




APPLE[®]



ROOTS



ASSEMBLY LANGUAGE PROGRAMMING



FOR APPLE[®] IIe AND APPLE[®] IIc



MARK ANDREWS

Apple[®] Roots:

Assembly Language Programming For the Apple[®] IIe & IIc

Mark Andrews

Osborne McGraw-Hill
Berkeley, California

Published by
Osborne **McGraw-Hill**
2600 Tenth Street
Berkeley, California 94710
U.S.A.

For information on translations and book distributors
outside of the U.S.A., please write to Osborne **McGraw-Hill**
at the above address.

Apple is a registered trademark of Apple Computer, Inc.
A complete list of trademarks appears on page 345.

**Apple® Roots:
Assembly Language Programming
For the Apple® IIe & IIc**

Copyright © 1986 by McGraw-Hill, Inc. All rights reserved. Printed in the United States of America. Except as permitted under the Copyright Act of 1975, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

1234567890 DODO 898765

ISBN 0-07-881130-9

Jonathan Erickson, Acquisitions Editor
Paul Jensen, Technical Editor
Michael Fischer, Technical Reviewer
Jessica Bernard, Copy Editor
Judy Wohlfrom, Text Design
Yashi Okita, Cover Design

Introduction

If your Apple doesn't understand you, maybe it's because you don't speak its language. Now we're going to break that language barrier.

This book will teach you how to write programs in assembly language—the fastest-running and most memory-efficient of all programming languages. It will give you a working knowledge of machine language, your computer's native tongue. It will enable you to create programs that would be impossible to write in BASIC or other less advanced languages. It also will prove to you that programming in assembly language is not nearly as difficult as you may think.

Many books have been written about assembly language, but this is the first assembly-language book to deal specifically with the Apple IIc and the Apple IIe, the two newest computers in the Apple II line. It is also the first book that explains how to write assembly-language programs using ProDOS, the Apple IIc/Apple IIe disk operating system that has now replaced its predecessor, DOS 3.3. The book also covers the advanced features of the 65C02 microprocessor, the new chip built into the Apple IIc that can also be installed in the Apple IIe.

In addition, this is the first assembly-language book that explains how to use three of the most popular assemblers for the Apple IIc and the Apple IIe: the *ProDOS Assembler Tools* package from Apple, the *Merlin Pro* assembler-editor system from Roger Wagner Publishing, Inc., and the ORCA/M assembler from The Byte Works of Albuquerque, New Mexico.

The Apple IIc and the Apple IIe offer a number of brand-new features that are of great importance to Apple programmers and potential Apple programmers. These features include an 80-column text display, double high-resolution graphics, and 64K of extra memory (all built into the Apple IIc and optional on the Apple IIe). Both the IIc and the IIe have expanded keyboards, including new function keys (OPEN APPLE and CLOSED APPLE keys) and new special-character keys. In addition, the Apple IIc has a built-in set of special characters designed for use with a mouse, and the same special characters are available on any Apple IIe equipped with plug-in mouse cards.

Both the Apple IIc and IIe are now being shipped with ProDOS, the new Apple II disk operating system that succeeds DOS 3.3. ProDOS is not just another revision of DOS 3.3; it is a completely new disk operating system that was designed specifically for the Apple IIc, the Apple IIe, and future computers in the Apple II line. ProDOS handles disk files and disk drives very differently from the way they were handled under DOS 3.3.

A point-by-point comparison between DOS 3.3 and ProDOS is beyond the scope of this introduction. However, please note that *there are so many differences between ProDOS and the systems it replaces that most assembly-language programs written under earlier disk operating systems will not work in a ProDOS environment.* This is the first book about writing assembly-language programs for the new ProDOS-equipped Apple IIc and Apple IIe computers.

Both the Apple IIc and the newest versions of the Apple IIe are now equipped with an advanced 8-bit microprocessor called the 65C02. The 65C02, a new member of the 6502 series of microprocessors, is designed to be programmed in standard 6502 assembly language. However, the 65C02 contains a number of new features. Along with the 56 instructions used in conventional 6502 assembly language, the 65C02 is equipped with several additional instructions. It also recognizes a number of addressing modes that were not available in earlier 6502-series microprocessors.

If you know BASIC—even a little BASIC—you can learn to program in assembly language. Once you know assembly language, you'll be able to

- Write programs that will run 10 to 1000 times faster than programs written in BASIC.
- Use up to 16 colors simultaneously in any Apple IIc or Apple IIe graphics mode—including double high-resolution graphics.

- Custom design your own screen displays, mixing text and graphics in any way you like.
- Create your own customized character sets.

Knowing assembly language can also enable you to create music and sound effects for Apple IIc/Apple IIe programs, write programs that will boot from a disk and run automatically when you turn your computer on, and intermix BASIC and assembly language in the same program, combining the simplicity of BASIC with the speed and versatility of assembly language.

In other words, once you learn how to program in assembly language, you will be able to start writing programs using the same techniques that professional programmers use. Many of those techniques are impossible without a knowledge of assembly language.

Finally, as you learn assembly language, you will be learning what makes computers tick. That will make you a better programmer in any language.

While teaching you assembly language, *Apple Roots* will provide you with a comprehensive collection of assembly-language routines that can be typed and assembled and then used in user-written assembly-language programs. It also contains a library of interactive tutorial programs that are especially designed to take the drudgery out of learning assembly language.

Chapter 1 is an easy-to-understand introduction to assembly language. The main feature of Chapter 2 is a clear explanation of binary numbers. In addition, Chapter 2 contains a series of four type-and-run BASIC programs that can convert numbers from one base to another.

In Chapter 3 you will learn about the 6502B/65C02 chip used in the Apple IIc and the Apple IIe. In Chapter 4, you'll start actually writing assembly-language programs. The rest of the book presents a number of advanced programming lessons and type-and-run assembly-language programs.

The first thing you need in order to use this book is an Apple IIc or Apple IIe computer equipped with a TV set or a computer monitor (preferably a color model) and at least one disk drive. A line printer is highly recommended but not essential. A game controller, a mouse, or both will also come in handy. So will a second disk drive.

The assembly-language programs in this book were written using three assemblers: the Apple ProDOS assembler, the Merlin Pro, and the ORCA/M. If you don't own one of those packages, it would be a good idea to buy one before starting this book. All of the programs in the

book were also written under ProDOS. If your Apple IIe was purchased before ProDOS was introduced, you will need to buy a ProDOS package from your Apple dealer and learn to use it.

One prerequisite for using the assembly-language lessons in this book is a basic understanding of ProDOS, which you can gain by reading a ProDOS manual. You should also have at least a fundamental knowledge of Applesoft BASIC or some other high-level programming language.

There are at least two other books that you should have before you start studying assembly language. The first of these books is the user's manual that came with your computer. The second is a reference manual for your computer. (Apple publishes separate reference manuals for the Apple IIc and the Apple IIe.) Other useful books include *Programming the 6502* by Rodnay Zaks, *Assembly Lines: The Book* by Roger Wagner, and *6502 Assembly Language Programming* by Lance A. Leventhal. These books, and others that may come in handy while you're studying assembly language, are listed in the Bibliography.

Contents

	Introduction	ix
1	Breaking the Assembly Language Barrier	1
2	Number Systems	19
3	In the Chips	33
4	Writing and Assembling an Assembly- Language Program	53
5	Running an Assembly-Language Program	85
6	The 6502B/65C02 Instruction Set	105
7	Addressing Your Apple	135
8	Looping and Branching	157
9	Single-Bit Manipulations of Binary Numbers	179
10	Assembly-Language Math	193
11	Memory Magic	211
12	Fundamentals of Apple IIc/IIe Graphics	233

13	Game Paddles, Joysticks, and the Apple Mouse	247
14	Apple Graphics	269
A	Assembly-Language to Machine-Language Conversion Chart	309
B	Machine-Language to Assembly-Language Conversion Chart	317
C	The 65C02 Instruction Set	323
D	65C02 Op Code Table	325
E	65C02 Addressing Modes	327
F	The 65802/65816 Instruction Set	329
G	65816 Addressing Modes	331
H	65816 Op Code Table	333
I	The ASCII Character Set for the Apple II	335
	Bibliography	343
	Index	347

1

Breaking the Assembly Language Barrier

If you want to learn *assembly language*, you've opened the right book. With this volume and an Apple IIc or Apple IIe computer, you can start programming right now in *machine language*. Then we'll see how machine language relates to assembly language. Turn on your computer and type HI.TEST.BAS, the BASIC program listed in Program 1-1. Then run the program, and you'll immediately see how it got its name.

Program 1-1
HI.TEST.BAS
The HI.TEST Program (BASIC Version)

```
10 REM *** HI.TEST.BAS ***
20 DATA 169,200,32,237,253,169,201,32,237,253,96
30 FOR L = 32768 TO 32778: READ X: POKE L,X: NEXT L
40 CALL 32768
```

Machine Language and Assembly Language

As you can see, Program 1-1 is written in BASIC. However, when you type the program and execute it, your computer will run a machine-language program.

Please note that this is machine language, not assembly language. As you'll see later in this chapter, machine language and assembly language are very closely related, but they are not exactly the same. Machine language is made up of numbers—nothing but numbers. Since “number-crunching” is what computers do best, machine language is ideal for a computer. In fact, machine language is the only language that a computer actually understands. No matter what language a program is originally written in, it must be converted into machine language before a computer can process it.

The main reason that assembly language is different from machine language is that it was designed for humans, not for machines. From the standpoint of both structure and vocabulary, assembly language is very similar to machine language. In fact, assembly language is not actually a programming language at all, but merely a notation system designed to make it easier to write programs in machine language.

Despite its structural similarity to machine language, however, a program written in assembly language looks quite different from a program written in machine language. Whereas machine language consists solely of numbers, assembly language uses three-letter abbreviations called *mnemonics*. It's therefore easier to write programs in assembly language than in machine language.

In one respect, though, assembly language is just like any other programming language: before an assembly-language program can be executed by a computer, it must be converted into machine language. For this reason, programs written in assembly language are often called *source-code* programs. And machine-language programs generated from source-code programs are often referred to as *object-code* programs.

Source-code programs are usually written with the aid of a special kind of software package called an *assembler-editor*, or simply an *assembler*. An assembler-editor package usually includes at least two kinds of utility programs: an assembly-language *editor*, which enables the user to write programs in assembly language, and an *assembler*, which can convert (or *assemble*) assembly-language programs into machine language.

Assembly language and machine language will be discussed in more detail later in this chapter.

How the HI.TEST.BAS Program Works

Now we're ready to take a closer look at the HI.TEST.BAS program shown in Program 1-1. The HI.TEST.BAS program begins with a title line. The next line in the program, line 20, is a line of data that equates to a series of machine-language instructions. Line 30 contains a loop that pokes the machine-language data in line 20 into a block of RAM (which we will define shortly) that extends from memory address 32768 to memory address 32778, or \$8000 to \$800A in hexadecimal notation. (A memory address—sometimes referred to as a *memory location* or *memory register*—is nothing but a number that can be used to pinpoint the location of any piece of data, or *byte*, stored in a computer's memory. There are 65,535 memory addresses in an off-the-rack Apple IIe, and there are 131,070 memory addresses in an Apple IIc or an Apple IIe equipped with an expanded 80-column card. More information on memory addresses will be provided later in this book, primarily in Chapter 11, which will focus specifically on the memory structure of the Apple IIc and the Apple IIe.) Finally, in line 40, there's a CALL instruction that executes the machine-language program that has just been loaded into memory.

To understand what your computer does when it receives the CALL instruction in line 40, it will help to have a basic understanding of the architecture of microcomputers and how your Apple processes a machine-language program.

Inside a Microcomputer

Every microcomputer can be divided into three parts:

- *A central processing unit (CPU).* A central processing unit, as its name implies, is the central component in a computer system, the component in which all computing functions take place. All of the functions of a central processing unit are contained in a *micro-processor unit* (or *MPU*). Your Apple computer's MPU—as well as its CPU—is a *large-scaled integrated circuit (LSI)* (a 6502B chip if you own an Apple IIe, and a 65C02 chip if you own an Apple IIc).

- A *memory*. Memory can be further divided into *RAM* (*random-access memory*) and *ROM* (*read-only memory*). These two types of memory are discussed in the following section.
- Some *input/output (I/O)* devices. Your computer's main input device is its *keyboard*. Its main output device is its *video monitor*. Other devices that your Apple can be connected to—or, in computer jargon, can be *interfaced* with—include telephone modems, graphics tablets, printers, and disk drives.

Figure 1-1 is a block diagram that illustrates the architecture of the Apple IIc and the Apple IIe. In this chapter we will not concern ourselves with the I/O. However, keyboard and screen I/O will be covered later, beginning with Chapter 8.

Your Apple's Memory

Figure 1-1 shows the two kinds of memory a computer has: random-access memory (RAM) and read-only memory (ROM). The important difference between them is that RAM can be modified, while ROM cannot. ROM is permanently etched into a bank of memory chips inside your Apple, so it's always there, whether the power to your computer is off or on. Every time you turn off your Apple, everything that you've

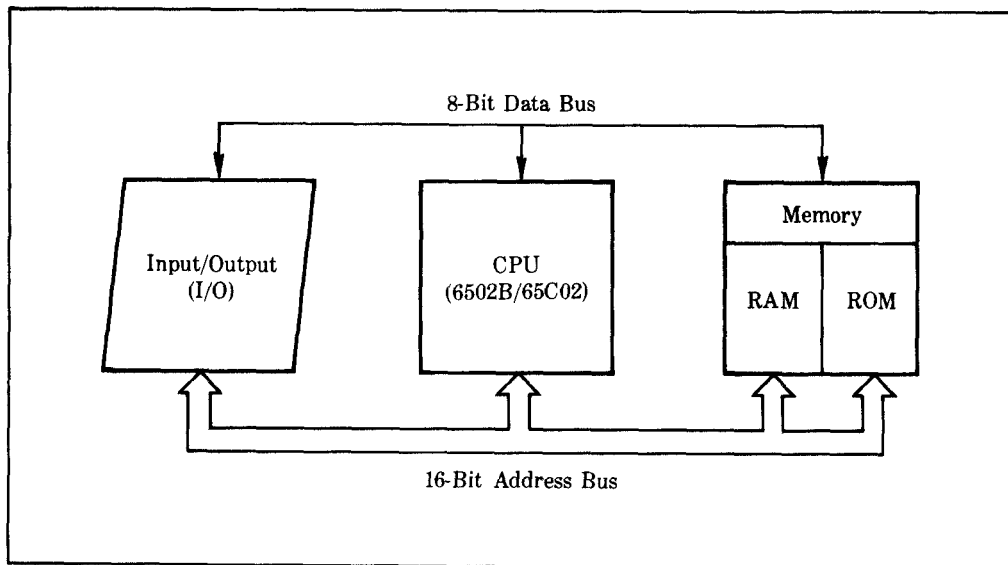


Figure 1-1. Block diagram of a microcomputer

stored in RAM immediately disappears. But everything in ROM remains and will spring back into action when you turn your computer on again.

The largest block of ROM in your Apple extends from memory address 53248 (\$D000 in hexadecimal notation) to memory address 65535 (\$FFFF in hexadecimal notation). A number of important programs are permanently situated in this block of ROM, including your computer's BASIC interpreter and its built-in machine-language monitor.

Machine-Language Programs in RAM In introductory books about computers, a bank of RAM is often compared to a bank of mailboxes built into a wall in a postal station. Each memory address in a RAM bank, like each mailbox in a tier of post office boxes, has an identifying number. And a computer program (like an ideal employee in a post office) can get to any of the memory addresses in a bank of RAM with equal ease. In other words, information stored in RAM can be retrieved at random. That's why RAM is called *random-access memory*.

What happens when your computer processes a machine-language program? Every machine-language program is made up of a series of numbers. When a machine-language program is loaded into a computer's memory, the numbers that make up the program are stored in a series of addresses in RAM. The starting address of the memory block in which the program is stored (known as the program's *origin* address) is usually stored in a special, predetermined memory location. Thus, when it is time to run the program, its starting address can be easily located.

Once a machine-language program has been loaded into RAM and its origin address has been stored in an accessible memory location, the program can be executed in several ways. For example, a machine-language program stored in an Apple II computer can be executed using a CALL instruction, a USR(X) instruction, a ProDOS dash (—) command, or a ProDOS BRUN command. These and other methods for running machine-language programs will be explained in Chapter 5.

Processing Executable Code When your computer goes to a memory location that has been identified as the starting address of a program, it should find the beginning of a block of executable code—that is, the beginning of a machine-language program. If it finds a program, it will carry out the first instruction in that program and then move on to the next consecutive address in its memory.

Your computer will keep repeating this process until it either

reaches the end of a program or encounters an instruction telling it to jump to another address.

Your Apple's CPU

In a microcomputer, a central processing unit (CPU) usually consists of a single microprocessor chip. Apple IIc and Apple IIe computers use either the 6502B chip or the 65C02.

The 6502B chip was designed for the Apple IIe and was originally built into all Apple IIe's. The 65C02 was designed for the Apple IIc and is the only microprocessor that has ever been used in the IIc. The 65C02 is now being built into all new Apple IIe's and is available as an optional, user-installable upgrade to older IIe's.

Both the 6502B chip and the 65C02 chip are improved and updated versions of an earlier chip, the 6502, developed by MOS Technology, Inc. The 6502 and chips based on it are used not only in Apple computers, but also in personal computers manufactured by Atari, Commodore, and several other companies.

The 6502B chip used in the Apple IIe is really just a faster-running version of the original 6502. But the 65C02 that's built into the Apple IIc and newer Apple IIe's has some extra capabilities that the old 6502 didn't have. In addition to being faster than the 6502, it uses less power, and it recognizes a number of instructions that the 6502 didn't understand. The 65C02 also has some additional addressing modes, a feature that will be explained in a later chapter.

For most purposes, however, the similarities among the 6502B, the 65C02, and the other chips in the 6502 family are more important than their differences. Although the model numbers of 6502-series chips may sometimes get confusing, all of the chips in the 6502 family are designed to be programmed using the same assembly-language dialect generically known as 6502 assembly language. Once you learn how to write programs in 6502/65C02 assembly language, you'll be able to program many different kinds of personal computers in addition to your Apple, including many manufactured by Atari and Commodore.

Even more important, the *principles* used in Apple assembly-language programming are universal; they're the same principles that all assembly-language programmers use, no matter what kinds of computers they're writing programs for. Once you learn 6502/65C02 assembly language, therefore, you can easily learn to program other kinds of chips, such as the Z80 chip used in Radio Shack and CP/M-based computers, and even the powerful newer chips used in 16-bit and 32-bit microcomputers such as the Apple Macintosh and the IBM PC.

Compilers, Interpreters, and Assemblers

Now that you have a basic understanding of what your Apple is made of and how it works, we're ready to take a closer look at the relationship among the three categories of computer languages: machine language, assembly language, and high-level languages.

High-level languages did not get their name because they're particularly esoteric or profound. They're called high-level languages merely because they're several levels removed from machine language, your computer's native tongue.

There are hundreds, perhaps thousands, of high-level languages, but most of them have at least one feature in common: they all bear at least a passing resemblance to English. BASIC, for example, is made up almost completely of instructions—such as PRINT, LIST, LOAD, SAVE, GOTO, and RETURN—that are derived from English words. Most other high-level languages also have instruction sets based largely upon plain-language words and phrases.

But computers can't understand a word of English; the only language they can understand is machine language, which is composed only of numbers. For this reason, a program written in any other language has to be translated into machine language before a computer can understand it. As mentioned previously, people who write programs in assembly language usually use special software products called assemblers to convert assembly language programs to machine language. Similarly, people who program in high-level languages use special kinds of software packages called *interpreters* and *compilers* to help them translate the programs they have written into machine language.

Interpreters and Compilers

The most important difference between interpreters and compilers is that an interpreter is designed to convert a program into machine language every time the program is run, while a compiler is designed to convert a program into machine language only once. When you write a program using an interpreter, you can store the program on a disk in its original form; your interpreter will automatically convert it into machine code every time you run it. But when a program is written using a compiler, it has to be converted into machine language and then stored on a disk as a machine-language program. Then it can be run like any other machine-language program, without any further need for a compiler.

Interpreters are easier to use than compilers because they're designed to be "transparent" to the user; that is, they are so unobtrusive that you're not even aware they're there. The BASIC utility that's built into your Apple IIc or Apple IIe is an interpreter, and using it will show you how transparent a BASIC interpreter can be. When you write a program in Applesoft BASIC, your computer's built-in interpreter translates every line of code that you write, as you write it, into a special kind of language called a *tokenized* language. Then, each time the program is run, it is translated into machine language.

This is a very roundabout way to run a program, and it's one factor that makes BASIC a rather slow-running language. But the process does work quite smoothly; if you've ever run an Applesoft BASIC program, you probably never even noticed the process of BASIC-to-machine-language translation.

One advantage of interpreters over compilers is that they can check each line of a program for obvious errors as soon as the line is written. If they don't check each line, they usually do spot errors as soon as the program is run. The errors can then be fixed on the spot. Compilers are less interactive. Most compilers can't check a program for errors until the program has been compiled. After an error is found and fixed, the program must be compiled again.

Compilers do have one significant advantage over interpreters: they produce faster-running programs. When a program is written using an interpreter, it has to be processed through the interpreter every time it's run. But a compiler has to do its job just once and never has to be used when a program is actually running.

Assemblers and Assembly Language

Assembly language, as we have seen, is neither an interpreted language nor a compiled language. Converting an assembly-language program into machine language requires an assembler-editor (also referred to as an assembler).

Because of the close relationship between machine language and assembly language, an assembler does not have nearly as difficult a task as an interpreter or a compiler. Each time an interpreter or compiler converts an instruction into machine language, a block—sometimes a very large block—of machine code has to be generated. But an assembler has to translate only one instruction at a time. The instructions used in assembly language perform much simpler functions than the instructions used in most high-level languages, so source-code programs written in assembly language tend to be much longer than similar programs