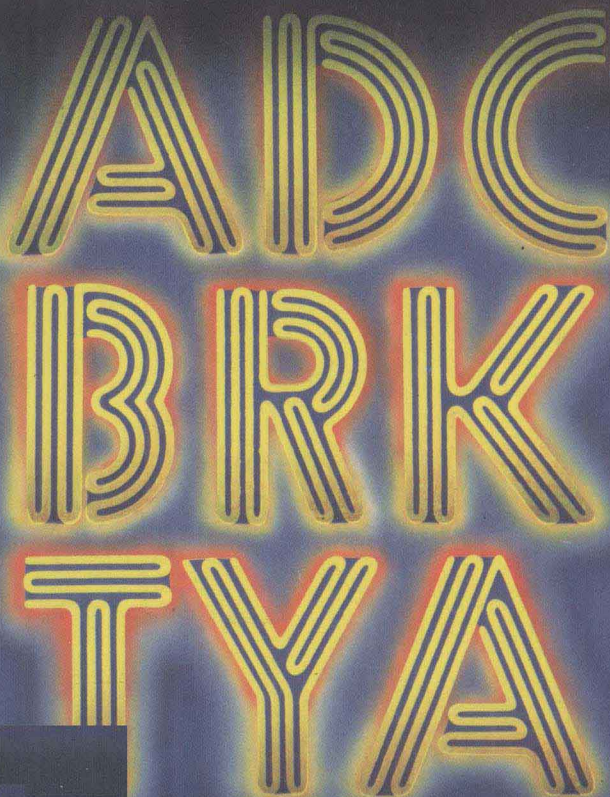


ASSEMBLY LANGUAGE PROGRAMMING FOR THE ATARI COMPUTERS

MARK CHASIN



ADC
BRK
TYA

The image features three lines of stylized, glowing text representing assembly language instructions: 'ADC', 'BRK', and 'TYA'. The letters are composed of multiple parallel lines, giving them a 3D, neon-like appearance. They are set against a dark blue background with a subtle gradient. In the bottom left corner, there is a dark, pixelated graphic element consisting of several overlapping rectangular blocks of varying shades of blue and black.

ASSEMBLY LANGUAGE PROGRAMMING FOR THE ATARI COMPUTERS

MARK CHASIN

McGraw-Hill Book Company

New York St. Louis San Francisco Auckland
Bogotá Guatemala Hamburg Johannesburg
Lisbon London Madrid Mexico Montreal
New Delhi Panama Paris San Juan
São Paulo Singapore Sydney Tokyo Toronto

The author of the programs provided with this book has carefully reviewed them to ensure their performance in accordance with the specifications described in the book. Neither the author nor McGraw-Hill, however, makes any warranties whatever concerning the programs. They assume no responsibility or liability of any kind for errors in the programs or for the consequences of any such errors.

ATARI is a registered trademark of Atari, Inc., Sunnyvale, CA.

ASSEMBLY LANGUAGE PROGRAMMING FOR THE ATARI COMPUTERS

Copyright © 1984 by Mark Chasin. All rights reserved. Printed in the United States of America. Except as permitted under the United States Copyright Act of 1976, 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.

A BYTE Book.

1 2 3 4 5 6 7 8 9 0 SEM SEM 8 9 3 2 1 0 9 8 7 6 5 4

ISBN 0-07-010679-7

LIBRARY OF CONGRESS CATALOGING IN PUBLICATION DATA
Chasin, Mark.

Assembly language programming for the Atari
computers.

(A Byte book)

Includes index.

1. Atari computer — Programming. 2. Assembler
language (Computer program language) I. Title.

II. Series.

QA76.8.A82C43 1984 001.64'.24 84-11214

ISBN 0-07-010679-7

The editor for this book was Barbara Brooks;
and the editing supervisor was Marthe Grice.
Book design by Sharkey Design.

ASSEMBLY
LANGUAGE
PROGRAMMING
FOR THE ATARI
COMPUTERS

SOFTWARE AVAILABLE

All programs described in this book are available on disk, fully documented and ready to run or modify. You can order this disk by sending a check or money order for \$12.95*, along with your name and address, to:

MMG Micro Software
P.O. Box 131
Marlboro, NJ 07746

*New Jersey residents, please add 6% sales tax. Please allow 2 weeks for all personal checks to clear.

NOTE This software is a product of MMG Micro Software and is not an offering of McGraw-Hill, Inc. We include information concerning this product as a service to our readers.

PREFACE

Since you've picked this book up and started browsing through it, you probably own or have access to an ATARI computer and are interested in progressing beyond BASIC. As you already know, the ATARI computers are among the most impressive of all home computers, but many of their special features are not available from BASIC.

This book is designed to teach assembly language programming to anyone who understands ATARI BASIC. Yes, anyone! You've probably read other books and articles which create a mystical aura around assembly language, or else use the phrase **machine language** as if it were the secret code to unlock the door to the Thief of Bagdad's treasure troves. Of course, only the privileged get to look at this secret treasure.

Bunk! Anyone who has programmed in BASIC, or any other language for that matter, can learn to program in assembly language, given the desire and the correct instruction in the language. This book provides the tools you need. Each programming language — BASIC or PILOT or FORTH or, yes, even assembly language — has its own words which stand for certain operations. One example is PRINT in BASIC, which directs information to your TV screen. The combination of these words, and the way they must be strung together to make the computer do what you want it to do, is called the **syntax** of the language.

In this book, you will learn the syntax of assembly language, and you will also learn, by frequent examples, how to use assembly language to make your ATARI perform tasks which are either impossible from BASIC or 200 times slower in BASIC. The examples are fully documented both by frequent remarks and by a thorough discussion of the purpose, programming techniques, and theory, where appropriate, of each program. This discussion allows you to progress beyond the examples and to write your own subroutines or

even whole assembly language programs for the ATARI. Furthermore, the routines in this book follow the “rules” established by ATARI for assembly language programmers, so they will work with any ATARI computer, from the earliest 400, to the most advanced 1450XLD, and everything in between.

Examples are given both in assembly language and, wherever possible, also in BASIC programs which incorporate these assembly language routines to perform tasks from BASIC. These routines can be used immediately in your own programs. In fact, you can use the enclosed order form to obtain on disk all assembly language and BASIC programs in this book. The disk is ready to run or modify for your own uses. Included on disk and here are such techniques as reading the joysticks, moving players and missiles, input or output to all possible devices such as printers, disk drives, cassette recorders, the screen and more, vertical blank interrupt routines, display list interrupts, fine horizontal and vertical scrolling, sound, graphics — in short, everything you’ve always heard the ATARI computers were capable of but had no idea how to program.

One entire chapter of the book is devoted to the use of assemblers and how to use this book with any of the many fine assemblers available for the ATARI computers. You’ll need an assembler, just like you need BASIC to program in BASIC, and this book will interact with any of them.

If you’ve reached the point where BASIC is no longer enough, and you’d like to progress to a language which gives you absolute control over all functions of your remarkable computer, then begin with Chapter 1, and you’ll see how easy it is. Who knows, maybe you’ll be the one to write the sequel to STAR RAIDERS!

Mark Chasin

PART ONE

BACKGROUND



CONTENTS

Preface vii

PART ONE **BACKGROUND**

- 1 Introduction 1
- 2 Getting Started 11
- 3 The ATARI Hardware 21

PART TWO **LEARNING ASSEMBLY LANGUAGE**

- 4 Nomenclature and the Instruction Set 37
- 5 Addressing Techniques 52
- 6 Assemblers for the ATARI 66
- 7 Machine Language Subroutines
 for Use with ATARI BASIC 79

PART THREE **APPLICATIONS**

- 8 The Display List and Using Interrupts 113
- 9 Input-Output on the ATARI 149
- 10 Graphics and Sound from Assembly
 Language 185

PART FOUR **APPENDIXES**

- 1 The 6502 Instruction Set 215
- 2 The Three Character Sets Used in ATARI
 Computers 279
- 3 The ATARI Memory Map 283

Index 287

CHAPTER ONE

INTRODUCTION

Welcome to the world of assembly language programming for the ATARI computers. By now, you've no doubt tried your hand at programming your ATARI in BASIC and found it to be a very easy-to-use and powerful language. But you've also probably found some things that just can't be done in BASIC, and you know that all of the excellent real-time action games and the fast sorts and searches are all programmed in some mysterious language called **machine language**. The purpose of this book is to teach you how to program your ATARI in the fastest, most powerful and versatile language available, assembly language. By working your way through this book, you will learn how to use all of the sophisticated and powerful resources of one of the most impressive home computers, the ATARI.

Most of the examples in this book will be related to BASIC, so an understanding of BASIC will be important to the understanding of this book. However, many types of programs that can be written in assembly language simply have no counterparts in BASIC, and so for these no such examples will be possible. Problems will be presented throughout the book and it is highly recommended that you try to work them out for yourself. In each case the answers will be presented and discussed, in order to help you if you are having trouble.

VARIETIES OF PROGRAMMING LANGUAGES

At a very fundamental level, your ATARI really only understands one programming language, which is called machine language, the language of the computing machine. A typical machine language program might look like this:

```
1011010110100101.....
```

Now, before you put this book down and go back to BASIC, let's understand one thing right away: virtually no one programs directly in machine language. Even the many programs advertised as being written in "100% machine language" weren't; they were written in assembly language and then translated into machine language. But all computer languages must at some time be translated into machine language in order to be executed, even BASIC. That's right, the central "brain" of your ATARI computer doesn't even really understand BASIC.

BASIC: AN INTERPRETED LANGUAGE

Let's spend a moment discussing how a BASIC program is executed, in an effort to understand better what assembly and machine language really are, and how they differ.

Let's first write a very simple BASIC program:

```
10 PRINT "HELLO"  
20 FOR I=1 TO 200  
30 NEXT I  
40 PRINT "GOODBYE"  
50 END
```

If we now type RUN and hit the RETURN key, we know that the word HELLO will appear on our TV or monitor screen and, after a brief pause, the word GOODBYE will appear directly below

it, followed several lines later by the word **READY**. But exactly how does this happen?

The cartridge containing ATARI BASIC is actually more properly called the ATARI BASIC Interpreter. An interpreter, just like the noncomputer use of the word, is someone or something that translates information from one form into another, whether from English into Russian, or from BASIC into some other language. In our case, the BASIC cartridge contains a program that can translate BASIC keywords into a form understandable to our computer's "brain." Let's see how.

As we type line 10, the word **PRINT** is translated to a code for the word **PRINT**, called a **token**. This process is called tokenizing your BASIC program, and is done as you type each line into your ATARI, and hit **RETURN**. It is this process that simultaneously checks the syntax, or grammar rules, to be sure that you typed the line correctly. If not, you'll see the familiar **ERROR** statement immediately after typing the line, and you then can correct your mistakes before proceeding. This ensures that when the BASIC cartridge begins interpreting your program, it may have logical errors to deal with, but at least each line is internally correct.

Having completely typed the above program, we would then type **RUN** and press **RETURN**, which would begin the interpretation of the program. The first thing this interpreter knows is that the beginning of the program, the place it must start when the word **RUN** is typed, is the lowest-numbered line of the BASIC program. Actually, before it ever gets there, it does quite a bit of housekeeping, such as setting all variables used in your program to zero, canceling out any previously used strings or arrays, and many other functions. Then it turns its attention to line 10, which is converted into machine language by means of something called a jump table, about which we'll learn a great deal in Chapter 9. In any case, first line 10 is translated, then it is executed, and then the machine language code is thrown away, to make room for the next line, line 20. The process of translation, execution, and discarding is repeated for line 20 and then again for line 30, and so on.

Having now executed the entire program, and seen the **READY** prompt that tells us that BASIC is ready for new instructions, what

do you suppose will happen if we type RUN again? Right! The entire process of translating, executing, and discarding each line will be repeated all over again. Then we'll see the READY prompt again. In fact, this entire process will occur as many times as we choose to type the word RUN. As you can no doubt see, this is a very wasteful process. BASIC continues to repeat over and over two of the three steps which are not actually needed to run the program, translation and the discarding of information. If we could only get away from the need for these two steps, imagine how fast our program would execute. After all, if we get rid of these two steps, the only one left is execution.

ASSEMBLY LANGUAGE: AN ASSEMBLED LANGUAGE

Now you know the purpose of assembly language programming! When we program in assembly language, by using a translator known as an **assembler**, we can produce the executable machine language code which we can store, and which the computer can execute directly. We translate it only once and we don't discard it at all, so we get maximum efficiency, and therefore, maximum speed. And that's the real benefit of assembly language programming, speed. In fact, it is possible to write a program in assembly language which will execute over 1000 times faster than its BASIC equivalent! For arcade games, and very time-consuming processes like moving blocks of memory around, searches, sorts and other such procedures, assembly language programming can be absolutely indispensable.

The other major advantage of assembly language is the absolute control it gives the programmer over the computer. In BASIC, the programmer is often separated from the nuts-and-bolts hardware of the computer and doesn't have detailed control over many of its functions. This control is available only through assembly language programming.

INTERPRETED VERSUS ASSEMBLED LANGUAGES

These are the advantages of assembly language programming: speed and control. How about the disadvantages? First, of course, is the need to learn a new computer language. This book will enable you to do that. Second, ATARI BASIC is an interpreted language, while assembly language is not. This becomes important when you need to make changes in a program. In BASIC, you simply make the change and rerun the program. For example, to change the above program, we might simply type:

```
40 PRINT "GOODBYE";  
50 PRINT "Y'ALL"  
60 END
```

Now when we run the program, it will say GOODBYE Y'ALL instead of just GOODBYE, as above. The entire change in the program might take 15 seconds for a very slow typist. This flexibility is a great advantage of interpreted languages. To make a similar change in an assembly language program would require much more typing, and then the program would have to be reassembled. This assembly process, converting the assembly language program to machine language, sometimes takes 15 minutes or more, depending on the size of the program and the assembler used. Of course, our example is very short and would not take this much time, but the point is that making even a very simple change to an assembly language program might take quite a while, and if you make a mistake, you'll need to repeat the process all over again!

A third disadvantage of assembly language is the amount of programming you'll need to do to accomplish even the simplest tasks. For instance, the PRINT statement in BASIC, which requires you to type only one word, might require 20 or 30 **lines** of programming in assembly language. For this reason, assembly language programs are usually very long.

The fourth, and last, disadvantage of assembly language is the difficulty of understanding a printout of the program. Certainly the PRINT statement in BASIC is far more understandable than a series of instructions such as:

```
LDA #$01  
STA CRSINH
```

or something equally obtuse. This problem can and should be overcome by all good assembly language programmers by the inclusion of comments on virtually every line. Comments are the assembly language equivalent of REM statements in BASIC: they help the programmer to remember what it was he or she was trying to accomplish with a given line. Certainly the above example makes somewhat more sense when presented below with comments, even to someone who doesn't understand assembly language at all.

```
LDA #$01      ;to inhibit cursor  
STA CRSINH    ;poke a 1 here
```

Now perhaps it's more understandable that when we see a program advertised as written in "100% machine language," what is really meant is that it was written in assembly language, and then translated once from its final form into machine language, which is the form in which it is being sold. Such programs generally are much faster to execute than BASIC programs, and the additional control the programmer has over the computer allows special effects not attainable from BASIC.

There is an additional distinction between BASIC and assembly language. BASIC belongs to a family of programming languages which are referred to as **high-level** languages. This nomenclature refers to the ability of one simple statement to perform quite a complicated task, such as the PRINT example used above. In a sense, this ease of programming also isolates the programmer from the hardware, placing him or her at arm's length, so to speak. It is from this view of languages such as BASIC that the term **high-level language** arose. Among *thousands* of other high-level languages are Pascal, FORTRAN, PILOT and Ada. In con-

trast to these, languages such as machine language or assembly language are referred to as **low-level** languages, because to program using them requires an understanding of the hardware and an ability to get into the real guts of the machine for which you are programming.

WORKING WITH ASSEMBLY LANGUAGE

In order to convert an assembly language program to machine language, we must use another program, called an **assembler**. There are a number of excellent assemblers available for the ATARI computers, and the techniques used in this book will work with any of them. Chapter 6 is devoted to the syntax and special functions of each assembler, but the assembly language programs listed in this book were produced using the Assembler/Editor cartridge from ATARI. Chapter 6 specifies all of the changes required to use these programs with each of the other assemblers.

COMPILERS

There is another way to convert programs to machine language. A compiler is a program which converts a program written in a high level language such as BASIC to machine language. These compilers generally convert the entire program all at once, in contrast to an interpreter, which translates each line one at a time. The converted program created by the compiler can then be run without a BASIC cartridge installed, and will generally be from five to ten times faster than the original BASIC program. Why only five to ten times? These compilers are very complex programs, which must take into account every possible combination of BASIC commands anyone might write. Therefore, they create machine language code which performs all of the correct steps in the original program, but they cannot optimize the code produced. Therefore, in general, programs written in assembly language and assembled into machine language will execute much faster than the same program written in BASIC and compiled.

The other major disadvantage of compiled code is its size. For instance, some of the subroutines in Chapter 7 are about 100 bytes long. The same routines written in BASIC and compiled could be as long as 8000 bytes! It would be very hard to use these as subroutines in a BASIC program as we do in Chapter 7.

TERMINOLOGY

Before we go on, let's talk about a number of terms that are frequently used by programmers. It's the jargon of their trade. Just so we all are speaking the same language, then, let's briefly review some of them. When we speak about computer memory, we frequently hear the terms ROM and RAM mentioned. ROM stands for **Read-Only Memory**, and memory of this type can be read but not written to. For instance, in the ATARI, all memory locations higher than 49152 are ROM, and although in BASIC we can PEEK them to see what is stored there, we cannot POKE new values into them. "But what about player-missile graphics?" you may ask. "We POKE memory locations higher than this all the time!"

True, but if you were to then PEEK at that location, you would find that you hadn't really changed anything at all. The value stored in that location is not changed by such POKES. It is the act of writing to that address which causes the changes you see in player-missile graphics or other applications requiring writing to memory locations above 49152.

This is in direct contrast to RAM, which stands for **Random-Access Memory**. Actually, both ROM and RAM are random-access, and RAM should more properly be called Read-Write Memory; but since RWM is unpronounceable, RAM has become the accepted term. The term **random access** refers to the method by which information is accessed, and is to be contrasted with **sequential access**, the other major method of storage. Sequential access can best be envisioned by imagining an audio tape. In order to play a song in the middle of the tape, you must somehow scan through the entire first portion of the tape, either by playing it or by using the fast-forward key. In contrast, think of a phonograph record. To