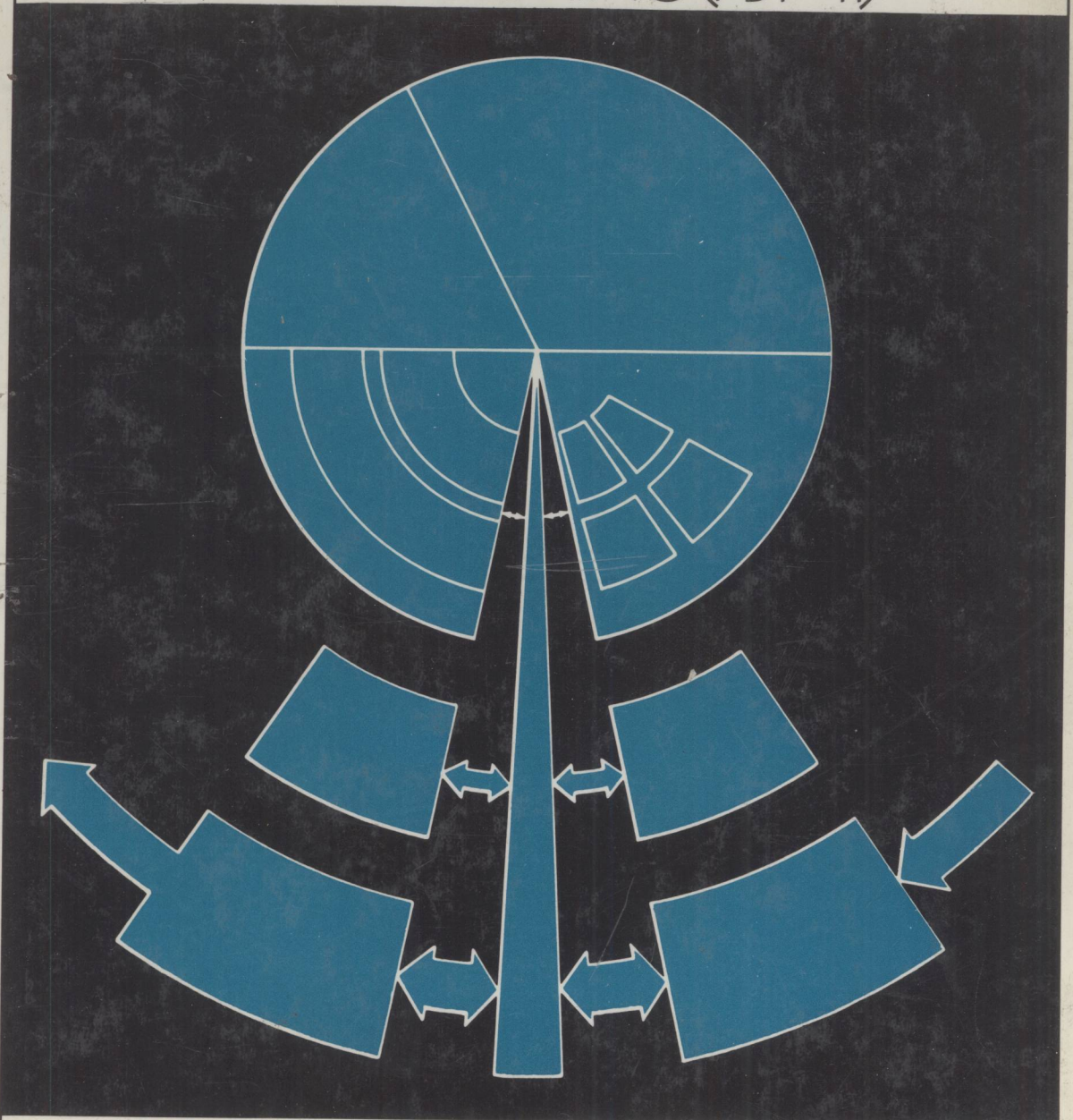
THE DIGITAL WAY

MACRO-11 ASSEMBLY LANGUAGE PROGRAMMING (PDP-11) T.M.



J.F. PETERS, III

The Digital Way: MACRO-11 Assembly Language Programming (PDP-11™)

J.F. Peters, III
Saint John's University
Collegeville, Minnesota

Reston Publishing Company, Inc.
A Prentice Hall Company
Reston, Virginia

Library of Congress Cataloging in Publication Data

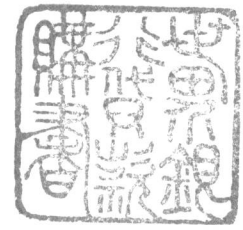
Peters, James F.
The digital way.

1. PDP-11 (Computer)—Programming. 2. Assembler language (Computer program language) I. Title.
QA76.8.P2P48 1985 001.64'2 84-18243
ISBN 0-8359-1315-5

Interior design and editorial supervision by Alice Cave

10 9 8 7 6 5 4 3 2 1

PRINTED IN THE UNITED STATES OF AMERICA



To Kathie

Foreword

Why would someone want to read a book about assembly-language coding in 1985? Everyone knows that assemblers are passe, and that even ''higher-order'' languages are being phased out by ''program generators'' and turnkey packages. Well ''everyone'' is wrong. There are more people using assemblers today than when they were first introduced about thirty years ago.

Assemblers are here to stay. Every serious programmer, whether a professional or a hobbyist, will have an assembler available in his toolkit. Communicating to a computer in its own terms gives the programmer an additional level of control that is lacking when he uses an intermediary (i.e. someone else's compiler). It requires additional effort, additional skill, and additional knowledge to exercise this control, but it is sometimes advantageous, or even essential, to do so. We can make a rough analogy to using a manual rather than an automatic shift in a car. The manual shift requires more effort, skill, and knowledge, but can result in more efficient use of the car (more miles per gallon of gas). And in driving on icy roads, the additional control can help recover from, or even avoid dangerous skids. But automatic shifts make it possible for many people to drive who would not otherwise be able to do so. So, too, with programmers.

I have been in the computer business for thirty-four years. For the first seven of these years my coding was done in machine language. I saw very little advantage to be gained by using such advanced tools as assemblers until I encountered my first binary machine with a ''large'' 4096-word magnetic-core

memory (until then, the computers I had been using were all decimal machines with relays or rotating drums for memory). When FORTRAN came along about a year later, it was a godsend. Even though it produced code that ran relatively slowly and used 50 percent more machine capacity, I could now have the "users" do their own coding for their trivial problems, while the efforts of my staff could be devoted to the larger or more difficult ones, using the more efficient assembler. The "dilettantes" and "amateurs" could now code their own programs, while the "professionals" stayed with machine and assembly language coding. But soon, machines became bigger, faster, and relatively less expensive, and even professionals recognized that it was often cost-effective to use compilers. The rapid growth in the demand for computer programs soon outstripped the ability of the professional programmers to keep up and the capacity of schools to train more. And so, assembly-language coding was relegated to those applications which required the ultimate performance of computers.

The discovery of assemblers has been repeated several times. When control engineers switched from relays and analog controls to micro-computers and digital controls, they, too, started with machine-language coding. In the late '60s, articles appeared in the engineering magazines explaining the advantages of using an advanced tool: an assembler. In the middle '70s, similar articles appeared in the magazines devoted to the hobbyist who put together his own computer.

Today, the assembler is being approached from a slightly different direction. There are approximately a million "home" or "personal" computers in use now. Most of these computers are being programmed with BASIC. The implementers of the BASIC interpreters recognized the need to use machine-language code to allow some control of the detailed capabilities of the underlying microcomputer, and to speed up certain functions. This led to a crude machine-language capability to be incorporated in the form of PEEK and POKE commands. It didn't take long before the popular computing magazines started publishing articles on the advantages of assemblers over the clumsy POKEing of decimal translations of machine-language codes.

And so, even today, assemblers are being touted as an advanced way of exercising the detailed control of, or achieving the speed and capacity inherent in computers.

Assemblers have died many times. We hear over and over again that computer hardware is cheaper than software. I, too, believed this fairy tale. But one time, when I proposed to a project manager that he could save a million dollars in software development for his project by simply doubling the memory size in the computer we were using, he pointed out the fact that we were going to make 500 or more systems, each with two computers, and multiplying the "cheap" memory price by 1000 came to more than a million dollars. The development cost of the software may have been greater than the production cost of a single computer, but amortized of 1000 copies, the unit software cost was negligible.

Of course, that was more than fifteen years ago. Today, things are different. Or are they? Let us take a more recent example: the Space Shuttle. That project, too, suffered from an overrun in the use of memory. And so, IBM installed a "double-density" memory that almost doubled the capacity. However, because an additional pound of non-payload weight costs $65,000 and an additional kilowatt of power consumption costs $35,000 in the Shuttle's operation, the cost of the additional memory was $.27 per bit, IN ADDITION TO the price IBM charged NASA for the memory and the paperwork necessary to change the specifications of the computer!

For more than twenty years, I made a career out of using assembly code to squeeze programs into the available computer capacity to execute within the required time.

Even as I write this, I am involved in a project that has overrun the capacity of the computer selected and is taking so long to execute that the system is unacceptable without adding an additional computer. Needless to say, the program was coded in a "higher-level" language.

In summary, there are enough occasions requiring coding at the machine level, that no serious programmer should be without an assembler in his toolkit. As an added incentive, learning how to use an assembler gives the programmer an insight into the functioning of a computer and the representation and organization of data that he would not get otherwise. Reading this book will get you started.

David Feign, Ph.D.
Santa Ana, California

Preface

This text was written with the idea that at each step model programs, lots of examples, will lead to parallel programs. These model programs can serve as a guide for those who are just learning to program on the assembly language level. They also can be helpful to those who want to tune their knowledge of the assembly language level.

It helps a great deal in the beginning to have assembly language to imitate, not fragments but complete, runnable programs. As much as possible in this text, complete sample programs have been presented. These programs have been written with the principles of structured programming in mind.

This text was also written to encourage new assembly language programmers to design their own instructions. It makes no sense at all to approach assembly language programming with the idea that only the native instructions provided by an assembler will be used. New instructions can be created, designed, implemented using subroutines like those found in MACRO-11. New instructions can also be developed using macros. The combination of subroutines and macros offer a powerful incentive to invent, to design, to learn the art of assembly language programming.

Learning to program on the assembly language level is a good way to get closer to the organization of a particular computer. Assembly language programming entails a keen perception of the way memory is organized inside a computer. An assembly language programmer needs to get very close to the

pulse of a machine, its registers, its methods of managing memory, its execution cycles, its native instruction repertoire.

In terms of both machine organization and assembly language programming, this text works on two levels. With machine organization, both generic concepts as well as machine-specific features are presented. With assembly language programming, both the generic notion of assembly in terms of either the two pass or one pass method as well as MACRO-11 programming specifics are presented. Finally, both the generic notion of macro assembly as well as the specifics of MACRO-11 macro assembly are presented.

The aim of this book is to give a perspective which goes beyond the demands of one computer system. The aim of this book is to develop a sense of what it takes to write an assembly language program on any machine as well as a PDP-11 or DEC Professional computer.

Assembly language programming on a PDP-11 is a pleasure. Its instruction repertoire is rich, broadly conceived, workable, and useful. It is a good system to learn on.

Note for the Student

You will probably find the selected solutions to the exercises and lab projects helpful. To try out the programs in the chapters or in the selected solutions, you will want to set up a copy of the macro library called paslib.mac. This is in Appendix H.

There are two versions of paslib.mac. The version in Appendix H.2 will be useful for RT-11 or RSTS users. The version of paslib.mac in Appendix H.3 is for RSX-11 users, since it relies entirely on keyboard registers instead of system i/o macros (.ttyout and .ttyin) available in RT-11.

To set up a version of one of these macro libraries in your account, use an editor to type them into a machine (a PDP-11, any model, or a professional with the tools package). Instructions for setting up the macro library using the librarian program are given in Appendix H, also.

You might want to set up a coop so that these libraries can be shared, rather than keep a copy of the macro library in each person's account. There are lots of programs you can write without a macro library like the one in Appendix H. A macro library like paslib.mac will just make life easier for you, especially in the beginning.

Each chapter has a summary and review quiz, which you will probably find helpful in preparing for exams. There are also references to outside reading you might want to follow up in connection with this book.

Finally, I really would appreciate any suggestions you might have. Please let me know if you would like to see any additional and/or different things in the summaries or in the examples or selected solutions which are not there now.

I really want to encourage you to write to me, if you have any questions, sample programs, suggestions. You can reach me by writing to:

Computer Science Department
St. John's University
Collegeville, Minnesota 56321

Note to the Instructor

Copies of all of the programs and libraries are available on 5¼, quad density, RX50 diskettes used on Rainbow-100s or on a DECMATE. These can be purchased by writing to me at the above address.

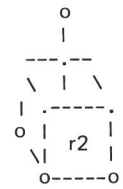
In using this book, you might want to try the following scheme:

	Topic	*Classtime*
1.	Assign chapters 1 and 2 to be read	0 hours
2.	Start with chapter 3 (examine memory to get started without a macro library)	2 hours
3.	Illustrate the uses of macros in chapter 4 (you will need paslib.mac in Appendix H to do this)	1 hour

Note: Try creating simple macros *without* arguments to demonstrate, further, the idea of inventing new instructions. R. Pattis, *Karel the Robot: A Gentle Introduction to the Art of Programming*, John Wiley & Sons, N.Y., 1981, ch. 2, is helpful, here. Try

```
begin:
    drawrobot
    drawramp
    drawarrow
    drawbeeper
    .exit
    .end begin
```

without any reference to the macros themselves at first, to draw a picture like the following one:



Then create the macros to *define* the instructions, using the chapter i/o macros, which come from Appendix H.

4. Control Structures, chapter 5 (this chapter draws a parallel between the control structures found in Pascal and those that can be put together in an assembler program) 2 hours
5. Stacks and queues, chapter 6 2 hours
   NOTE: At this point, it is helpful to look at the material in chapter 2 on base conversions
6. Closed routines, chapter 7 (problems from cryptography and operating systems have been brought into this chapter). 2.5 hours
7. Assign chapter 8 to be read 0 hours
8. Directives, chapter 9 (stress uses of list files in debugging and use of storage directives) 1 hour
9. I/O, chapter 10 (begin creating personalized i/o routines to manage traffic to-and-from a keyboard) 1 hour
10. Arrays, chapter 11 (intensive) 1.5 hours
11. Advanced i/o, chapter 12
12. Sidelights (assign to be read) 0 hours
13. Chapter 14 (random numbers) 1.5 hours
14. Chapter 15 (assign to be read) 0 hours
15. Chapters 16–17 (intensive) 3 hours
16. Chapter 18 (recursion) 1 hour
17. Chapter 19 (traps) 1 hour
18. Chapter 20 (files) 1 hour

Organization of the Text

Exercises are easier than lab projects. They take less time. They are more for individual work. Lab Projects have been designed with a team approach in mind.

A key aim of this text has been to encourage writing macros from the beginning. These are followed, soon afterward (in chapter 7), with an introduction to subroutines. By chapter 10, it should be possible to write subroutines to deal directly with the problem of terminal i/o.

Chapters 1–7 are i/o macro dependent. Chapters 9–20 rely on the free use of both macros and closed routines. Chapters 15–18 demonstrate how new, complete macros with parameters can be created.

Acknowledgments

I want to pay special tribute to Mark Tinguely and Susan Rosenberger at St. John's University. They have worked with me for several years, helping to prepare programs, debugging and smoothing out programs and macros. Mark Tinguely and Bob Meierhofer produced the bulk of the solutions to the exercises and lab projects.

I also wish to thank the following persons for their help and encouragement:

Lori Kaufenberg, Patrick Holmay, Geoff Brunkhorst, Paul St. Michel, Diana Messina, David Weigel, Dr. Hamed Sallam, Dr. Sylvester Theisen, Ruth O. Devolder, Gordon Tavis, O.S.B., Richard Pletcher at Compusolve, Hank Marvin at Western Electric, Leon J. Schilmoeller at 3M, and Ginger DeLacey and Alice Cave at Reston.

J.F. Peters, III
Collegeville, Minnesota

Contents