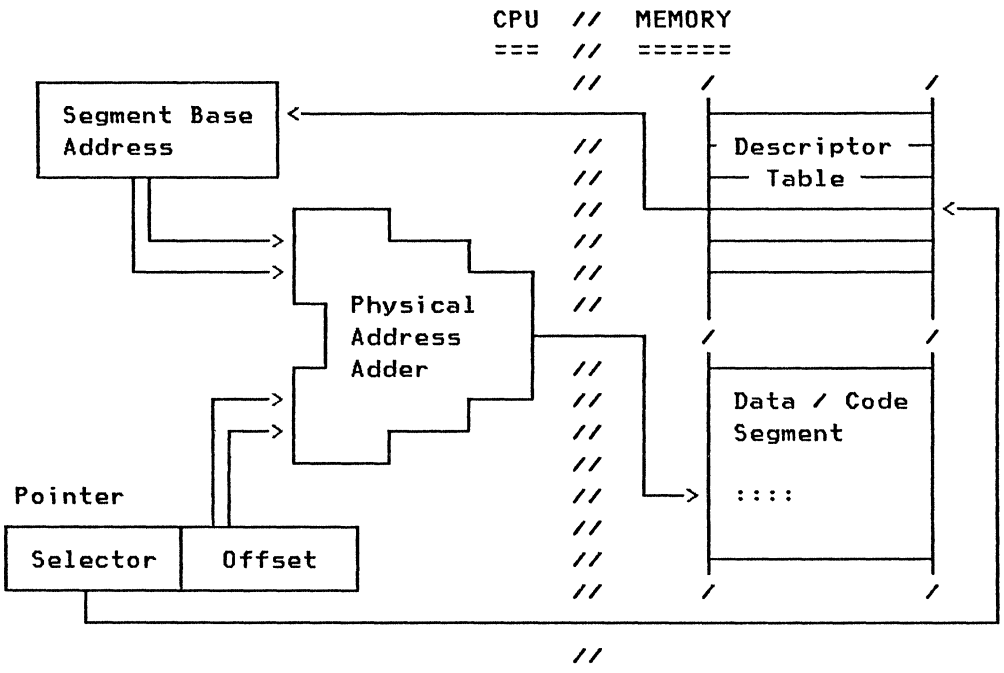


Figure 12. Protected Mode Addressing

What are descriptors and what do they contain?

Each descriptor in the Global or Local descriptor table contains 8 bytes. Figure 13 shows us the layout.

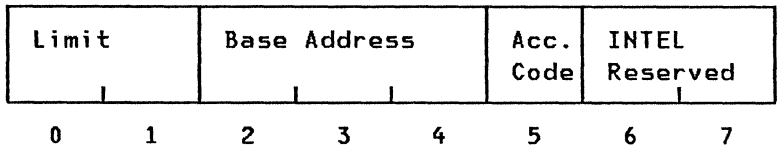


Figure 13. Descriptors.

The first two bytes contain a size limit value which points to the last byte of the segment. When we try to address storage through this descriptor, the processor will always check to see that our offset value is within the limit. Any attempt to access memory outside our segment will cause an exception, which hopefully some operating system routine will handle.

The next 3 bytes (24 bits) contains the physical base address of the segment when it is mapped into real storage. This is the only place where we can find a genuine real-storage address. This enables the processor to start a segment at any physical address within 16 megabytes, not just on a paragraph boundary.

The fifth byte is the Access Code. This byte contains fields that identify the type of descriptor and some other fields controlling the access available to this segment.

There are four types or classes of descriptors. They are:

1. DATA Segment Descriptors (these refer to segments containing system or application data, including stacks).

2. EXECUTABLE Segment Descriptors (refer to segments containing executable instructions).

3. SYSTEM Segment Descriptors (refer to segments containing special hardware-recognized data structures).

4. GATE Descriptors (define entry points for control transfers).

When we addressed storage in Figure 12 on page 139, we described it as if we fetched the segment base address from the descriptor table at that time. This is not exactly true. It would be too much work for the processor.

If we look back at our segment selectors again, we will find that the processor will not have to reference the descriptors every time it is calculating a physical address. The program invisible part of the selector has the space needed to keep the current information for every selector, as you can see in Figure 14.

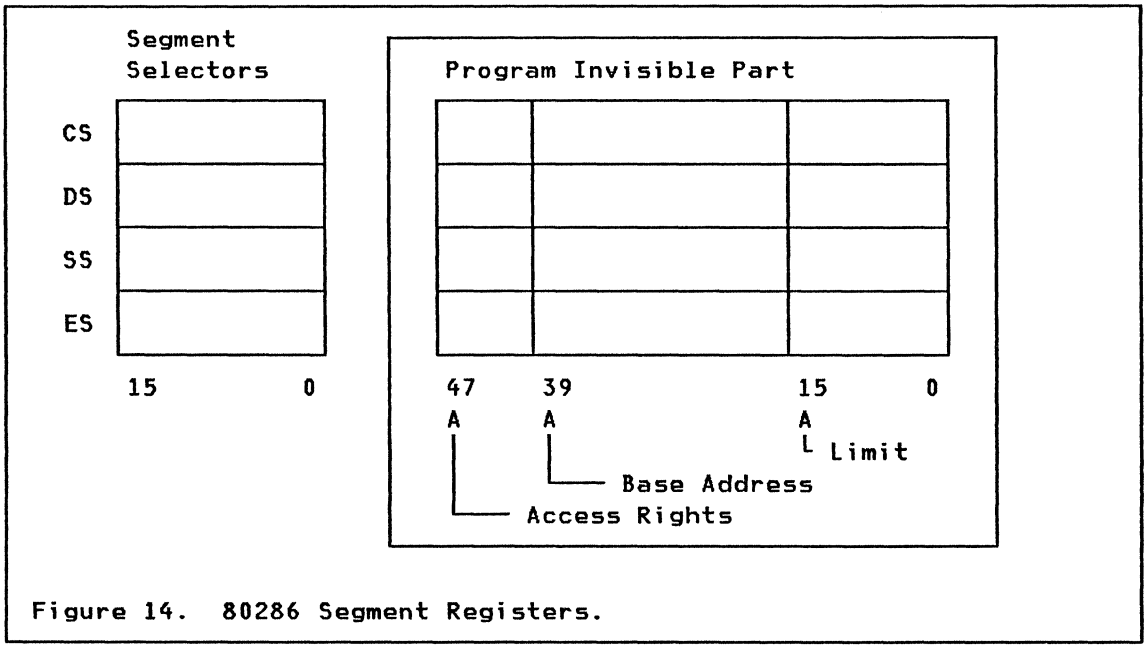


Figure 14. 80286 Segment Registers.

How do we get that information into these invisible selector extensions?

Figure 15 shows one of the descriptor tables, on the right of the figure.

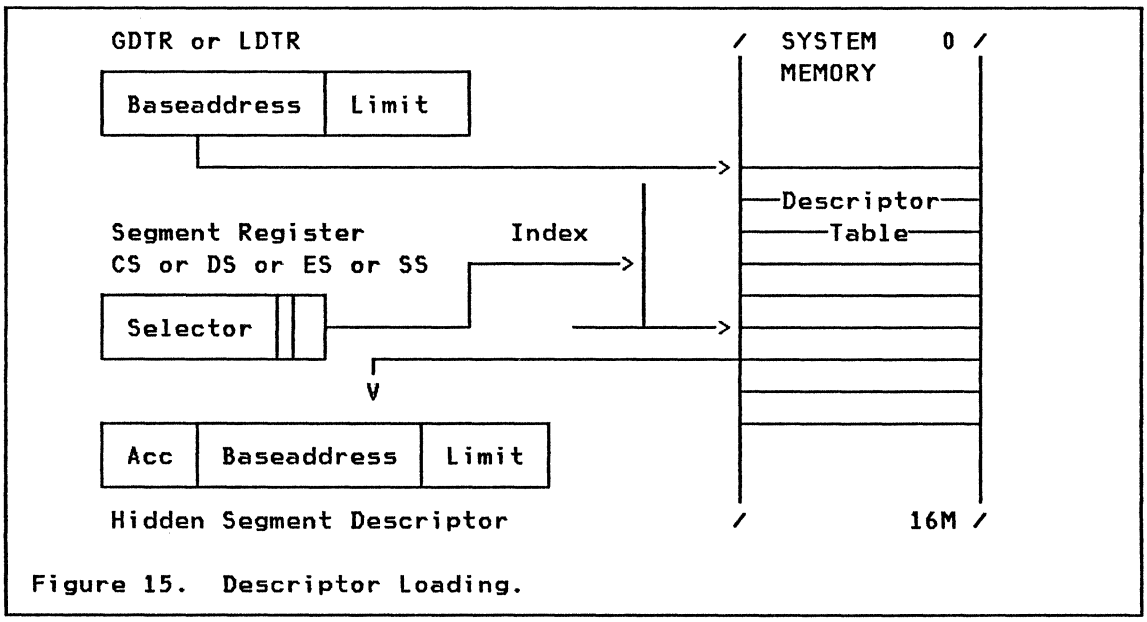


Figure 15. Descriptor Loading.

This could be either the global or the active local descriptor table. There are two new registers that point to these tables, the GDT register and the LDT register. Don't worry about these for now.

When the program loads a new value into one of the four segment selectors CS, DS, ES or SS, the processor will then use the Table Indicator bit to point to the right descriptor table. The Index value in the selector points to one of the descriptors, and the addressing information is moved from real storage into the hidden part of the segment selector within the processor.

From now on, any addressing that uses that specific selector as a segment pointer can be resolved within the processor.

Let's now look at some new hardware registers needed to enable these addressing schemes.

There are four new system registers used in protected mode. These are shown in Figure 16.

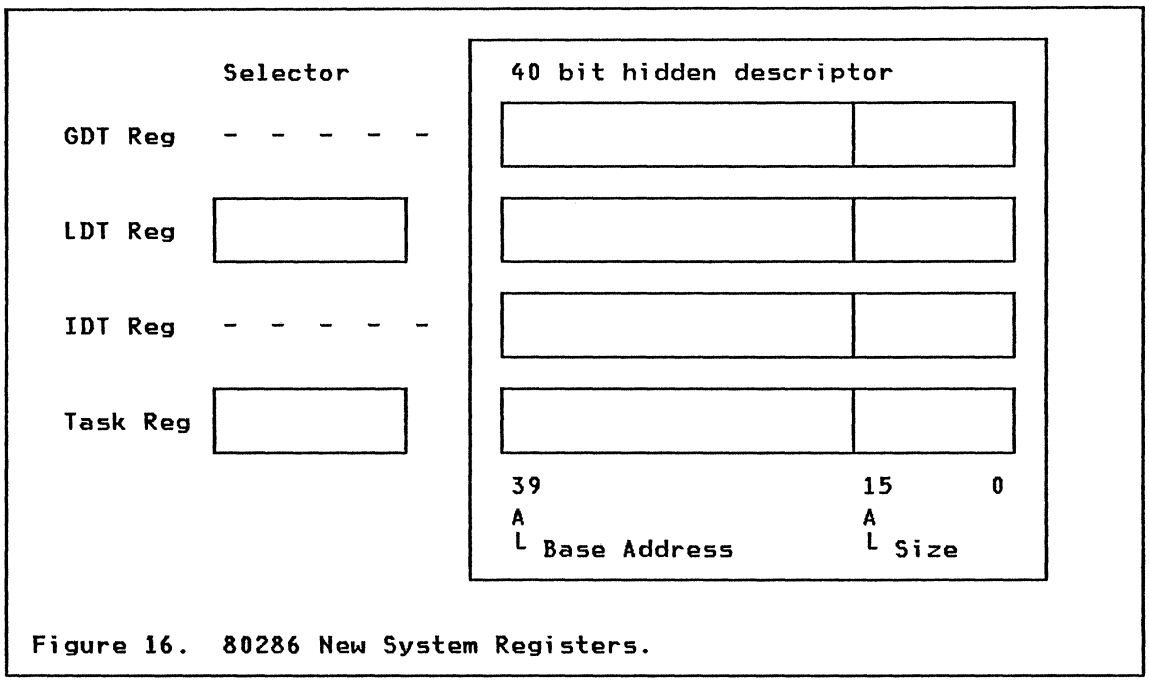


Figure 16. 80286 New System Registers.

They all contain a hidden 40-bit part and two of them also contain a visible selector part.

GDT Reg The Global Descriptor Table Register points to the table that provides a complete description of the global address space. It is normally static and loaded during initialization. Two dedicated instructions (only available at the most privileged level in the system) are used for loading and storing this 40-bit value.

LDT Reg The Local Descriptor Table Register is a dedicated 40-bit register that contains, at any given moment, the base and size of the local descriptor table (LDT) associated with the currently executing task. Unlike GDTR, the LDTR register contains both a "visible" and a "hidden" component. Only the visible component is accessible, while the hidden component remains truly inaccessible even to dedicated instructions.

The dedicated, protected instructions LLDT and SLDT are reserved for loading and storing the selector part of the LDT register. The 40-bit hidden part is automatically loaded from the descriptor entry in the global descriptor table, pointed to by the selector.

IDT Reg The Interrupt Descriptor Table Register points to the table that defines interrupt handlers for up to 256 different interrupts. It is normally static and loaded during initialization. Two dedicated instructions (only available at the most privileged level) are used for loading and storing this 40-bit value.

Task Reg The Task Register is a dedicated 40-bit register that contains the base and size of the active Task State Segment (TSS). The Task register contains both a "visible" and a "hidden" component. Only the visible component is accessible, while the hidden component remains truly inaccessible even to dedicated instructions.

The dedicated, protected instructions LTR and STR are reserved for loading and storing the selector part of the Task register. The 40-bit hidden part is automatically loaded from the descriptor entry in the global descriptor table, pointed to by the selector.

There is also two new flags used in the flags register. If you look at Figure 17 you will find all the familiar flags from real mode, but also two new ones.

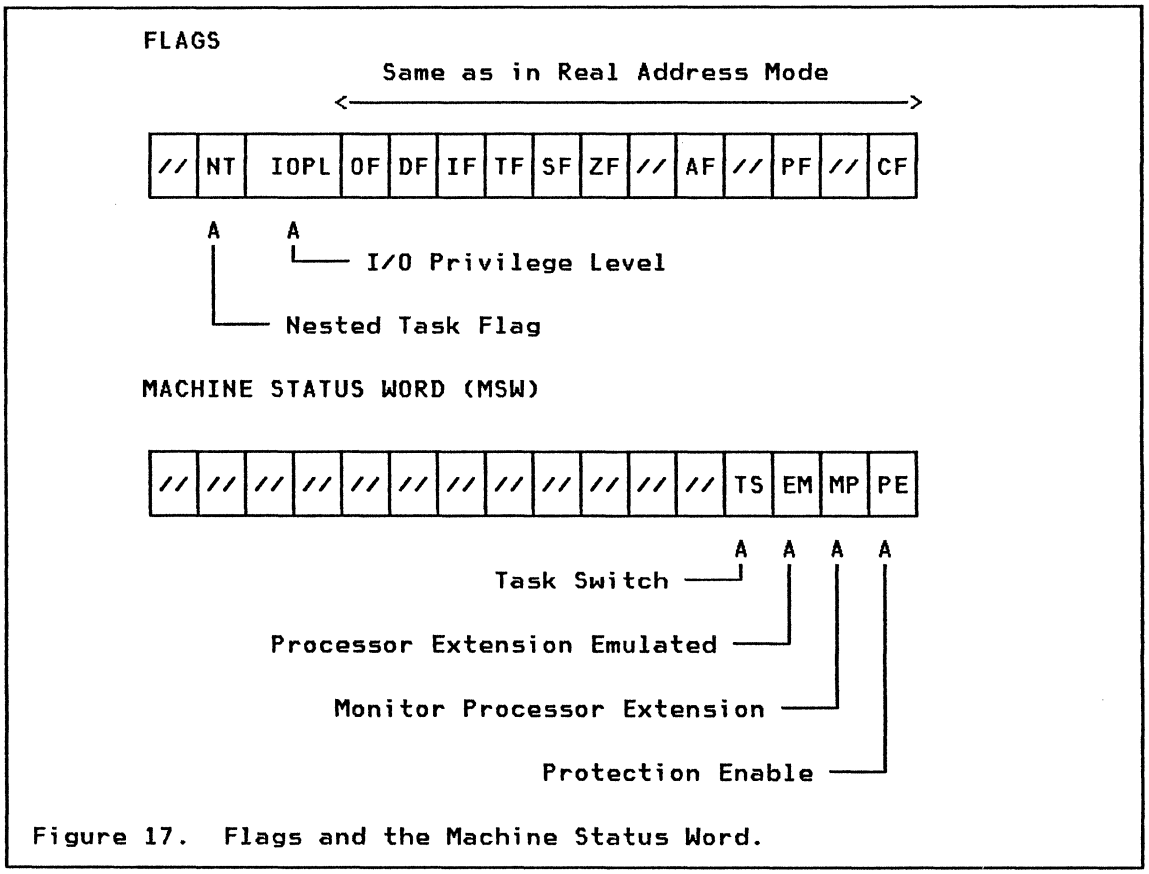


Figure 17. Flags and the Machine Status Word.

- NT -- Nested Task Flag indicates that one task has called another task. This helps the processor to find the way back when the called task ends.

- IOPL -- I/O Privilege Level indicates the highest priority that is needed to perform the privileged I/O instructions.

The last new register is the MSW (Machine Status Word).

- TS -- Task Switched indicates the next instruction using a processor extension will cause an exception 7, allowing software to test whether the current processor extension context belongs to the current task.

- EM -- Emulate processor extension causes a processor extension not present exception on ESC instructions to allow emulating a processor extension.

- MP -- Monitor processor extension allows WAIT instructions to cause a processor extension not present exception.

- PE -- Protected mode Enable places the 80286 into protected mode and can not be cleared except by RESET.

So you can see that the MSW is used to get the processor from real to protected mode. It is also used to indicate if you have processor extension of type 80287 (numerical processor) present or if programs should take over and emulate the extension processors function.

Before looking a little closer at the segment descriptors, we need to understand the concept of privilege levels.

The 80286 processor protection scheme supports four privilege levels where Level 0 is the most trusted level and Level 3 is the least trusted. The four levels may be visualized as concentric rings, with the most privileged level in the center. Figure 18 is a visualization of the concept of the four levels.

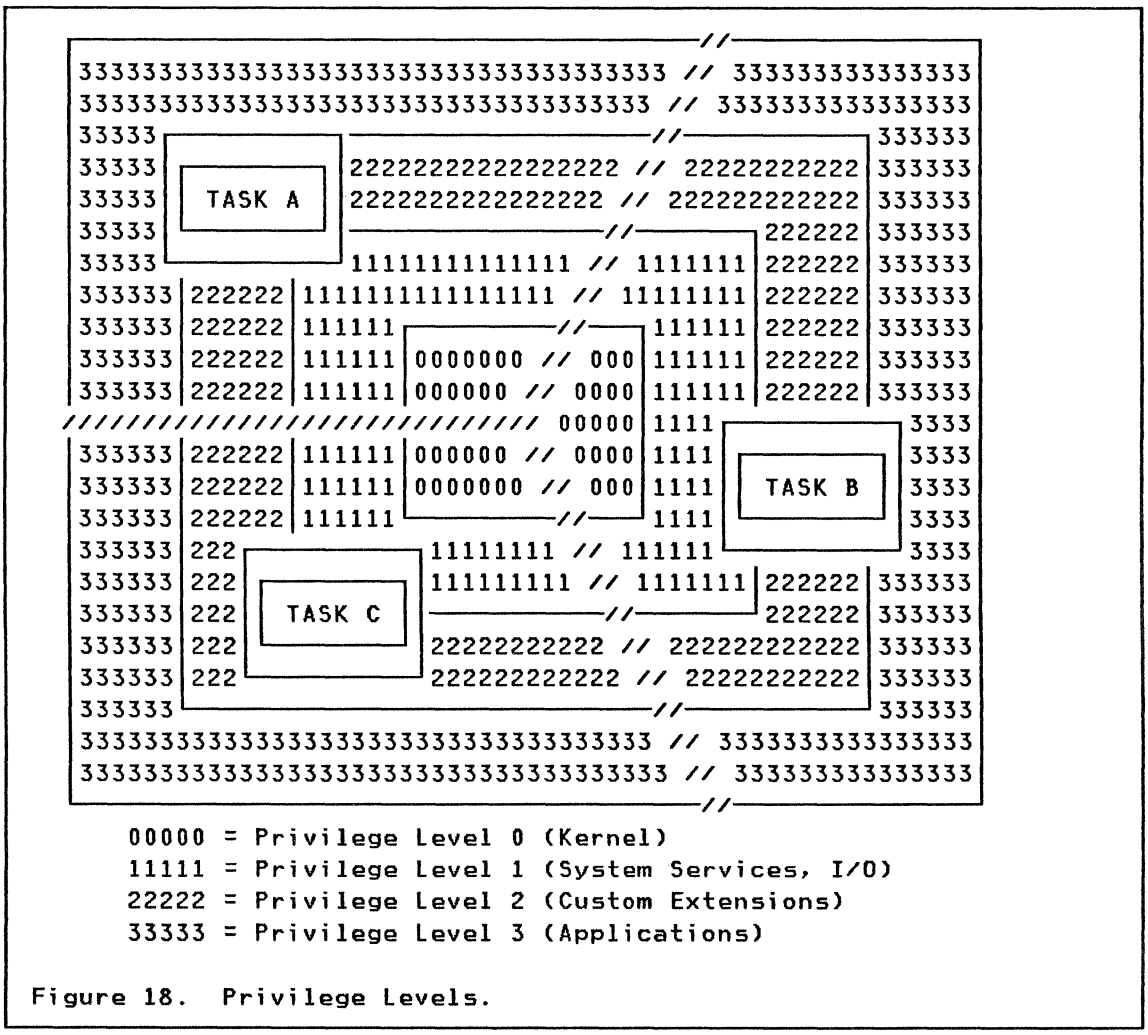


Figure 18. Privilege Levels.

The levels and their recommended use are:

Level 0 This is the most trusted level, and code executing on this level can use all the 80286 instructions. This level is used by the routines in the operating system that are essential for resource allocation and control. This part of the system is often referred to as the kernel in some operating systems and the nucleus in others.

Level 1 The second most trusted level is normally used for the rest of the operating system routines and for the Input/Output support routines.

Level 2 This level could be used as the application support level. It can be used for routines which do not belong to the operating system, but which should still be protected from applications. Examples of these are communications and database management routines.

Level 3 This is the least trusted level and this should be the level used for normal applications.

Let's now return to the segment descriptors and take a closer look at them.

A reference from one segment (e.g. a code segment) to another (e.g. a data segment) is realized indirectly through a **descriptor**, which contains information about the referenced segment.

All descriptors reside in a descriptor table. Every segment must have at least one descriptor, otherwise there is no way of addressing the segment. Referring to Figure 19 we see that each descriptor has a size of eight bytes. Six of these are used by the 80286 and the other two are reserved for use by the next generation of processor from Intel -- the 80386.

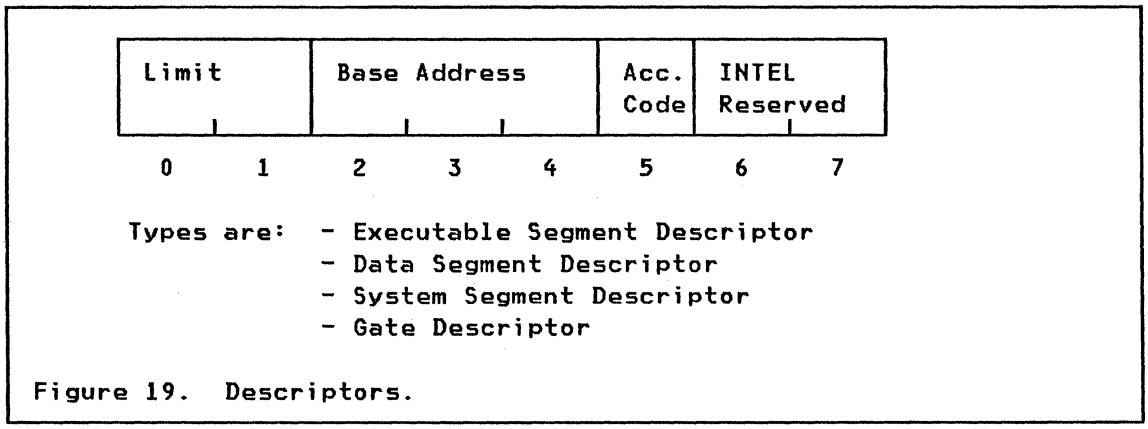


Figure 19. Descriptors.

The main common fields are:

Segment Limit The value of this field is one less than the length of the segment (in bytes) relative to the beginning of the segment. The 16 bits of this field make it possible to have segments up to 64K bytes long. The hardware automatically checks all addressing operations to ensure that they do not exceed the segment limit of the segment to which they refer.

Segment Base This field contains the **physical address** of the beginning of the memory segment referred to by this descriptor. The 24 bits of this address give the 80286 a 16-megabyte range of real

addresses. This is the only place in which physical addresses are used. All other addresses are relative to the physical addresses stored in descriptors, making it possible to relocate executable and data segments without making any changes to the relocated segments or to the code that refers to the segments. The only necessary change to relocate a segment is to change the physical address stored in the descriptor.

Access code This byte first defines the descriptor to be one of the four different types. Depending on the type field, the remaining fields in the byte have different meanings and purposes. The fields will be explained when we look at each different descriptor type below.

INTEL reserved This portion (16 bits) of the descriptor is reserved for use by the next generation processor -- the 80386. It should always be initialized with zeros.

Now it is time to look more closely at the information in the **access code** field of the descriptor. We will start off with the executable segment descriptor illustrated in Figure 20.

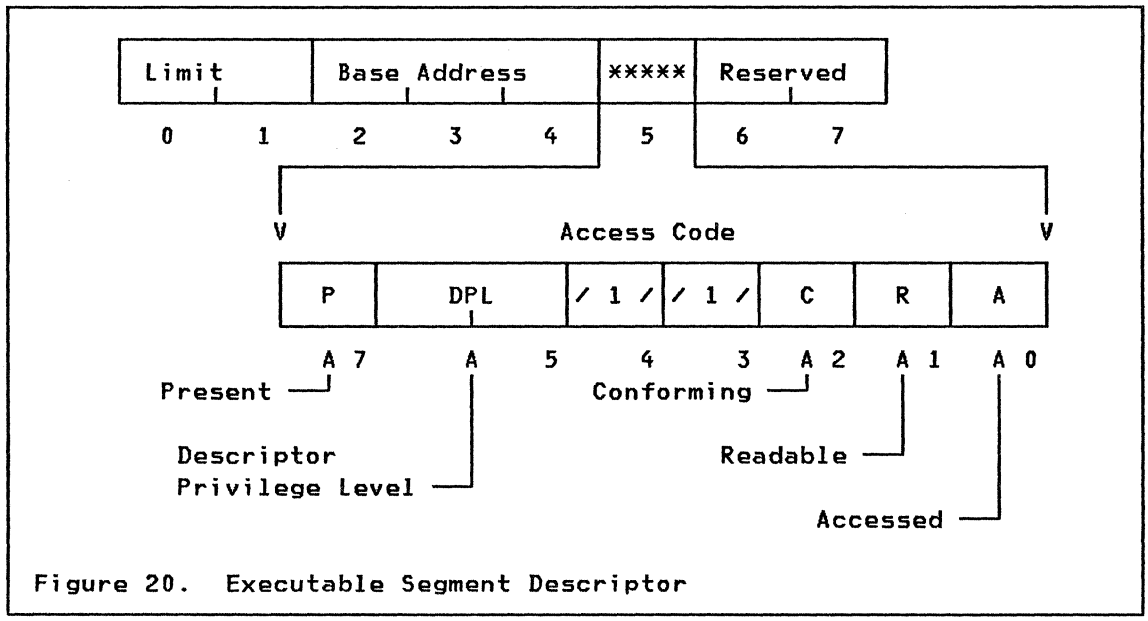


Figure 20. Executable Segment Descriptor

The two ones in bits 3 and 4 designates this descriptor as an **executable segment descriptor.**

we will now look at the rest of the fields one at a time.

Present bit This bit is set on and off by the operating system and it is checked by the processor hardware when we try to load the selector.

- If the bit is **on** the segment addressed by this descriptor is actually present in memory.
- If the bit is **off** any reference via this descriptor will cause a fault and provide the operating system with the opportunity to load the segment from external storage.

Descriptor Privilege Level The value of this item defines the privilege level of the segment addressed by the descriptor. This value could be 0, 1, 2 or 3. For the executable segment this will normally be the privilege level of the actual running code.

Conforming This field applies to executable segment descriptors only. Ordinarily (when the bit is zero) a called procedure executes at the privilege level defined by its own DPL. When the conforming bit is set however, the called procedure executes at the calling procedures privilege level (which never could be lower than the called procedure's own DPL).

Readable This field applies only to executable segment descriptors. When this bit is **off**, any attempt to read the code within this segment will case a fault. The code is executable only. When the bit is **on**, the code is executable and readable.

Accessed The processor sets this bit when the descriptor is accessed (ie. loaded into a segment register). Operating systems which implement virtual memory may use this bit to indicate whether this segment should be written to secondary storage before the RAM space it occupies is reused. This should not be necessary for a executable segment since in cannot be written to, but it would be used for a data segment (there is no way to know whether a segment has been written to or has only been read).

What then is different for a **data segment descriptor**? As you can see in Figure 21 the data segment descriptor is designated by bit 3 being off and bit 4 being on. Bit 3 and 4 are checked by the processor hardware when we try to refer to a segment. Trying to load the CS selector with a data segment descriptor or DS, ES and SS with code segment descriptors will always generate a fault interrupt.

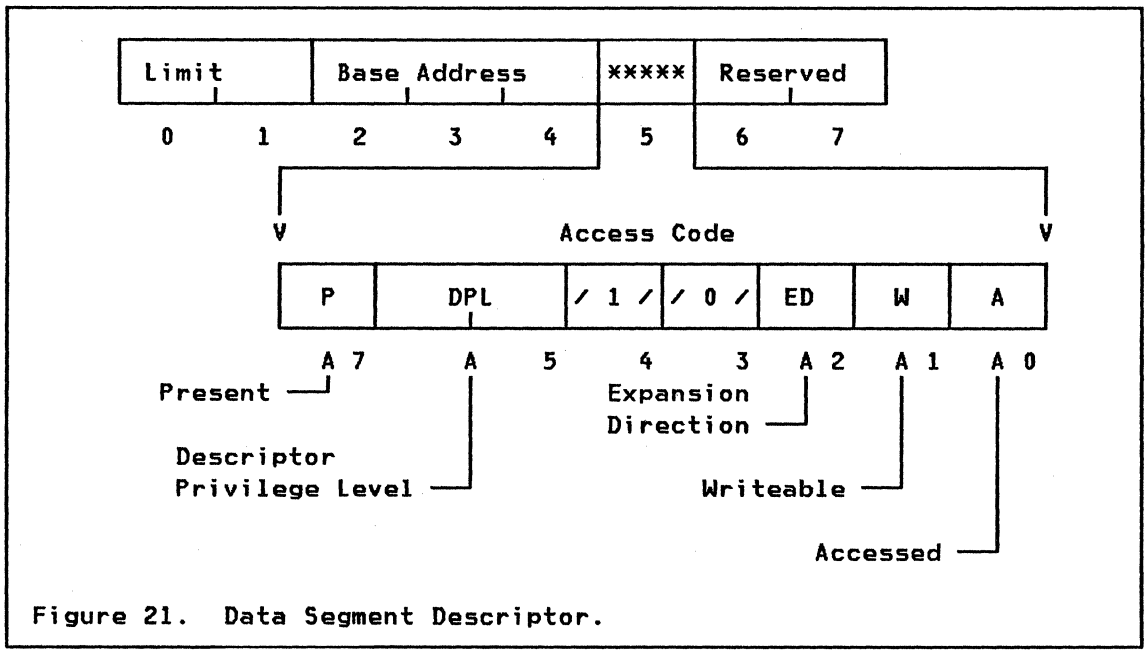


Figure 21. Data Segment Descriptor.

Let's look at the other fields in the data descriptor access code:

Present bit Same as in the executable segment descriptor.

Descriptor Privilege Level This defines the privilege level of the data segment addressed by the descriptor. The data segment could only be read from or written to by processes executing on the same or a more privileged level.

Expansion Direction Data segments may contain stacks as well as other data structures. Stacks expand toward lower addresses while most other data structures expand toward greater (higher) addresses. This field indicates the growth pattern for the segment.

If the bit is off, it indicates growth from the base address upwards. The offset value must be less than the descriptor limit value

If the bit is on, it indicates growth from offset FFFFh downwards. The offset value must be equal or higher than the descriptor limit value.

Writable This field applies to data segment descriptors. A value of one permits writes into the segment and a value of zero inhibits writes into the segment (read only).

Accessed Same as in the executable segment descriptor.

The third type of segment descriptors is the **system segment descriptor**. Bits 3 and 4 are both off and bit 2 is off. Bit 2 enables the 80286 processor to differentiate this descriptor from the fourth type -- the **gate descriptor** which we will look at later.

There are two types of system segments defined by this descriptor. As you can see by Figure 22, they are specified by the type field in bits 0 and 1. The first type of system segment is the **Local Descriptor Table segment (LDT)** and the second type is the **Task State Segment (TSS)** which could be marked either as available or busy. We will talk more about the TSS a little later.

The **Present** bit and the **Descriptor Privilege Level** bits are used in the same way as in the data segment descriptor above.

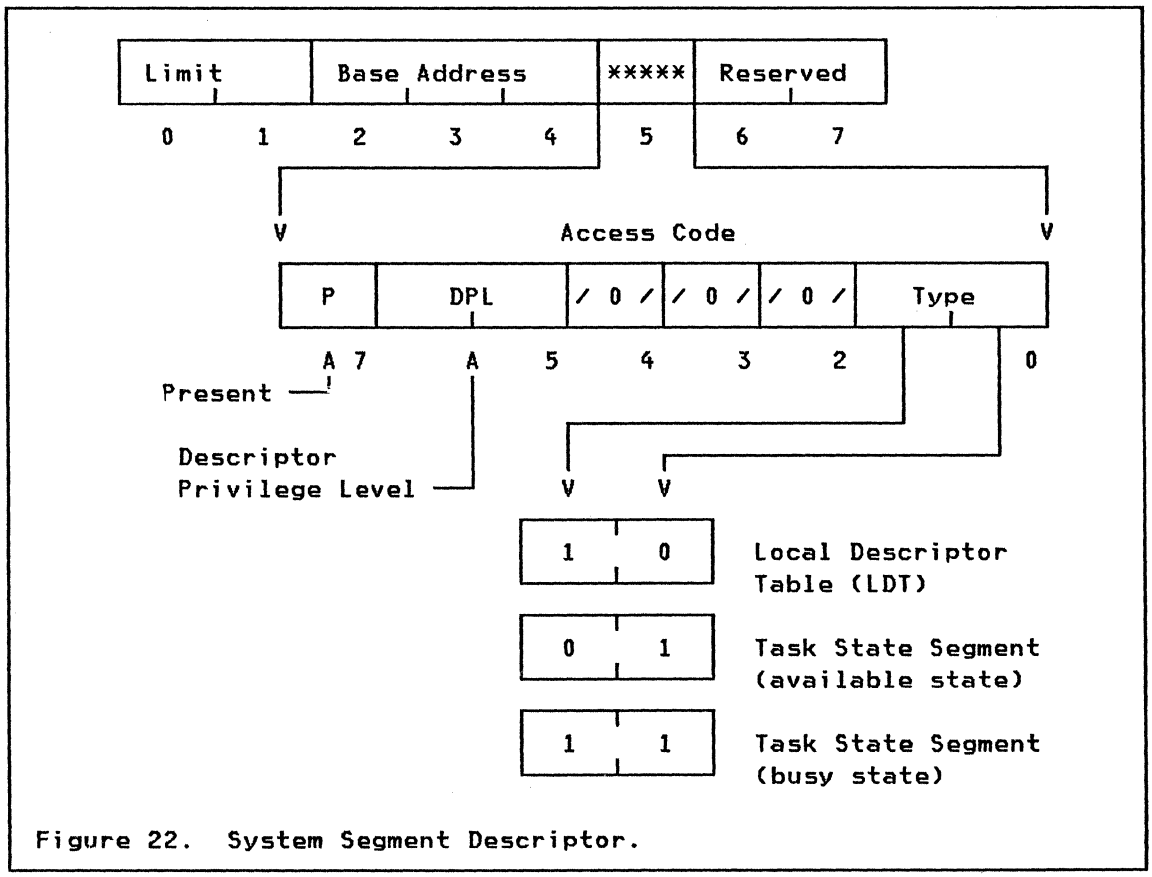


Figure 22. System Segment Descriptor.

9.6 CONTROL TRANSFER MECHANISMS

We can divide the control transfer mechanism of the 80286 into two categories.

- Transfer of control within a task

- Transfer of control between tasks (multitasking)

Transfer of control **within** a task can be of three kinds:

1. Within a segment, causing no change of privilege level (short jump, call, or return)

2. Between segments at the same privilege level (long jump, call, or return)

3. Between segments at different privilege levels (far call or return)

The two first types of control transfers need no special control with respect to privilege protection. The third type is an inter-level transfer and requires special considerations to maintain system integrity. The 80286 hardware must check that:

- The task is currently allowed to access the destination address.

- The correct entry address is used.

To achieve control transfers, a special descriptor type called a gate is provided to mediate the change in privilege level. Control transfer instructions call the gate rather than transfer directly to a code segment. From the viewpoint of the program, a control transfer to a gate is the same as to another code segment.

Gates allow programs to use other programs at more privileged levels in the same manner as a program at the same privilege level. Programmers need never distinguish between programs or subroutines that are more privileged than the current program and those that are not.

Let's look at that last type of segment descriptor now.

A gate is an 8-byte descriptor used to redirect a control transfer to a different code segment in the same or a more privileged level, or (as we will discuss later) to a different task. There are four types of gates: **call, trap, interrupt and task gates**. Figure 23 on page 156 shows the format of the gate descriptor.

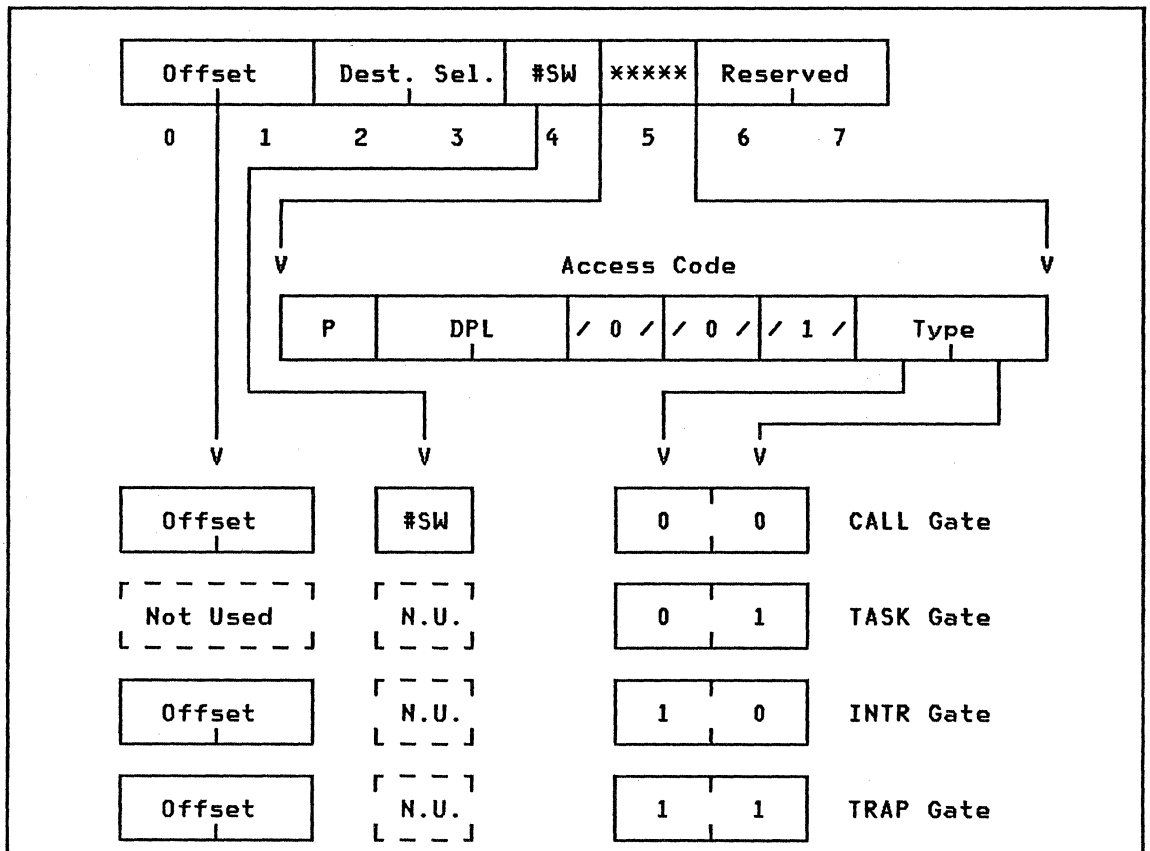


Figure 23. Gate Descriptor.

A key feature of a gate is the redirection it provides. All four gate types define a new address which transfers control when invoked (the destination selector plus offset in the gate descriptor). This destination address normally cannot be accessed by a program. When a program invokes a gate to transfer control, only the selector portion (loaded into CS) is used. The offset portion specified in the program is ignored and the offset in the gate descriptor is used instead. All that a program need know about the desired function is the selector required to invoke the gate. The 80286 will automatically start the execution at the correct address.

A further advantage of a gate is that it provides a fixed address for any program to invoke another program. The calling program's address remains unaltered even if the entry address of the destination program changes. Thus, gates provide a fixed set of entry points that allow a task to access, for example, operating system functions such as simple subroutines, yet the task is prohibited from simply jumping into the middle of the operating system's code.

Call gates are used for control transfers within a task which must either be transparently redirected or which require an increase in privilege level. A call gate normally specifies a subroutine at a greater privilege level, and the called routine returns via a RET instruction.

Trap and interrupt gates handle interrupt operations, and we will come back to them when we discuss interrupt vectoring.

Task gates are used to control transfers between tasks, and this leads us to the second category of control transfers -- transfers of control between tasks (multitasking).

An 80286 task is a single, sequential thread of execution. Each task can be isolated from all other tasks. There may be many tasks associated with a 80286 processor, but only **one** task executes at any time. The state of a task (from the processor's point of view) is the contents of the registers used by the task.

Switching the processor from executing one task to executing another can occur as the result of either an interrupt or an inter-task call or jump. The architecture of the 80286 defines a special type of hardware-recognized segment, the Task State Segment (TSS), for storing the 80286-related state of a task. The format of the task state segment is shown in Figure 24 on page 158.

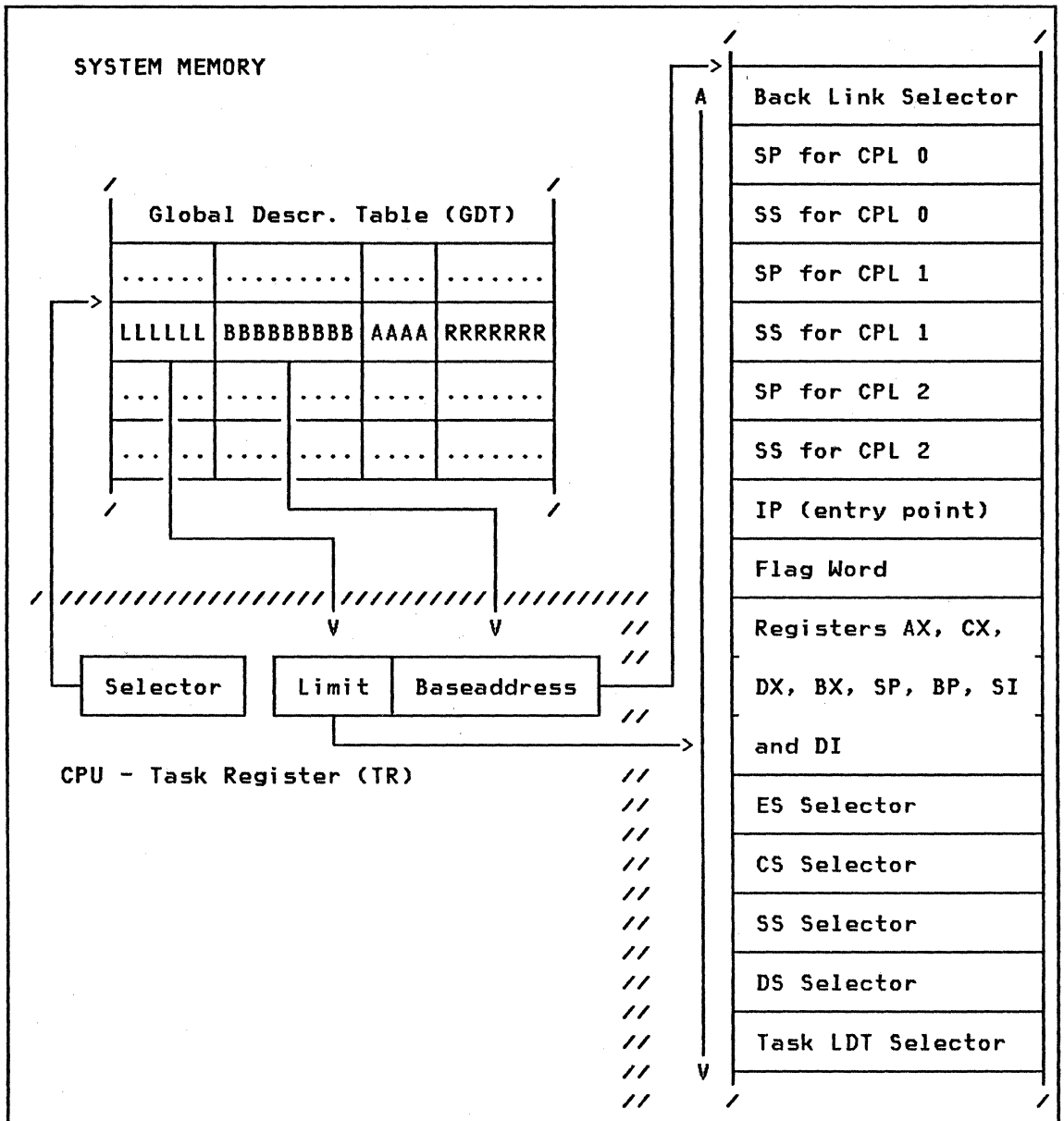


Figure 24. Task State Segment.

Multitasking operating systems on any processor need to store similar information. The 80286 processor requires a specific format of the TSS that will enable it to provide a very high performance in task switch operations with complete isolation between tasks. A full task-switch operation takes only about 25 microseconds on a normal PC/AT. This would enable the PC/AT to give control to about 35- to 40,000 tasks in one second (assuming the tasks do nothing more than a return instruction).

A special segment descriptor is used for Task State Segments. Refer to Figure 22 on page 154. This type of descriptor must be accessible at all times. Therefore, it can only appear in the Global Descriptor Table (GDT). The Task Register within the processor contains a selector to the

current task segment descriptor. Each TSS selector value is unique, providing an unambiguous "identifier" for each task.

A TSS contains 22 words that define the contents of all registers and flags, the initial stacks for privilege level 0 to 2, the Local Descriptor Table (LDT) selector for the task's private address space, and a link to the TSS of the previously executing task.

9.6.1 Task switching

A task switch may occur in one of four ways:

1. The destination selector of a long JMP or CALL instruction refers to a TSS descriptor in the GDT. The offset portion of the destination address is ignored and the IP (entry point) in the TSS is used instead.

2. An IRET instruction is executed when the NT (Nested Task) bit in the flag register is on. The new task TSS selector is in the back link field of the current TSS.

3. The destination selector of a long JMP or CALL instruction refers to a task gate. The offset portion of the destination is ignored and the new task selector is in the gate. Refer to Figure 23 on page 156.

4. An interrupt occurs. The interrupt's vector refers to a task gate in the Interrupt Descriptor Table (IDT). The new task TSS selector is in the gate. (We will come back to this when discussing interrupt handling later).

There is no new instruction required for a task switch operation. The standard JMP, CALL and IRET instructions used in real mode and in the 8088 processor are still valid. The distinction between the standard transfer-of-control instruction and a task switch is made either by the type of descriptor referenced or by the NT (nested task) bit in the flag register (for the IRET instruction).

Access to TSS and task gate descriptors is restricted by the rules of privilege levels. The data access rules are used, thereby allowing task switches to be restricted to programs of sufficient privilege. Address space separation does not apply to TSS descriptors since they must be in the GDT.

Once access to the TSS has been granted, the task switch operation involves six steps:

1. Recognizing that the JMP/CALL/IRET instruction or the interrupt requires a task switch. The new TSS to use is defined either directly by the TSS descriptor granted or is in the task gate descriptor.

2. Checking that the current task is allowed to switch to the designated task. The current task becomes the outgoing task.

3. Checking that the new task is present and has a proper TSS limit. The new task becomes the incoming task.

4. Saving the state of the outgoing task. The outgoing TSS selector is in the task register (TR). The dynamic portion of the outgoing TSS is written with the corresponding processor register values (e.g. AX, BX, CX, DX, SI, DI, BP, SP, ES, DS, SS, CS, IP and flags register).

5. Load TR with the incoming task selector, mark the incoming task's descriptor as busy, and set on the TS bit in the Machine Status Word (MSW).

6. Load the incoming task state (the following registers are loaded: LDT, AX, BX, CX, DX, SI, DI, BP, SP, ES, DS, SS, CS, IP and flag register). If the switch was due to a CALL (or interrupt), the NT (nested task) bit is set on, and the back-link field in the new TSS is set to point to the previous TSS. If the switch was due to JMP or IRET, the old task's descriptor type code is reset, indicating that the task is no longer busy. Resume execution of new task.

9.7 INTERRUPT VECTORING IN PROTECTED MODE

Let us now look at what happens when the processor is interrupted by some external or internal event.

The processor associates each event with an identifying number in the range 0-255. The processor recognizes three classes of events:

External Events occurring outside the 80286 processor's environment are communicated to the processor via the INTR or NMI (non-maskable interrupt) pins. The NMI is interrupt 2. Other external interrupts share the INTR pin via one or more 8259A Programmable Interrupt Controllers, which can map each interrupt to a unique interrupt ID in the range 32 - 255.

Processor When the processor detects a condition that it cannot handle (e.g. Divide error, Undefined Opcode, General Protection Exception etc.), it communicates this fact by causing an interrupt with an ID in the range 0-16.

Software Programs can signal events by executing the instructions 'INT n' and INTO (INTerrupt on Overflow). With 'INT n', the value n can be any interrupt identifier in the range 0-255. This gives software the ability to simulate hardware interrupts as well as the ability to cause interrupts that are not directly associated with hardware events.

When the 80286 is running in real address mode the interrupt vector is used as an index from the bottom of real memory (address 0000:0000). This is exatly the same process that has been described in earlier lectures on the 8088 and PC architecture.

In protected mode the interrupt vector is instead used as an offset into an Interrupt Descriptor Table (IDT). The IDT is a segment somewhere in storage based on the IDT register in the processor. This register is like the GDT register normally loaded during the initialization phase.

The IDT contains a minimum of 32 descriptors for the lowest 32 interrupt vectors. If more interrupts are used, descriptors are added to the IDT and the limit value of the IDT register has to reflect these additions.

The descriptors in the IDT are all gates and can be of three types:

- Trap Gate Descriptor
- Interrupt Gate Descriptor
- Task Gate Descriptor

The only difference between a trap gate and an interrupt gate is that the interrupt gate specifies a procedure that enters with interrupts disabled, while entry via a trap gate leaves the interrupt status unchanged.

Referring to Figure 25 on page 162 we can see that when the interrupt arrives, the interrupt vector is used as an index into the IDT. The gate descriptor contains a segment selector part and an offset part. The selector part is loaded into the CS selector (pointing to a segment descriptor in LDT or GDT), and the offset part is loaded into the IP register (the old values in CS and IP plus the flag register are saved on the active stack).

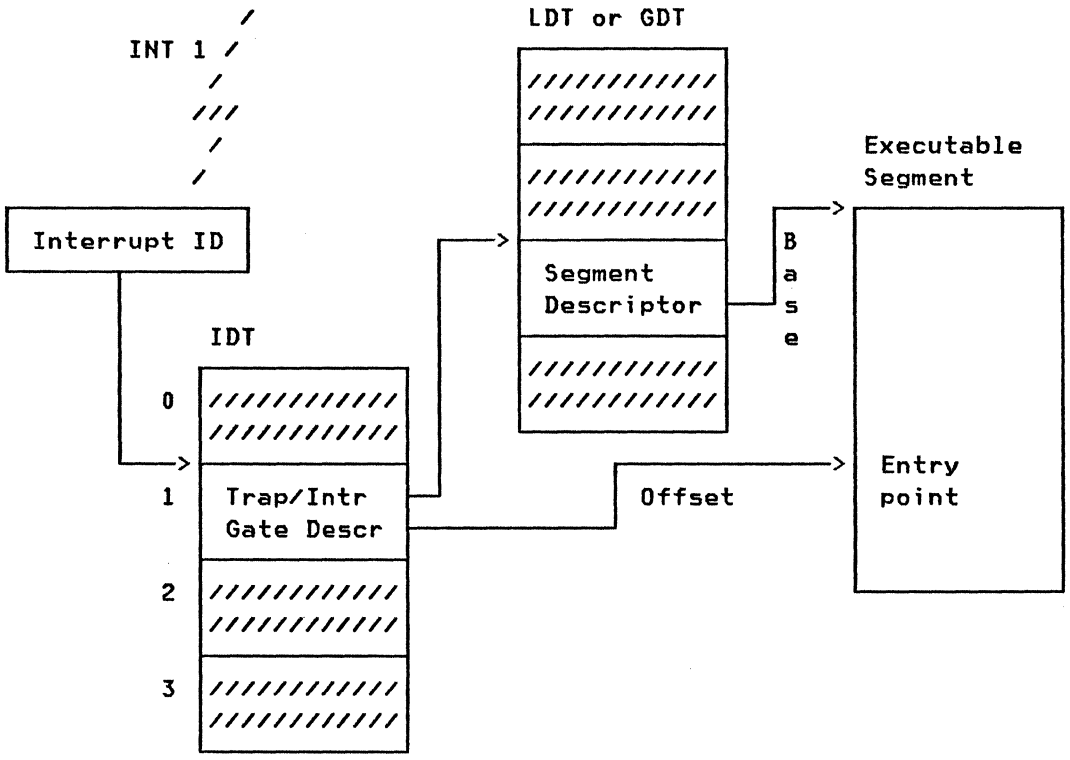


Figure 25. Interrupt Vectoring for Procedures.

Another way of implementing interrupt handling routines is to have a task gate descriptor in the IDT. Refer to Figure 26 on page 163.

Lets look at the interrupt vectoring for tasks:

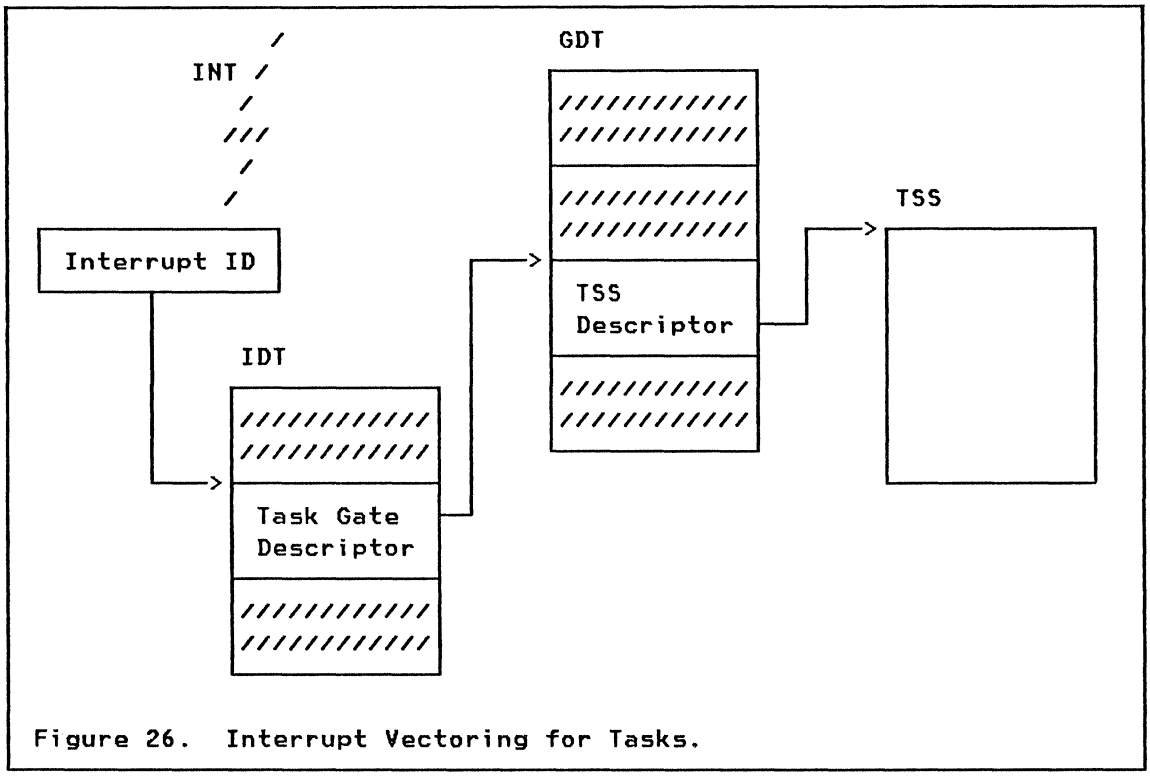


Figure 26. Interrupt Vectoring for Tasks.

The task gate points to a Task State Segment Descriptor in the GDT. This causes the processor to switch tasks.

A task gate offers two advantages over trap/interrupt gates.

1. It automatically saves all of the processor registers as part of the task-switch operation whereas a trap/interrupt gate saves only the flag register and CS:IP.

2. The new task is completely isolated from the task that was interrupted. Address spaces are isolated and the interrupt-handling task is unaffected by the privilege level of the interrupted task.

9.8 INPUT/OUTPUT AND PROTECTION

The concept of protection as applied to I/O consists of two parts. One part is the set of memory protection schemes which we already have described in detail. This part of the protection is very useful to protect memory-mapped I/O which is often used in conjunction with intelligent controllers. The communication with the controllers is achieved by interchanging information in specific common memory areas. These areas can then easily be restricted to specialized system tasks and certain privilege levels. The second part of the I/O protection scheme is what makes it possible to limit the right to execute the I/O instructions and some related instructions.

The instructions are:

IN	input (byte or word)
INS	input string
OUT	output (byte or word)
OUTS	output string
STI	set interrupt flag (enable interrupts)
CLI	clear interrupt flag (disable interrupts)
LOCK	lock bus

When the processor is interpreting any of these restricted instructions, it compares the Current Privilege Level (CPL) with the I/O Privilege Level (IOPL) in the flags register. If CPL exceeds IOPL, the processor causes a general protection exception and does not carry out the instruction.

The IOPL part of the flags register can only be changed by a procedure running on privilege-level 0. There is no instruction that explicitly affects IOPL. However, any of the operations that load the flag word can, in some cases, change IOPL. The only mechanisms for changing the flag word are:

- A task switch
- The POPF (pop flags) instruction
- IRET

When CPL is greater than zero, the POPF instruction does not change IOPL, even though it changes other flags in the flag word. The processor issues no error indication when this occurs.

A task switch loads the flags from the Task State Segment (TSS). As long as the operating system does not make data-segment aliases for the TSS available to less privileged levels, only the operating system can change IOPL in the TSS.

To summarize the protection of I/O-operations, we can say that the 80286 has complete protection for both memory-mapped and port I/O, if the operating system uses it in the right way.

9.9 CONCLUSION

We have come to an end of the 80286 architecture module. The reason for the module in this course was to give some information about the protected mode for workstation support personnel. There is almost nothing published (outside INTELs manuals) about how the processor works in this mode, and we already have some products that are using it on the market.

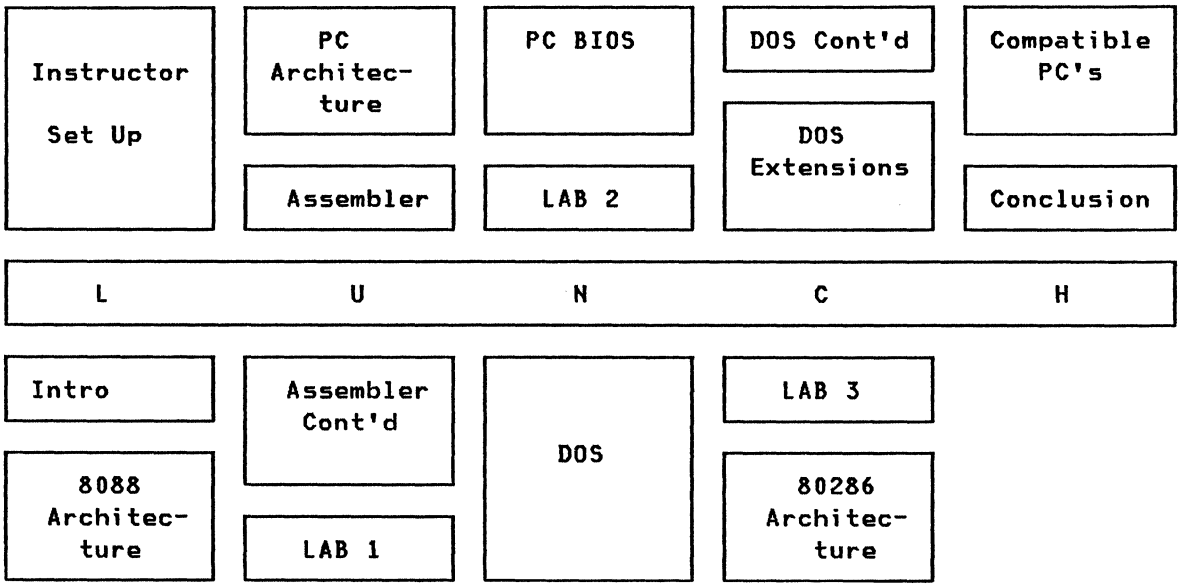
We have attempted to give the student an understanding of what is going on in a very complicated microprocessor. If you feel that you have not understood it after reading it once, then remember that this is something of a chicken-and-egg problem: a second reading may make it a little clearer.

APPENDIX A. SAMPLE "C" PROGRAM COMPLETE LISTING

Use this listing if you want to compile the sample "C" program, because the one cited in the guide will not work. This version merely adds two "include" files for the compiler's use. The program is compiled using the IBM PC "C" Compiler for DOS.

```
/* TESTC.C */
/* Gets a character from keyboard                */
and echoes its uppercase to display.              */
/* This program loops 20 times, then exits.       */

#include <conio.h> /*needed for C function calls */
#include <ctype.h> /*needed for C function calls */
int count;
int keystroke;
main()

     {
     int keystroke;   /*reinitialize for this routine */
     while (count < 20)
          {
          keystroke=getch();
          putch(toupper(keystroke));
          count++;
          }
     }
```

APPENDIX B. SCHEDULE OF IN-CLASS VERSION OF COURSE



APPENDIX C. 8088 ASSEMBLER LAB

This lab is designed to accomplish three objectives:

1. To teach you how to use some of the features of the Resident Debug Tool

2. To show you an assembler program in both source and object formats, and allow you to follow the object through while using the source for reference

3. To let you use some of the instructions we have seen so far, and to let you change some of them using machine language encoding.

You should not have any difficulty in completing the lab within one hour, providing your instructors have set up the PC's in advance for the lab. You should have a diskette from the instructors containing the sample programs, and a PC with which to work.

1. Turn on the PC and load DOS if it doesn't load itself. At the DOS prompt, type "RDT". (In this all future instructions on what to type, we assume you will press the Enter key after each instruction). This will load the Resident Debug Tool, which allows you to step through the machine code of a program. You will see a logo screen; press a key and the DOS prompt will return.

2. Now type "ASMLAB". This loads the file ASMLAB.COM, which is the object code for the source file "ASMLAB.ASM", a printout of which is attached to these lab notes.

3. A totally different screen will appear: The RDT Memory screen. How did this screen appear? It is not part of the ASMLAB program. It was, however, invoked by our program, using an interrupt, INT 3, which RDT traps.

Type "DW". This will take you to the Disassembly window, which we will examine first. Read through the following example and compare it to what you see on the screen. Make sure you understand what our definitions of the lines mean. If you don't, ask a lab instructor.

DISASSEMBLE WINDOW

```
1. Release and program title line
REL 1.00       IBM PERSONAL COMPUTER RESIDENT DEBUG TOOL    D1    07/01/84

2. Variable line - V1...V9 can hold 20-bit hex values
V1:.... V2:.... V3:.... V4:.... V5:.... V6:.... V7:.... V8:.... V9:....

3. Breakpoint variable line - S1...S9 can hold breakpoint addresses
S1:.... S2:.... S3:.... S4:.... S5:.... S6:.... S7:.... S8:.... S9:....

4. Window ID line. Characters are in ASCII, window is DISASSEMBLY
                                      DISPLAY: ASCII     WINDOW: DISASM

5-7. Register area, with contents                  Flag area

AX:  0000  BX:  0000  CX:  00FF  DX:  1BA2           TR:00  .....-.....
SP:  FFFE  BP:  0882  SI:  0100  DI:  FFFE   FL:F246 OF:0 DF:0 IF:1 TF:0
CS: 1BA20  DS: 1BA20  SS: 1BA20  ES: 1BA20      SF:0 ZF:1 AF:0 PF:1 CF:0

8. Current instruction
    LC: ....        INT     3                                  OP: .....

9.  IP value        CS+IP   Machine instruction
    IP: 0100    EX: 1BB20   CC              STEP CT: 0001  CO: .....

10. Command line ==>

11. Status line
                BLANK

12-25. Disassembly lines L1-M5.

a      b     c     d             e       f                    g

L1: *  1BA20:0100  CC            INT     3
L2:    1BA20:0101  B91400        MOV     CX,0014
L3:    1BA20:0104  BE2101        MOV     SI,0121
L4:    1BA20:0107  BF4101        MOV     DI,0141
L5:    1BA20:010A  8A04          MOV     AL,DS:[SI]           1BB20=CC
L6:    1BA20:010C  3C61          CMP     AL,61
L7:    1BA20:010E  7206          JB      1BB36                1BB36
L8:    1BA20:0110  3C7A          CMP     AL,7A
L9:    1BA20:0112  7702          JA      1BB36                1BB36
M1:    1BA20:0114  2C20          SUB     AL,20
M2:    1BA20:0116  8805          MOV     DS:[DI],AL           2BA1E=00
M3:    1BA20:0118  47            INC     DI
M4:    1BA20:0119  46            INC     SI
M5:    1BA20:011A  E2EE          LOOP    1BB2A                1BB2A

a. Line identifier L1...M5
b. Code segment
c. Offset
d. Hex for machine code (1-6 bytes)
```

e. Instruction mnemonic, disassembled from machine code
f. Operand(s)
g. Address references for control statements (eg. L7)
Address references indicating value at address, for addressing statements (eg. L5)
Blank for instructions which do not reference a location through addressing or control transfer (eg. L1)

Now type "MW" to go back to the memory window. Compare the screen you obtain with the one we have identified below.

MEMORY WINDOW

```
1-10: Same as in Disassembly Window
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:  1BA2           TR:00  .....-.....
SP:  FFFE  BP:  0882  SI:  0100  DI:  FFFE  FL:F246 OF:0 DF:0 IF:1 TF:0
CS: 1BA20  DS: 1BA20  SS: 1BA20  ES: 1BA20     SF:0 ZF:1 AF:0 PF:1 CF:0
    LC: ....        MOV     CX,0014                          OP: .....
    IP: 0101    EX: 1BB21   B91400          STEP CT: 0001  CO: .....
==> IP=IP+1;EX
               EXECUTING
12-25. Memory lines L1...M5
a       b        c        d        e        f              g
L1  * 00000  E84E2E01 7B96BF09 7104BF09 C304BF09  *.N......q.......*
L2    00010  F0017000 54FF00F0 23FF00F0 23FF00F0  *..p.T...........*
L3    00020  A5FE00F0 9607BF09 23FF00F0 23FF00F0  *................*
L4    00030  23FF00F0 600700C8 57EF00F0 F0017000  *........W.....p.*
L5    00040  65F000F0 4DF800F0 41F800F0 AB097000  *e...M...A.....p.*
L6    00050  39E700F0 59F800F0 2EE800F0 D2EF00F0  *9...Y...........*
L7    00060  000000F6 DD0A7000 6EFE00F0 EA017000  *......p.n.....p.*
L8    00070  4BFF00F0 A4F000F0 22050000 000000F0  *K...............*
L9    00080  C3122E01 11031009 2F01DC1A 3C01DC1A  *................*
M1    00090  C404DC1A 5B142E01 9E142E01 4D5B2E01  *............M...*
M2    000A0  C9122E01 4A017000 C9122E01 C9122E01  *....J.p.........*
M3    000B0  C9122E01 C9122E01 57023B08 AB152E01  *........W.......*
M4    000C0  EACA122E 01122E01 C9122E01 C9122E01  *................*
M5    000D0  C9122E01 C9122E01 C9122E01 C9122E01  *................*
```

a. line identifier
b. 20-bit address
c,d,e,f. 4 bytes each of memory contents, low to high memory
g. ASCII equivalent of values in the hex portion of the line

Notice that the addresses start at 00000. What is in this location? The interrupt vector table. Each block of four bytes of memory is the address for one of the interrupt vectors. When a program or an I/O event issues an interrupt, the 8088 looks in this table to find the address of the interrupt routine. The address is stored in the following format: OL OH SL SH where O indicates offset, S indicates Segment, and L and H indicate

Low and High. See if you can figure out the interrupt routine address for INT 0. On our example above, it's 012E:4EE8.

Now type "DW" and watch the disassembly window reappear. Notice that the IP register contains 0100 hex, not 0000. Why is this? We have not executed 100h bytes of our program yet. DOS however has reserved a 100h byte header to our COM program; this header contains information we could use if we wished. However, we will not study this header until the DOS lecture.

4. Let's do some executing. Type "ST" at the command line. What happens? Apparently, not much. "ST" means "single-step": it steps the current piece of code one or more steps at a time. By asking RDT to step the INT 3 instruction, we simply reinvoked RDT and nothing has happened. In order to actually execute our program now in debug mode, we will have to bypass the INT 3 instruction.

5. Type "IP=IP+1". It is illegal to change the contents of IP in this way as far as the 8088 is concerned, but RDT lets you get around this. We are now at the instruction "MOV CX,0014". This instruction should be on the L2 line, and on the line which indicates the current instruction. Check our source code for this step. Why does the source code say 20, and this say 14?

Because the value in our source code is decimal, whereas RDT operates in hexadecimal. 20D=14H.

6. Now we can step through parts of our program. Watch the CX register as you enter the next command. Type "/ST". Notice that the CX register now contains 0014; its previous contents are lost. However, they are not needed by the program, and if DOS needed them, it saved CX on the stack before loading our program.

7. Because you typed a slash in front of "ST", the same command reappears. We will step through another instruction. Here, we load SI with a value. Wait! Where did this value come from? Check your source code.

The value is actually an offset address into the current data segment (which in this case is also the current code segment, the current extra segment, and the current stack segment!). The assembly process has translated a label into a relative address. By loading this value into SI, we know that DS:SI will point to the source data area in our program. Before you check this, press Enter to load SI; then press Enter again, and DI will also be loaded, so it now points to our destination string.

8. Now let's check that our strings are being addressed by DS:SI and ES:DI. Type "MW". Notice that lines L1 through M5 no longer contain code, they now contain data.

9. We would like to look at the contents of DS:[SI] and ES:[DI]. So type the following: "L1=DSSI;L3=ESDI". You can type it all on one line; the semicolon separates the two commands. We can now view our source and destination data areas directly on the screen. If you have done everything correctly, the first line should contain the phrase contained in our source string. Check the ASCII characters to the right to confirm this.

The third line should contain zero's, since we haven't put anything there yet.

10. Let's go back to the disassemble window. Type "DW". What is our current instruction about to do?

It is about to load the AL register with the memory location at DS:[SI]. We know what is in this location: the first byte of our source string. DS:[SI] is generally used to address a source string, whereas ES:[DI] usually addresses a destination string.

11. Let's load AL with the byte at DS:[SI]. Type "/ST". Notice that AL changes, but that AH remains unchanged.

12. Keep pressing Enter to reinvoke the "/ST" command until you get to the statement "LOOP". You should understand what each of the assembly language instructions is doing. If not, consult the source code listing, which explains each instruction's function. Notice as you step through how the registers change contents according to what the instructions are doing. Notice that we update SI and DI after we have done our work on the byte in question -- the next time around, we want to access the next byte.

13. You should now be at the "LOOP" statement. If you accidentally stepped past it, don't worry; just don't step it when we tell you to in a moment. What does the "LOOP" statement do? Check our source code listing to find out. "LOOP" is a control instruction. Why do we want to loop if CX is greater than 0? Because CX indicates how many times we still have to do the work, and until zero times remain, we want to keep at it. Now press Enter to go back to the top of the loop. Try to find where we are in the source code listing. Also, look at CX: its value has decreased automatically from 0014 to 0013. The LOOP instruction decrements CX before it transfers control.

14. Let's step through all the instructions again. This time, watch for the flags before and after we do the two "compare" instructions. (Or only the first instruction -- sometimes we never get to the second one. Try to figure out why we might not.) Keep stepping until you get to the LOOP statement; step one more, and the top code line should return to the statement which loads the next source byte into AL.

15. Go back to the memory window and look at the ASCII zone on the right of L1 and L3. What has happened? Has the program worked? If you don't see a character or two of the source string capitalized in the destination string, something terrible has happened -- our program hasn't worked. Call an instructor.

16. Go back to the disassembly window. We'll now use one of the Breakpoint variables -- S1 through S9. These variables allow us to specify an address in memory. They are called Breakpoint because, when we execute our program, if one of these locations becomes the current instruction location (ie. the sum of CS and IP) execution will stop and we will be able to use RDT once again.

You should be able to find the line in the code area which contains the LOOP statement. If not, try stepping through your code until the LOOP statement appears. Once you have found it, you will load S1, the first breakpoint variable, with the address of the LOOP statement. How do you do this? Simple! Type "S1=Xn", where "Xn" is the line identifier of the line containing the LOOP instruction -- L1...L9, or M1...M5. Now when we execute, the program will be stopped each time the LOOP instruction is encountered, in other words, after each character is processed.

17. Go back to the memory window and execute -- type "EX". Notice that another letter gets converted to uppercase and placed in the destination area. Execute a few more; more letters should appear. Don't go beyond the space after "there" -- we have other work to do with Annabel.

18. Go back to the disassembly window. Step from the LOOP statement to the statement which loads AL. Our next project is to alter the assembled code dynamically. RDT allows us to change the hex values of the opcodes and operands -- the machine code -- just by moving the cursor down to the hex value to change, and typing in the new value. What we want to do is change the program so that instead of converting lowercase to uppercase, it converts uppercase to lowercase. You must figure this out for yourself.

Here are the only three hints we can provide for the moment:

- Lowercase 'a' is 61 hex. Uppercase 'A' is 41 hex.
- Lowercase 'z' is 7A hex. Uppercase 'Z' is 5A hex.
- The machine code for "SUB" with byte values is 2C hex, followed by the operand byte. The machine code for "ADD" is 04 hex, followed by the operand byte.

Go to it. Think of what statements need modifying. First of all, if you're checking for uppercase values, not lowercase, you'll have to change the range of values to search for. Secondly, you'll have to change the way in which you alter those values which are in the valid range. You don't want to subtract; you want to add. You can do all this just by poking three bytes of the machine code. Don't bother reading on until you've accomplished your task. Ask the instructor for help.

19. Did you succeed? The three bytes to change were:

a) At offset 010DH: 61 change to 41
b) At offset 0111H: 7A change to 5A
c) At offset 0114H: 2C change to 04.

The first change alters the lowest ASCII value we want to change: Instead of 'a', we are looking for 'A'. The second change alters the highest ASCII value to 'Z' instead of 'z'. The third change converts any of these values to lowercase from uppercase, instead of vice versa, by adding 20H instead of subtracting 20H ('a'-'A'=20H).

20. Press HOME to return to the command line. Now step through your program from the disassembly window once -- until you've executed your

changed instructions and gone back to the LOOP statement. Switch to the memory window; make sure the conversion hasn't produced a wrong value. The value may not have changed -- it should only change if it was upper-case to start with.

21. Type "EX" three or four times. Notice the conversion: it should be reversed from the initial conversion. Once you've converted most of the characters, if the program hasn't already exited to DOS, type "S1" (ess-one, not ess-eye), which will clear the breakpoint register. Type "EX" again, and you will exit to DOS without seeing any of the remaining changes.

Through this lab we covered a number of 8088 assembly language instructions -- MOV, ADD, SUB, CMP, conditional jumps (JA, JB), INC, LOOP, and INT 3. These are some of the more commonly used instructions. In later labs we will use other instructions as well. It's best just to learn a few instructions at one time. We also learned a bit about how to use RDT, and saw how we can use some of the registers to address data and code. We did not use the stack in this example, although it was used by DOS and RDT in hidden ways. If you have extra time at the end of this lab, and an instructor is available, ask him or her to show you how the stack is used.

```
;* ASMLAB.ASM
;* Reads a character from the source data area of the program,
;* converts it to uppercase, and places it in the destination
;* data area. This program loops 20 times, then exits.


CODE    SEGMENT PARA PUBLIC 'CODE'
        ORG 100H                ;---establish structure of a .COM file
        ASSUME CS:CODE,DS:CODE,ES:NOTHING,SS:NOTHING

MAIN    PROC    NEAR            ;---MAIN PROC is the main procedure of
                                ;   the program

        INT     3               ;---Interrupt 3 invokes the Resident
                                ;   Debug Tool if RDT is loaded. If
                                ;   no debugging program is
                                ;   loaded, INT 3 has no effect.

STEP01:                         ;---STEP01 sets the count to 20, which
                                ;   means we do this routine 20 times.
       MOV      CX,20      ;CX is count register

STEP02:
       MOV      SI, offset SOURCE
                                ;---Sets the source index register
                                ;   to point to the first byte of
                                ;   the source data area (see the
                                ;   bottom of the program for the
                                ;   contents of SOURCE and DEST)

       MOV      DI, offset DEST

                                ;---Sets the destination index register
                                ;   to point to the first byte of
                                ;   the destination area
STEP03:
       MOV      AL, [SI]        ;---Moves a character from the
                                ;   source data area into AL

STEP04:
       CMP      AL, 'a'         ;---Compares the value to 'a'

       JB       STEP07          ;---Jump if the character is
                                ;   below lowercase 'a'
                                ;   (since it's not lowercase)

STEP05:
       CMP      AL, 'z'         ;---Compares the value to 'z'

       JA       STEP07          ;---Jump if the character is
                                ;   above lowercase 'z'
                                ;   (since it's not lowercase)
```

```
STEP06:
        SUB     AL, 32          ;---Subtract 32 from the character
                                ;   (this converts it to uppercase)

STEP07:
        MOV     [DI],AL         ;---Places the character into
                                ;   the destination data area

        INC     DI              ;---Increment both DI and SI
        INC     SI              ;   pointers so that they point
                                ;   to the next character to be
                                ;   converted

STEP08:
        LOOP    STEP03          ;---If CX>0 then decrement CX,
                                ;   loop and keep on converting.
                                ;   LOOP LABEL is the same as the
                                ;   following two statements:
                                ;   DEC CX
                                ;   JNZ LABEL (jump not zero)

STEP09:                         ;---STEP09 is reached if the LOOP fails,
                                ;   ie. if CX is 0 and we've done our 20
                                ;   chars.

         MOV    AX, 4C00H       ;return control to DOS
         INT    21H

SOURCE   DB      'Hello there ANNABEL!'
                                ;---SOURCE is a data area which contains
                                ;   the data we wish to convert to
                                ;   uppercase. Here it is coded in
                                ;   ASCII format.

         DB      12 DUP (0)     ;---Defines 12 zeros.


DEST     DB      20 DUP (?)     ;---DEST is the data area we will
                                ;   put the results of the calculation
                                ;   into. The ? means the data is
                                ;   not initialized -- its contents
                                ;   are undefined at program load time.

         DB      12 DUP (-1)
                                ;---This data definition defines
                                ;   twelve bytes of -1 (FF hex)

MAIN     ENDP               ;end of MAIN procedure
CODE     ENDS               ;end of contents of CODE segment
         END     MAIN       ;END directive -- says where program begins.
```

APPENDIX D. BIOS LAB

The purpose of the BIOS lab is to give you a chance to learn how some of the BIOS routines are invoked, to look at some of the reserved memory areas of the PC and how they are used, and to strengthen your familiarity with RDT and with 8088 assembly language. In the course of this lab you will invoke a program coded for the BIOS interface, which performs the same things our "C" program in the introduction did: reads 20 keys from the keyboard, and outputs them in uppercase to the display. You will also issue some BIOS calls yourself to see what they do.

1. From DOS, load RDT with the 'A' and 'K' options: Type "RDT A K"

2. Invoke the BIOSLAB.COM file.

3. RDT's memory window appears. Change to the disassembly window.

4. The source code listing for this program is provided in these lab notes. Read through some of the first lines and see what the program is doing. Notice that INT 16H fetches a keystroke from the keystroke buffer. Let's step through some of what INT 16H does. Wait -- we'd better set a breakpoint first.

5. Set breakpoint variable S1 to the instruction after the INT 16H instruction. Remember -- type "S1=Nx" where Nx is the line designation for the line following the INT 16H. Then type "IP=IP+1" to set the instruction pointer to the instruction right after the INT 3. Now let's step, using the "/ST" command.

6. Notice the "STI" instruction, the first instruction INT 16H invokes. This allows other interrupts to be enabled while the keyboard routine is processing input, so that I/O is not lost. After the STI instruction, we see the INT 16H routine pushing two of the registers onto the stack since it modifies these registers. It then calls a special routine which we don't need to understand, but the routine is short so we can step through it. You'll know you're back in the routine that called it when you see the OR AH,AH instruction. Keep stepping until you encounter another STI instruction. You should see the following:

```
STI
NOP
CLI
```

These three steps turn interrupts on, perform a no-operation instruction, and turn interrupts back off. The NOP instruction does nothing at all, except take a bit of processor time to process the instruction. This tiny delay allows an interrupt to occur -- that is, allows a key to be struck and read in by the INT 9 routines, which we WON'T be looking at. Let's step through STI, NOP and CLI. If you keep stepping, you'll see that we keep coming back to STI, NOP and CLI. Why? No key has been struck, so the routine just keeps looping, waiting for a key. You could try hitting a

key in the brief instant after you start one of those steps. But chances are the key won't get to BIOS.

7. Let's be done with following the bits and bytes of INT 16H -- it's there to make our lives easier, not to be admired and worshipped. Now we'll execute. The system will wait for you to type a keystroke. You can type "Hello there Annabel" if you like, but we can't enforce this. Type something; before the second character can even be entered, we're back into our program. Because RDT is doing funny things with the keyboard hardware interrupt routine (INT 9) the input isn't being buffered right now. Don't worry about why. You'll do some buffering later, if you have time.

8. Let's now go through some of the video BIOS routines. Step through our program until you get to the statement "MOV AH,0A". This is the first line of preparation for a BIOS video function call. Three lines down you'll see the "INT 10" instruction which calls the BIOS interface. Set a breakpoint variable to the line which follows the CALL statement which is directly below INT 10H.

9. You may have noticed a pause between the character being displayed and the return to RDT. This is because we placed a CALL statement immediately below the BIOS interrupt which invoked a small routine to force a pause on the system. The routine CALLed essentially places FFFF into CX and decrements CX by 1 until CX=0; then it returns. The reason we placed this pause in the program is that for students using RDT with a single display on their system, without the pause there would not be enough time to see the character appear on the display.

10. You should have noticed that the cursor did not advance after the character was displayed. The Video BIOS does not actually advance the cursor automatically. Instead, we must ask the BIOS what the cursor position is, then tell the BIOS to update the cursor position. Step down to the statement "MOV AH,03". Check your source code listing to see what this and the following steps do. Set a breakpoint variable to the line **after** the "CALL" statement which calls the same old pause routine. Execute again. You won't see anything happen; but look at the contents of the DX register. DH is set to the current row number; DL is set to the current column number. If we want to move the cursor one column right, what do we do?

11. We increment DL. This is what our program in fact does soon afterwards. Note that, if we were really in the business of programming, we would also check that the column was below 80 decimal -- the last column on the display -- and if it wasn't, we would include a routine to increment the row number as well so that we wouldn't overwrite previous information.

12. The source code listing tells you that by loading AH with 2 and BH with the current video page, and issuing the interrupt, the cursor position will be updated. Don't forget, though, that we also used the value returned to us in DX by the previous interrupt, and changed it.

13. Clear all breakpoint variables that are still set (unless you feel like stepping or executing your way through them again -- it won't matter

which you do, except that if you don't clear them things will take a little longer). Now set one breakpoint variable only -- to the "INC DL" line. Execute the program. If you left other breakpoints in, step through them. You will have to type a character, as usual. But **stop** when you hit the breakpoint at "INC DL". Step one step passed, to "MOV AH,02".

14. We shall now play a little trick on our program so that it doesn't do exactly what it was designed for. We will change the location of the next character to print. How? By changing the current cursor location. See if you can figure out a way to do this. We give only one hint and one caution. The hint is to reread step 11 above, where we identify how the cursor position is set. The caution is to remember that the row and column maximums are 25 and 80 decimal respectively, and that, since RDT works in hex, you will want to limit yourself to 19 and 50 hex.

So set up your registers for the INT 10H call; now execute. Type another character; it should appear where you moved the cursor to. If it didn't, here's what you should try next time through:

After you have stepped the INC DL step, you want to change the contents of DX so that they address the middle of the screen. The middle of the screen is at row 19h/2, column 50h/2. This gives us row 0Dh, column 28h. So set DH to 0Dh, and DL to 28h. To do this type "DX=0D28". After have done so, execute again. This should set the cursor to the middle of the screen. If not, call your instructor.

15. We have had a good look at keyboard and video BIOS support. Now let's do some BIOS bypassing -- first, with video. As you recall, video memory is mapped to segment B000 for monochrome, B800 for color graphics. If your station is using an EGA adapter, ask the instructor what compatibility mode it's set up for. (If the screen displays multiple colors, it's in Color Graphics mode, so segment B800.)

We will look at the contents of the video memory -- well, sort of. Switch to the Memory Window; type "L1=XXXX0", where XXXX is the proper segment for the display type you are using. Believe it or not, you are now looking at the memory contents of the screen you are now looking at. If you turn your eyes to the ASCII character section of the screen, you should be able to read "R.E.L...1...0.0." and so on. What is this? It's the information on the top line of the screen. (If you don't see this, you've set L1 to the wrong value. Call an instructor if you need help.)

16. We can actually write directly to the screen in both senses of the phrase here. Use the cursor keys to move the cursor into the hexadecimal memory contents area. Start typing any hex values that come to mind. As you do so, notice how the contents of the upper portion of the screen are changing. Pretty neat, eh?

17. For our next video number, we'll just poke around a little more mischievously. Skip this step if you're running behind -- the keyboard section will be more worthwhile. On the command line type "L1=L1+6E0". This will cause the first line of the memory display to display a representation of -- the first line of the memory display. Here we have a little more trouble writing directly to the screen. Try overwriting the

hexadecimal memory area. What happens? Every time you move to the next byte, the previous one changes back. Why?

The reason is that every time you change a value on a line, RDT refreshes the whole line. So if you change the value for the "L" in "L1", then move the cursor, RDT refreshes the whole line; "L1" reappears, and therefore so does the hex value representing "L1" in the buffer. If you don't understand this, don't worry. It's not at all important.

18. Finally, let's look at the keyboard buffer. Clear all your breakpoints; execute. You will have to keep typing the full twenty characters. Then your program will exit to DOS.

Remember that when we invoked RDT today we used the "K" option. The "K" option, or switch, is used in RDT to tell it to trap the keyboard PrintScreen interrupt. What does this mean? It means that when you press Shift-PrtSc, which generates an INT 5 through the keyboard BIOS routine, that INT 5 is captured by RDT and you are brought into RDT. We are going to use this trick to fill the keyboard buffer up and jump into RDT before the buffer can empty.

Read this entire paragraph before you do any of it -- you will have to be very fast at typing. The best typist in your lab group should do this step. First, place a diskette in drive A: and leave the door **open**. Then, from the DOS command prompt, type "DIR A:" then Enter. Without waiting a second, type "Hello X" (not something else) and "Enter", then hit Shift-PrtSc as fast as you can. RDT will load; and your keyboard buffer is still full.

19. Close the diskette door. Now go to the memory window of RDT; set L1=0041E. You should see in the ASCII area beside L1 and L2 the characters "H.e.l.l.o...X." and other meaningless characters. You just typed these. They are still in the keyboard buffer. The beginning of your phrase and its end may be switched around -- you may, for example, see "llo X" and later "He" -- a good example of the keyboard buffer wrapping around from its highest position to its lowest. Move your cursor to the ASCII area of L1. You will now write over characters in the buffer. The buffer is located on lines L1 and L2. Start at the "H" of "Hello". Change the characters as described, without pressing Enter; just move around with the cursor. Make sure as you do this that you wrap around if you reach the end of L2, since the buffer itself wraps around also.

1. Where you see the "H" of "H.e.l.l.o", type "B".
2. Over the "e" type "I". Keep moving over two characters at a time.
3. On the first "l" (ELL) type "O" (oh).
4. On the second "l" type "S" (ESS).
5. On the "o" (oh) of "Hello" type "L".
6. Two positions further along, type "A".
7. Two more, type "B".

20. What did we just do? We just changed the contents of the keyboard buffer from "Hello X" to something else. You know what else, we trust. Move the cursor to the command line (press "Home"). Type "EX" which will execute us back out to DOS. The diskette drive should spin; you'll get a

directory; finally, BIOSLAB will be invoked. Ta-da! Lo - we load BIOSLAB again.

21. If you did all the above in less than an hour, congratulations, and here's more work for you. If you feel like stopping now, by all means do. The following exercises merely reinforce what you have already learned, they are not essential in any way.

We're sick of our little program by now, so let's not actually use it. Instead, we'll rewrite it. We will do two things: First, write 2,000 G's on the screen. Then, we'll load Cassette BASIC.

To write 2,000 G's, we'll first use the BIOS video call to set the cursor to the top of the screen. To do this, load AH with the function call (AH=02). Set DX to the row and column desired -- zero (DX=0). Set BH to the current page number, zero (BH=0). Now we want to invoke INT 10H. How shall we do that?

22.Go to the disassembly window. The first disassembled instruction should be INT 3, with the "CC" machine code byte to its left. We want to change the code from this point on to INT 10 and then INT 3 (so that RDT is reinvoked by INT 3 after the BIOS INT 10 returns. So: move the cursor over the first "C" in "CC". Type the following: "CD10" very slowly. Move the cursor down to the next line, on the "1" in "1400", and type "CC". These three values you just entered are the hex machine codes for INT 10H and INT 3, and right now you should see the mnemonics for these instructions to the right of the two lines you just changed. We have set up our machine code to invoke the BIOS video routine, then RDT again. So now execute. What happens? The cursor should move to the top of the screen.

23. Now we'll print our G's. First, let's set the instruction pointer back two bytes to point to the INT 10 instruction. Type "IP=IP-2". Don't worry about not being able to see the INT 10H instruction; it's there in the code segment.

We'll set up the registers using BIOS Video call 0Ah. So: Type "AH=0A". We've set the function call. Type "AL=47". 47 hex is the ASCII value for "G", and in the function call we're using we place the character to print in AL. Now put zero in BH (to make the active display page 0);place 900 in CX. CX contains the count of times to display the character. By setting the value to 900 we know we'll display **lots** of G's. Finally, execute. Lo and behold!

24. The final step is to invoke the ROM BASIC. When the BIOS bootstrap routine fails to find a valid boot record, it loads BASIC via INT 18H. So we will do just this -- load BASIC via INT 18H. INT 18H points to the BASIC ROM code. How do we do this? Move your cursor to the "CC" which popped up on the "L1" line after you displayed lots of G's. Change the "CC" to "CD". "CD", as you may have noticed earlier, is the machine code for the opcode in most "INT" instructions. Change the byte value which now follows "CD" to "18". The current instruction is now INT 18. Execute this; Cassette BASIC should appear. Now, you're stuck! You can play with BASIC; better just to end the lab by powering off the PC.

D.1 CONCLUSION

If you managed the whole lab you either skipped lunch or already know the BIOS intimately. In this lab we have looked at some common BIOS calls, played with the video and keyboard buffers, and done some general mischief with various BIOS calls. Please note that the final step in this lab, the invocation of BASIC, was not a BIOS call! It just happens that Cassette BASIC, like the BIOS, is stored in ROM, and that it's invoked by an interrupt. It has no other connection with BIOS.

```
;* BIOSLAB.ASM
;* Gets a character from keyboard and echoes its uppercase to  *;
;* display. This program loops 20 times, then exits.           *;


CODE    SEGMENT PARA PUBLIC 'CODE'
        ORG 100H                ;---establish the structure of a
                                ;   .COM file
        ASSUME CS:CODE,DS:CODE,ES:NOTHING,SS:NOTHING

MAIN    PROC    NEAR            ;---MAIN PROC is the main procedure
                                ;   of the program
        INT     3           ;invoke RDT

STEP01:                         ;---STEP01 sets the count to 20, so
                                ;   means we do this routine 20 Xs.
        MOV     CX,20       ;CX is count register

STEP02:                         ;---STEP02 saves the count each time we
                                ;   enter the loop, so that if CX gets
                                ;   modified by the interrupt routines
                                ;   we don't lose track.
        PUSH    CX          ;count now on stack

STEP03:                         ;---STEP03 uses the BIOS keybd routine
                                ;   to get a character from the keyboard.
        MOV     AX,0        ;Get character function call
        INT     16H         ;BIOS keyboard routine

STEP04:                         ;---STEP04 and STEP05 check the returned
                                ;   character to see if it's between 'a'
                                ;   and 'z'.
        CMP     AL,'a'
        JB      STEP07      ;if below 'a' then it's not lowercase so skip

STEP05:
        CMP     AL,'z'
        JA      STEP07      ;if above 'z' then it's not lowercase so skip

STEP06:                         ;---STEP06 subtracts 32 from the ascii
                                ;   value of the character, thus making
```

```
                                ;   it an uppercase character.
        SUB     AL,'a'-'A'

STEP07:                         ;---STEP07 through STEP09 print the
                                ;   character then move the cursor fwd
                                ;   one position.

        MOV     AH,10       ;function call to print character
        MOV     CX,1        ;1 character to print (character is in AL)
        MOV     BH,0        ;video page 0
        INT     10H         ;BIOS video display routine
        CALL    DELAY       ;Call a routine to cause a delay
                            ;so we have time to view the screen

STEP08:
        MOV     AH,3        ;function call to get cursor position
        MOV     BH,0        ;video page 0
        INT     10H
        CALL    DELAY

STEP09:
        INC     DL          ;add 1 to column value
        MOV     AH,2        ;function call to set cursor position
        MOV     BH,0        ;video page 0
        INT     10H
        CALL    DELAY

STEP10:                         ;---STEP10 restores the counter into CX
                                ;   from off the stack so that the LOOP
                                ;   statement works correctly.
        POP     CX          ;move stack value back into CX
        LOOP    STEP02      ;return to beginning of loop if CX <> 0

STEP11:                         ;---STEP11 is reached if the LOOP fails,
                                ;   ie. if CX is 0 and we've done our 20
                                ;   inputs.

        MOV     AX,4C00H    ;return control to DOS
        INT     21H

DELAY:
        MOV     CX,0FFFFH       ;set count to ffff
DELAY02:
        NOP                     ;no operation (increases delay)
        NOP
        NOP
        NOP
        LOOP    DELAY02         ;keep looping until CX=0 (causes long
                                ;delay)
        RET
MAIN    ENDP                ;end of MAIN procedure
CODE    ENDS                ;end of contents of CODE segment
        END     MAIN        ;END directive -- says where program begins.
```

E.1 OBJECTIVES

This lab should help you to understand how DOS directories are organize and maintained, how redirection of standard input and standard output operate, and how some DOS function calls are used. In this lab we will not be going into the Assembly Language interface very much; rather, we will concentrate on what high-level functions are required.

E.2 MATERIALS

You will need the PC Lab diskette.

E.3 INSTRUCTIONS

1. First we will see how DOS directories are structured. Load DOS on your PC (if you're loading a 3270 PC, it's best to make sure the Control Program doesn't load). Insert the lab diskette in drive A: and switch the default drive to A:. Type "DR". This command loads the Disk Repair program. How? Well, the DOS Command Interpreter's transient portion scans the directory entries for a file named "DR.COM". If it finds none, it looks for "DR.EXE". If it finds none, it looks for "DR.BAT". If it finds none, it looks in the next directory in the PATH. You know the rest. Fortunately, it found "DR.COM". (If it didn't, you don't have the right lab diskette.)

2. Voila the first screen of Disk Repair. Press a key to continue. You will come to a screen which explains some of the functions Disk Repair can perform. Look through them if you like; when you're done, select the Directory option by pressing <F7>. (Anything to be typed which we've indicated in <> brackets should be interpreted as an actual key -- function key, <ENTER> key and so on.)

3. A directory of your diskette appears, but Disk Repair tells you more than just the file name, extension, date and time. It also has an attribute byte to the right of each file, a cluster address for the first diskette cluster of the file, and a set of reserved bytes we won't worry about. The attribute byte is used to indicate a number of aspects of the file's status, as we saw in the DOS lecture. This byte is broken down as follows :

Bit:	Function:
1 (01H)	Read-only
2 (02H)	Hidden
3 (04H)	System file
4 (08H)	Volume label
5 (10H)	Subdirectory
6 (20H)	Archive bit
7	not meaningful
8	not meaningful

How do you interpret the actual byte you see on the display? Suppose the attribute byte of a file is 11H (the values on this screen are in hexadecimal). We can write 11H as 00010001B. We can consider the bits to be numbered 8 to 1 from left to right. In this case the byte indicates the following:

Bit:	Function:
1 set	Read-only
2 clear	NOT hidden
3 clear	NOT a system file
4 clear	NOT a volume label
5 set	Subdirectory
6 clear	Archived (it hasn't been changed since the last backup)
7	not meaningful
8	not meaningful

So an attribute of 11H means that the file is a directory and is marked read-only, which means you cannot remove the directory.

4. Look at the attribute for BIOSLAB.COM on your A: drive's directory. It should be 20. This means the file is a regular, read-write, unhidden, non-system file which has not been changed since it was last backed up. Don't worry if the attribute isn't 20. We're going to change it anyhow.

As with RDT, Disk Repair allows you to move the cursor around and change numerical values on the display. So move the cursor up to the attribute byte of BIOSLAB.COM and change the attribute so that the file is hidden. How do we do this? Bit 2 indicates a hidden file, so we type "02" into the file attribute byte.

Next we have to write this change out to diskette. Press <ENTER> so that your cursor returns to the command line, then type "W <ENTER>". Thi will write the change out to diskette. Then press "Q <ENTER>" to quit Disk Repair. We'll use it again later.

5. Do a directory of the diskette. Notice that BIOSLAB.COM is no longer there. However, if you type "BIOSLAB" and press <ENTER> the BIOSLAB program will load and you will have to type twenty characters before control is returned to DOS. 6. Load Disk Repair again and change the attribute of BIOSLAB.COM to non-hidden (attribute=0). Do a Write again (as directed above) and quit.

7. Next we will create some files in a new directory. Make a directory called "JUICY" (type "MD JUICY"). Change to that directory ("CD JUICY"). Now we will use some of the standard DOS handles to create a file. p.8. You should be familiar with the "COPY" command which copies one fi to another. "COPY" is an internal DOS command. What we will now do is copy from one file (the keyboard) to another file (the display). However, the device name for both keyboard and display is "CON" (for "console"). So type the following:

```
COPY CON CON
HELLO THERE ANNABEL
IT'S GREAT TO SEE YOU
<F6><ENTER>
```

You should see the two lines of text displayed on the screen, followed by "1 File(s) copied". What happened?

You copied from CON (the keyboard) to CON (the display). By pressing <F6> you generated a Ctrl-Z character (as you may have seen -- the "^Z" which was displayed indicated a Ctrl-Z). The Ctrl-Z is the end-of-file marker, so DOS knew that you had finished inputing from "CON". It then did the output operation which displayed the information on the screen.

9. Now create another file called "RAISINS.YUM". Type the following:

```
COPY CON RAISINS.YUM
GOOD MORNING GUSTAV
WHY DID YOU GET UP?
<F6><ENTER>
```

If you do a directory of drive A:, you should see a file called "RAISINS.YUM" listed, about 60 bytes long.

Next we will redirect from standard output to a file. We will type out a file using the DOS "type" command, but instead of displaying the file on the screen, we'll send it to disk, like this:

```
TYPE RAISINS.YUM > BANANAS.YUM
```

If you do a directory, you will see that a new file called "BANANAS.YUM" was created. Also, "RAISINS.YUM" did not really get typed on the screen, it was redirected into a file.

Do the above step again, only instead of using BANANAS.YUM as the second file name, use YUMMY.EGG. There is a reason for doing this -- don't ask, just wait.

In the root directory of your lab diskette is a file called "DEBUG.YUK". This file contains ASCII characters which will be used as input for the DEBUG program. You will have to find the debug program on the "C:" drive of your machine or off a DOS diskette, or ask an instructor for a copy. First of all, copy the "DEBUG.YUK" file into the "JUICY" directory. Then type the following (where <path> indicates the drive and path in which the DEBUG program can be found):

```
<path>DEBUG <debug.dat
```

12. What happens? You should see Debug loaded, and a screen full of information will appear. What has happened? Don't ask. All we did was use an input data file instead of the keyboard; the commands contained in that file went and created another file using facilities available for that purpose in DOS Debug. How it worked is irrelevant; what matters is that we actually used a file, instead of the keyboard, to tell Debug what to do.

13. Piping. You may need instructor help with this step if you aren't familiar with DOS. First, you must locate, on the "C" drive, two files: "SORT.EXE" and "FIND.EXE". These will likely be in either the root directory or in a directory called "DOS". Once you have found them, set your path up to point to the appropriate directory (for instance, you could type "PATH=C:\DOS" if the files were in the DOS directory on drive "C:"). Then, from the A: prompt (still in the "JUICY" directory), type the following line:

```
DIR | FIND "YUM" | SORT
```

What happens? You get an alphabetically sorted listing of all files in the JUICY directory which contain the characters "YUM" in either their extension or their file name. What happened?

1. "DIR" did a directory of the diskette's JUICY directory

2. The first "|" piped the results of the directory to another command, the "FIND" command

3. "FIND" used the directory listing as its input. The purpose of "FIND" is to find the given string (in this case "YUM") in the specified input, and to send any lines containing that string to the standard output. So "FIND" finds all files with "YUM" somewhere in their name.

4. The second "|" piped the standard output (the results of the find) to the SORT command.

5. The SORT command sorts those file names into alphabetical order, and displays them on the screen. We could have redirected the sorted list to a file as well.

Notice that this set of piped commands found all files with "YUM" in EITHER the file name OR the file extension. The DIR command has no way of doing this in one single command.

14. Subdirectories, we learned, are actually just files like any other file, but they contain listings of files instead of data or code. So we will use our juicy directory as a file. Go into Disk Repair. Look at the root directory; you will see the Juicy directory there. Change the attribute byte of the Juicy directory from 10 (directory) to 00 (read-write, unhidden file). Move to the "length" field for the JUICY entry and type in a length of 200 or more. Press <ENTER> to return to the command line,

then "W <ENTER>" to write the change out to disk. Now exit to DOS using the "Q" command.

15. Change to the root directory if you aren't already there. Do a DIR and you will see that JUICY is now a file, not a directory. Display its contents using the DIR command. You will see that it contains the names of the files we created earlier, as well as lots of strange codes which are in fact some of the attribute and reserved bytes for directory entries. We can see, then, that a directory other than the root directory is really just a file which DOS sets as a directory file; the file contains information on other files.

16. Enough of that -- go back to Disk Repair, and change the attribute of JUICY back from 00 to 10, so that it becomes a directory again. Use the Write command again to save the change; quit, do a directory, and make sure you've restored it as a directory and that it still contains the files you created. Now you can erase the files in it and remove the directory. Make sure you don't erase the root directory instead!

17. We will perform one last experiment, this time using batch files and the DOS terminate function call. You will find a file in your diskette's root directory called "ERROR.BAT". Display its contents by using the "TYPE" command. It should read as follows:

```
DR
  If errorlevel 3 goto A
  If errorlevel 4 goto B
  Echo No Error
  Goto Exit
:A
  Echo Error Three
  Goto Exit
:B
  Echo Error Four
:exit
```

What does this batch file do? First, it invokes Disk Repair. Then, when Disk Repair has finished, the batch file checks Disk Repair's exit code. If Disk Repair exited with exit code 3, the command at :A is processed ("Echo Error Three"); if the code is 4, the command at :B is processed, and if the code is neither of these, the batch file exits.

How does Disk Repair return a return code? Normally it doesn't, but we will make it do so.

18. Type "ERROR". The batch file will invoke Disk Repair. Now press <F10>. This brings you to the Interrupt window. In this window you can load registers and issue interrupts.

19. What we will do is issue a DOS interrupt to terminate Disk Repair We will use DOS Interrupt 21H, function call 4CH. To do this, move your cursor up to the hex codes to the right of the AX register (use the HOME

key). In AH (the first two characters of AX) type "4C". This sets AH to the 4C function call, "Terminate a process". In AL (the second two characters of AX), type either "03" or "04", for error code 3 or 4, whichever you like. Then press Enter to return to the command line.

20. Make sure that the "INT" command in the middle of the screen is set up for "INT 21". If the "INT" is set up for any other interrupt, change it to "21". Ask an instructor for help if needed.

21. Type "I" then <ENTER>. This will invoke INT 21, Function call 4CH Return code 03 or 04. Disk Repair will exit (we forced it to); the batch file should continue executing, and it should tell us which error code we specified in the DOS function call.

This concludes the DOS lab. You will notice that we did not use the Assembly Language interface to DOS (except with our final function call). This is because by now you should know enough about assembly language to figure out how it works. The DOS assembly language interface looks much like the BIOS one -- load a register with a function call, load others with parameters or variables or data, and issue the appropriate interrupt.

You may also have noticed an abundance of references to food in this lab. This is because it's 7:30 pm right now and I haven't yet had my dinner. Ah, the overtime joys of being a PC guru!

READER'S COMMENTS

Title: **IBM PC Internals Fundamentals Course notes** (GG24-3057-00)

You may use this form to communicate your comments about this publication, its organization or subject matter with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Comments:

Reply requested

Yes / No

Name : _____

Job Title : _____

Address : _____

_____

Cut or Fold Along Line

Reader's Comment Form

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

IBM International Technical Support Center
Department 91J, Building 235-2
901 Northwest 51st Street
Boca Raton, Florida 33432

Fold and tape | Please Do Not Staple | Fold and tape

IBM®

READER'S COMMENTS

Title: IBM PC Internals Fundamentals Course notes (GG24-3057-00)

You may use this form to communicate your comments about this publication, its organization or subject matter with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Comments:

Reply requested

Yes / No

Name : _____

Job Title : _____

Address : _____

_____

Cut or Fold Along Line

Reader's Comment Form

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

IBM International Technical Support Center
Department 91J, Building 235-2
901 Northwest 51st Street
Boca Raton, Florida 33432

Fold and tape | Please Do Not Staple | Fold and tape

IBM®

GG24-3057-00

IBM PC Internals Fundamentals Course Notes GG24-3057-00 PRINTED IN THE U.S.A.

IBM