Creating Utilities With Assembly Language

10 Best for the IBM PC & XT

- Undelete
- Font
- One Key
- Protect
- Dbug Scan
- Pcalc
- Clock
- Cleanup
- Diskwatch
- Notepad

teven Holzner

Brady

Creating Utilities with Assembly Language

10 Best for the IBM PC & XT

Steven Holzner

Brady Communications Company, Inc.
New York, New York 10020
A Simon&Schuster Publishing Company

Library of Congress Cataloging-in-Publication Data

Holzner, Steven.
Creating utilities with assembly language.

Includes index.
1. IBM Personal Computer—Programming.
2. IBM Personal Computer XT—Programming.
3. Utilities (Computer programs) I. Title.
QA6.8.I2594H65 1985 005.4′3 85-13266

ISBN 0-89303-584-X

Printed in the United States of America
86 87 88 89 90 91 92 93 94 95 96 1 2 3 4 5 6 7 8 9 10

Acknowledgments

The Author would especially like to thank Terry Anderson, Barbara Werner, and Patty Mahaffey for their tireless energies and unflagging devotion to producing work of the highest quality.

About the Author

Steven Holzner is currently a contributing editor at PC magazine, where his articles appear monthly. He got his first real taste for computing at MIT, where about half the student body is similarly inclined. He is inordinately fond of traveling, and has made it his business to do so in over thirty countries. He has spent a year each in Hong Kong and Hawaii, and spends most of his summers either in the Austrian Alps or on the western coast of Ireland. Besides keeping his sleeves rolled up, Mr. Holzner always keeps his collar unbuttoned.

Introduction

All you need to use this book is the program MASM.EXE and a word processor. Your DOS disk includes the other two needed programs, LINK.EXE and EXE2BIN.EXE, already. The idea behind this book is to ~~give~~ you a number of already working assembly language programs that will do as much as your IBM PC can do for you. These programs can be typed in with your word processor and run through MASM, LINK, and EXE2BIN to give you a COM file that can be run immediately. Everything we will do will be on the intermediate to advanced assembly language level, although we will review the basics.

Although this book was written with the Macro Assembler Version 1.00, the differences between it and version 2.00 are minor. Any necessary changes are given for each program listing, ready to be typed in.

Very often, computer books discuss all the individual commands of languages such as assembly language without using them in any solid applications. This approach is like learning to drive by studying an auto parts catalog. What we will do here is to develop complete programs of real merit on the intermediate assembly language level. We'll restore files , let two programs run at once, make our own keyboard "macros," redesign the PC's character set, and develop a notepad that will pop on the screen any time.

Many of the programs that we'll write are similar to ones available in the market for a good deal of money. The tools needed to build these same programs, however, are available to everyone. Owning a PC without using any of these tools at all is like buying a yacht for its air conditioning. The PC is a machine of solid potential and by adding some skills we'll see what it is really like and what it is capable of. As often as we can, we'll work to penetrate the mysteries of the PC to a core of gold. The PC is compact enough to permit us a real understanding of what is going on behind the scenes.

The examples in this book are built up progressively. While we add more and more to the program, we'll keep track of what came first to see how it all connects. Throughout the book, new program lines will be marked with arrows, and the text will discuss what has been marked. The entire listing is also available at the end of each section, ready to be typed in. The programs we will cover have been picked since they all provide short, bite-size examples. An entire book on a single program would be tough to work through. With smaller examples, the actual coding will get less in the way of the principles and techniques.

There is no better place to list the programs than right in the beginning, so here they are:

1. CLEANUP	"CLEANUP *.*" will skim through your entire diskette and ask you file by file which ones to delete.

2. UNDEL	UNDEL will restore files on double-sided floppy diskettes (not hard disks). If the file can be recovered, UNDEL will get it.
3. CLOCK	This program will be our first contact with memory-resident programs. It will continually display a digital clock display in the upper right corner of the screen, even if other programs are running.
4. ONEKEY	Using this package we can set up keyboard "macros," allowing us to replace a single key with an entire command. The F1 key, for instance, can become "DIR B:."
5. NPAD	NPAD is a reverse video notepad that pops on the screen whenever you need it (even if other programs are running). You can keep notes, error messages or memos here for future reference.
6. PROTECT#	Here we will modify the DOS DEL or ERASE command so that files can be securely protected from deletion but still be modified.
7. DBUGSCAN	This and the next program, PCALC, are on the optional diskette accompanying the book, and only the parts of interest are discussed in the book. DBUGSCAN is a much-needed utility that runs with DEBUG, displaying the value of your program's variables while you step through the code.
8. PCALC	PCALC is a calculator that pops on the screen for use anytime. It includes Hex, Logical operators, a memory, a help screen, and a reverse video display.
9. DSKWATCH	This watchdog program will monitor the diskette drives for errors while other programs are running. Diskette drives are susceptible to failure because of their complexity, and DSKWATCH will tell you beforehand if errors are starting to mount up.
10. FONT	FONT will allow you to redesign the PC's character set (graphics screens only). It includes an editor for your characters and the program that will install them in the PC.

Although two of these programs, PCALC and DBUGSCAN, are only on the disk, we will cover all their important parts in this book.

These programs are arranged to build up knowledge progressively. What we learn in one is designed to be applied in the next. If you are just learning the techniques of intermediate or advanced assembly language, moving from one to the next along this list will flesh out your expertise. If, on the other hand, you already have experience with this level of programming, you might want to read each section independently.

For many people, the operation of the PC is a complete mystery. Their expertise consists mostly of pushing a set of keys and hoping for the best. The power that lives under that placid grey cover is out of their reach. Our object here is to dissolve the mystery and to allow you to accomplish things as powerfully as DOS, on as advanced a level as DOS can reach. None of this has to be difficult. It cannot be done, however, by presenting a disorganized set of assembly language commands. We will have to see everything constructed from beginning to end. There's no better way to learn than by example, and we'll spend our time dissecting some high-power ones.

The Possibility of Bugs

All of these programs have been extensively tested, both by the author and by reviewers. Every effort has been made to make sure that the code is bug-free. It is possible, though, that errors have slipped through. If you find something that you're sure can't be right, please write in immediately to me, Steven Holzner, c/o Brady Communications. Some of these programs have appeared in *PC magazine* already (without any programming). Two hundred fifty thousand of the best proofreaders anyone could ask for can't all be wrong. I'd also like to thank the hundreds of readers who have written in about the articles and my column.

All of these programs were developed on a standard IBM PC with two diskette drives, 256K of memory, a monochrome screen, and DOS 2.10 (the programs will work under any DOS version). Of course, if I could afford it, I'd be using an AT. The word processor used was PC-Write, still the best bargain in word processors.

Contents

SECTION 1

CLEANUP

Chapter 1 Assembly Language Remembered

If all the pieces fit together as we might hope, this book will be like few others for the IBM PC. Our intention is to develop the most exciting capabilites of the PC and get them working for us. Our goal is to use all of the sophisticated inner resources of our computer to push us from the mundane to the exceptional with occasional flings into sheer extravagance.

Perhaps you've read a book about what's inside the PC and now want to put the pieces together. Perhaps you'd like to explore the cloudy region where the real power lies in the PC, flexing some hidden muscles. Maybe you've wondered what the PC can really do or what you can make it do. If you've wondered how two programs can run at the same time, or how to put a notepad on the screen, or how to restore deleted files, then we're on the same track.

We're going to build our programs on the same backbone that professional software developers use, which means using assembly language. To read this book you ought really to have some knowledge of assembly language—if not an intimate knowledge at least a passing acquaintance.

This first chapter is going to give us an accelerated review of the subject, but it would make a skinny textbook. We've concentrated a great deal into these few pages and so if you're lost by the time you've finished this chapter, then you should spend some time getting to know assembly language. On the other hand, if you know what JGE means, then you already know enough to skip this chapter entirely.

The choice of assembly language is not made whimsically. For most of what we're going to do—running two programs at once, putting a calculator on the screen while another program is running, restoring deleted files—there is no alternative to exploiting the sheer but simple power of assembly language. We will discuss virtually everything new in some detail before using it. After all, there is no better way to learn than by example and we'll try to fill the book with some strong ones.

Keep in mind, though, that this book is not intended to teach assembly language. Assembler is the tool that we'll use to probe for what we want. What we're after is a solid knowledge of what's interesting to use inside the PC. It's good to have heard about the video controller, but the real interest lies in its application. The same thing may be said for the directory or the keyboard buffer. The use of everything inside the IBM PC is the real substance of this book. So that we do not, however, start on an ill-prepared expedition, let's spend a few moments sharpening our tools in this speedy and not necessarily exhaustive review:

The Basics

The essentials of the IBM PC are more or less these; it is a 16-bit machine with many clever components, among which is a sophisticated 8088 microprocessor. This processor includes four general registers which play the part of variables in our programs. They are named AX, BX, CX, and DX. Each of these 16-bit registers in turn may be broken into two 8 bit (one byte) registers. For instance, AX can be thought of as being made up of AH (its high byte) and AL (its matching low byte). Most commands can be used with either full registers or half registers (for instance, one may say either ADD BX,5 or ADD BL,5).

The IBM PC uses ASCII. Each ASCII character, like "w" or "A", is converted into an ASCII code of 8 bits and so each character takes up one byte in memory.

We'll frequently use hexadecimal numbers. Their attraction is that one hexadecimal digit represents four bits. We can convientiently express the contents of a 16 bit register as a four digit hex number. For example, AX can hold the numbers 1234H, ABCDH or 8B4CH but not 12345H.

The 8088 also uses segmentation, an unusual method of addressing memory. Each address or location in memory is pointed to by two 16 bit numbers, the segment address and the offset address.

A segmented address is written like this: 1234:5678, always in Hex. Here the segment address is 1234H and the offset address from the beginning of the segment is 5678H. The real address in memory (we'll always use segmented addresses) is found by multiplying the segment address by 16 (10H) and adding the offset address. Here that gives us the 20 bit address 179B8H. An address of 1234:5678, therefore, is the same as a distance from the beginning of memory of 179B8H bytes.

An address such as AAAA:BBBB may be thought of as a distance or offset of BBBBH bytes into a segment starting at AAAA0H. These segments may be started at any multiple of 10H bytes in memory. Using segmentation, the PC can express 20-bit addresses in two 16-bit words.

Since you have to be able to reach everywhere you want with a segmented address, special segment registers have been introduced. One of these is the code segment register. The PC will find the next command to execute using the segmented address CS:IP, using the numbers in the code segment register CS and the instruction pointer register IP to form the command's address. DOS takes care of these addresses for you when it loads your program and the 8088 is in charge of them when the program is running.

Also of importance is the data segment register, DS, which is usually more under our control than CS. After we've decided where we want to get data from, we'll set DS and then only have to worry about the offset address inside the data segment (like DS:1111 or DS:EEEE).

Talking our way through segments provides a rather substandard introduction to them. The best way to learn about segment registers is to see them in action, and we'll put them to work as soon as we can.

The Heart of Assembly Language

A small group of commands make up the real core of assembly language. Here is a brief review of each:

MOV

MOV is the very heart of assembly language. If you've understood MOV it may be said that you're on your way. Some MOVs:

```
MOV   BX,5CH
MOV   AX,[BX]
MOV   AX,CX
```

The first command moves 5CH into the register BX. The next command uses the terrific ability of BX to act as an index. This is one of the capabilities that have set the 8088 and like microprocessors apart from the previous generation, and it will be important for us. If the 16 bits you want to pluck from memory start at address 1234:5678, then DS must be 1234H and BX must be set to 5678H. When BX is put into brackets, MOV AX,[BX] will move that word into AX. The PC will assume you're using the data segment since this method of addressing is used for data, not commands in a program. MOV AX,[BX] is really shorthand for MOV AX,DS:[BX]. This is called indirect addressing. What makes it so flexible is that by continually incrementing BX you can scan up or down a whole line of data. The utility of indirect addressing is almost inexhaustible, as we will see.

MOV AX,CX, as you probably know, simply moves whatever number is in CX into AX, leaving the number in CX unchanged.

Intel's 8088 stores words in memory in a maddening fashion, with the low byte first and the high byte last (that is, with the higher byte higher in memory). 1234H, for example, would be stored as 34H 12H. If the hex number 1234H is in AX and you MOV [BX],AX then 34H (the lower byte) will be stored at address DS:[BX] and 12H at DS:[BX+1] so that 1234H will appear in memory as 34H 12H.

Conversely, MOV AX,[BX] will restore AX to 1234H. This method of storage may try your patience for a while but with practice it will quickly become familiar.

CMP

CMP is usually used in conjunction with jump commands so we'll now introduce the JE command—Jump if Equal—for the purposes of illustration. Jumps need to be supplied with some label to jump to in your code and in this example we'll be jumping to a line labelled END. Here are some CMPs and JEs:

```
    CMP AX,5               CMP AX,[BX]            CMP AX,BX
    JE END                 JE END                 JE END
    :                      :                      :
    [Some Commands]        [Some Commands]        [Some other Commands]
    :                      :                      :
END:[Exit]             END:[Exit]             END:[Exit]
```

In the first jump example above, the contents of AX are compared with 5 and, if AX does hold 5, JE END will cause us to jump to the line labelled END. If AX was not equal to 5 then the program would simply continue with the next line.

CMP compares by actually subtracting the second number from the first without changing either. It does, however, set the PC's internal flags. Every command with some numerical result (ADD, CMP, SUB, and so on) sets some of the flags. We will use none of these flags directly. We will usually use CMP to set the flags and then use a jump which will first check how the flags are set and then jump accordingly. Although other commands are occasionally used, CMP is most often used to set the signals for a following jump. Here's a list of the PC's flags (for future reference only):

Flag	Meaning	Where (If) Used
Carry	= 1 if an operation left a carry	UNDEL
Parity	= 1 if result has a even # of 1s	[Communication Programs]
Auxillary Carry	Used with Binary Coded Decimal	[I/O Programs]
Zero	= 1 if the result is zero	CLEANUP
Sign	= 1 if result is negative (Sign bit = 1)	PCALC
Trap	Used in single-stepping programs	[Debugging Programs]
Interrupt Enable	= 1 to allow hardware interrupts	CLOCK
Direction	Sets direction in string searches	[All kinds of Programs]
Overflow	Result exceeds signed number range	PCALC

All of these flags are stored in a flag register. The only contact we'll have with this flag register is pushing or retrieving it onto or from the stack.

In the second jump example above, AX is compared with the data word at the address DS:BX. If DS = 1111 and BX = 2222, we will compare the number in the AX register to the value at address 1111:2222.

```
    CMP AX,[BX]
    JE END
    :
    [Commands]
    :
END:[Exit]
```

If the two are equal we will jump in the next step to the line labelled END. If they are not, program execution will continue with the command after JE END.

In the third jump example the number in AX is compared with the number in BX. If the two are equal, we will jump to END.

The Jumps

There are many jumps that the IBM PC, always a versatile machine, can make. The small army (don't memorize them) that we'll use includes:

```
                    JMP      JE      JNE      JZ      JNZ
                        JL      JG      JLE      JGE
                            JA      JB      JC
```

The mnemonics on the first line mean Jump (unconditional—always jump), Jump if Equal, Jump if Not Equal, Jump if Zero, Jump if Not Zero. We'll see these in use shortly.

The next line means Jump if Less than, Jump if Greater than, Jump if Less than or Equal to, Jump if Greater than or Equal to. These jumps are used if you are comparing two signed numbers—that is, they respect the sign bit (the highest bit in a number) and treat negative numbers as less than positive ones. The sign bit and signed numbers will be reviewed in the section covering our calculator, PCALC.

The last line has the commands for Jump if Above, Jump if Below, and Jump if Carry flag set. These jumps are made by comparing the unsigned magnitude of two numbers. A 1 in the highest bit position merely means a big number as far as these jumps are concerned, not a negative one. We won't have to worry much about the difference between Jump if Above and Jump if Greater because we'll rarely use signed numbers.

Each mnemonic usually offers enough letters to figure out the jump's meaning. To use a jump you must label part of your code so that you can jump to it when the time comes. Labels in this book are done like this:

```
          CMP   AX,5
          JE    OK
          MOV   AX,5
--> OK:   MOV   BX,CX
```

If AX is equal to 5 then when the program reaches JE OK, it will jump immediately to the line labelled OK instead of continuing on to execute the line MOV AX,5, which means that here we will end up with 5 in AX no matter what.

From Fundamental to Fancy

If you've mastered the above core of commands you've got a credible foothold in assembly language. Despite its humble reputation, assembly language can include some fancy commands. We'll review an additional set that will tighten our grip on programming.

INC

A small but useful and fast command, *INC* increments. For instance:

```
INC BX
```

increases the number in BX by one. DEC is the antonym of INC.