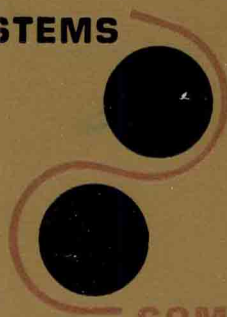


**INTRODUCTION TO COMPUTER SYSTEMS**  
**Using the PDP-11 and Pascal**



**COMPUTER  
SCIENCE  
SERIES**

---

# **INTRODUCTION TO COMPUTER SYSTEMS**

Using the PDP-11 and Pascal

---

**Glenn H. MacEwen**

**McGraw-Hill Book Company**

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

## **INTRODUCTION TO COMPUTER SYSTEMS**

Using the PDP-11 and Pascal

Copyright © 1980 by McGraw-Hill, Inc. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

67890 HDHD 898765

This book was set in Times Roman by Progressive Typographers.  
The editors were Charles E. Stewart and Frances A. Neal;  
the production supervisor was Richard A. Ausburn.  
The drawings were done by Burmar.

### **Library of Congress Cataloging in Publication Data**

MacEwen, Glenn H

Introduction to computer systems.

(McGraw-Hill computer science series)

Bibliography: p.

Includes index.

1. PDP-11 (Computer) 2. Pascal (Computer program language) I. Title.

QA76.8.P2M3 001.6'4 79-22646

ISBN 0-07-044350-5

---

# PREFACE

---

This text provides an introduction to the internal logical structure of computers and the techniques of machine-level programming. It is intended to give the student an understanding of the basic structure and functioning of conventional computer systems.

The student should possess a programming ability in some high-level language as the necessary background. The level of competence required is that obtainable from one full course of study in programming and applications.

On completion of the course described in this text the student should have the ability to read the documentation of a conventional computer system and be able to understand the basic functioning of the machine as well as the services provided by the operating system. For example, the documents commonly called “Principles of Operation” and “System Programmer’s Guide” should be understandable.

Basic computer structure is taught in the context of a particular machine, the Digital Equipment Corporation PDP-11. At first, assembly-language programming is used to provide an understanding of machine functioning. High-level system programming languages are then introduced as a vehicle for programming in the remainder of the text. Various architectural and programming topics are taught in the context of two particular PDP-11 operating systems, RT-11 and UNIX,<sup>1</sup> although each topic is generalized as much as possible in accompanying discussion.

Data structures are not treated as a separate topic but are introduced in the context of particular applications. This is very specifically done on the assumption that most students will probably receive a complete treatment of data structures in another course.

This text has been used in a course at the second-year level of a computer science program and has also provided the basis for a graduate half course in

<sup>1</sup> UNIX is a trademark of Bell Laboratories.

electrical engineering. Although the text assumes a good exposure to programming in a well-structured high-level programming language, it may be that not all students will have this preparation. It is usually necessary, therefore, to supplement the chapters on the languages Pascal and C with some tutorial material. Pascal, particularly, should be introduced early if sufficient familiarity has not been attained, since many algorithms early in the text are expressed in Pascal-like notation.

All material required for the curriculum of course CS-3, Assembly Language Programming, prepared by the ACM Committee on Curriculum in Computer Sciences, is included. Specifically, parts of Chapter 1, Chapters 2 through 9, and Chapter 14 provide a basis for this course with sufficient optional material to give the instructor some flexibility. In addition, Chapter 0 gives a preview of some material so that the student can begin to apply, almost immediately, what he or she will learn in more detail later. It is recommended, also, that some material on number systems from Chapter 1 be included in any version of CS-3 to be offered from this text.

The text provides an excellent basis for further study in computer architecture and operating systems. The approach taken here is to present some of the mechanisms that will be encountered in the study of operating systems without attempting to introduce many of the abstractions that are useful in designing operating systems. In this way the student will be well prepared to study these abstractions, having seen examples of the problems they are intended to solve. The text is, therefore, essentially a bottom-up approach to systems programming, since the author feels that students must understand examples before they are ready to fully understand abstractions. To this end the advanced material stops just short of the issues and complexity that are more properly studied with the abstractions of operating systems theory.

Material has been carefully sequenced to avoid forward references and to present only sufficient information for the reader to progress in relatively modest steps. The major exception to this is Chapter 0, in which a bit of a preview is given so that the student can begin to program immediately. This is intended to avoid making the early material too dry while the groundwork is laid for programming topics.

Chapter 1 contains the necessary introduction to number systems. Where this material has already been covered in another course, one can start with Chapter 2, which discusses the basic components of a computer in general terms. Chapter 3 gives a partial view of the PDP-11; the intent here is to avoid becoming involved in a discussion of the unique way that addressing is accomplished in the PDP-11. Consequently, students can move on quickly to programming in Chapter 4 without getting stuck on these details, which are covered in Chapter 5.

Chapter 6 comprises some elementary topics concerned with structuring programs. Macros, however, are left to Chapter 7 where they are treated rather extensively. It is important to include at least the basic material on macros in order to understand the material that follows.

Chapter 8 fulfills two major objectives. First, it explains the basic algorithms of an assembler so that the translation process is made clear to the student. Second, it uses the symbol table as an example within which to discuss the topic of sorting and searching. This, of course, follows the policy of treating data structures only in the context of particular applications. Much of the material in this chapter can, however, be omitted without affecting comprehension of following material.

Chapter 9 brings together all considerations of how a program is transformed from assembly-language form into its final machine-language form. Chapter 10 introduces concurrency, which was carefully omitted from the introduction to I/O in Chapter 5. It is not until this point that interrupts are introduced, so that the student has a rather solid foundation before having to cope with this difficult area. Chapter 11 introduces the supervisor as a program that makes the machine more reliable and more convenient to use. Traps and interrupt handling services are covered here.

Chapters 12 through 17 constitute a set of special topics that can be selected according to the needs of a specific curriculum. In our course at Queen's approximately 25 percent of the time is spent on these topics. However, since some features of Pascal that do not appear in earlier chapters are used in these chapters, a review is included in Chapter 12. The description of the language is not complete and is given rather informally so that a language text or manual is necessary if programming is to be assigned. The language C is also included here as an example of another system language and to enable those with access to UNIX system documentation to read programs.

Although Chapters 13 through 17 may be selected as desired, I would expect Chapter 15 on multiprogramming to be given priority because of its central importance to the subject of systems programming. Chapter 17 is rather ambitious, attempting to introduce methods of systematic software system design. The material here derives largely from the work of D. L. Parnas and J. F. Guttag. I must, however, take responsibility for attempting to properly interpret their work.

Finally, Chapter 18 describes the structure of a moderately large program that was designed according to a disciplined method. There are several ways to use it. Portions can be introduced throughout the course and given as programming assignments so that a student has a complete program at the end of the course. (This is the way that I have used it.) It can be used as an exercise in group programming. It can simply be studied as an exercise in design principles. Whichever route is chosen, this project provides exposure to large software system design, something that is not often gained by students.

A particularly difficult aspect of producing a book is deciding when to stop. One is never quite satisfied with the work nor completely confident that errors do not lurk in hidden places. But at some point it is necessary to say "this is it" and to send it off to the publisher. I therefore take full responsibility for deficiencies or errors that are found and would appreciate hearing about them quickly.

The manuscript was prepared in the Department of Computing and Information Science at Queen's University and partly in the Department of Electrical Engineering at Royal Military College of Canada, to both of which I am grateful for the facilities to do this. The former made available the computing facilities with which the manuscript was prepared using a text editor written by I. A. Macleod and D. G. Ross. Appreciation is due to many people who contributed typing, editing, and reading of the manuscript. To attempt to list all would risk omissions; those involved are aware of their contribution. Particular mention, however, is due to T. P. Martin, who did the original design of the MITE system described in Chapter 18. His M.Sc. thesis work, which included the design, was supported by a grant from the National Research Council of Canada. Finally, I wish to acknowledge the tolerance of those to whom I made work commitments that were somewhat neglected as book writing tended to swallow the available time.

*Glenn H. MacEwen*

---

# CONTENTS

---

Preface		xv
Chapter 0	Introduction	1
0-1	Algorithms and Languages	2
0-2	Assembly Language	4
0-3	Getting Started	7
	0-3.1 Machine Access	8
	0-3.2 Files	9
	0-3.3 The Operating System	9
	0-3.4 Systems Programs	9
Chapter 1	Representation of Numbers and Data	11
1-1	General	11
1-2	Arithmetic in Nondecimal Bases	12
1-3	Conversion of Numbers between Bases	13
1-4	Representation of Negative Numbers	15
1-5	Overflow	20
1-6	Fractional Numbers	22
1-7	Alphanumeric Data Representation	23
Chapter 2	Basic Computer Structure	26
2-1	Memory	26
2-2	The Processor	27
2-3	Representation of Programs	30
	2-3.1 Three-Address Machines	31
	2-3.2 Single-Address Machines	34
	2-3.3 Stack Machines	36
	2-3.4 Practical Processors	37
2-4	Internal Processor Structure	37
2-5	Program and Data Separation	40



<b>Chapter 3</b>	<b>The PDP-11: A Simplified View</b>	<b>42</b>
3-1	PDP-11 Memory	42
3-2	PDP-11 Processor	44
3-3	Direct Addressing	45
3-4	Register-Mode Addressing	48
3-5	Immediate Addressing	50
3-6	Indexed-Mode Addressing	53
3-7	Indirect-Mode Addressing	58
3-8	Single-Address Instructions	61
<b>Chapter 4</b>	<b>Assembly Language</b>	<b>64</b>
4-1	Assembly Time versus Execution Time	64
4-2	Statements	65
4-3	Labels	65
4-4	Reserving and Initializing Memory	66
4-5	Example Program 1	68
4-6	Example Program 2	69
4-7	Example Program 3	71
<b>Chapter 5</b>	<b>The PDP-11: A More Detailed View</b>	<b>74</b>
5-1	Addressing Modes	74
5-2	Program-Counter Addressing	79
	5-2.1 Immediate and Absolute Addressing	79
	5-2.2 Relative Addressing	81
5-3	Condition Codes and Branching	83
5-4	Using the Condition Codes	86
5-5	The Jump Instruction	92
5-6	Example Program 4	93
5-7	Logical Operations: Bit Addressing	94
5-8	Input-Output	96
5-9	Input with the Paper-Tape Reader	98
5-10	Output with the Paper-Tape Punch	100
5-11	Example Program 5	101
<b>Chapter 6</b>	<b>Program Segmentation</b>	<b>107</b>
6-1	The System Stack	107
6-2	Subroutine Instructions	112
6-3	Input-Output Subroutines	114
6-4	Example Program 6	116
6-5	Parameter Passing	120
6-6	Register Saving	126
6-7	Coroutines	129
6-8	Reentrancy	132
6-9	Recursion	135
6-10	Position-Independent Code	137
6-11	Program Module Interface Specifications	140

6-12	Example Software System	142
6-13	Error Returns	151
<b>Chapter 7</b>	<b>Macro Assembly</b>	<b>154</b>
7-1	Defining a Macro	154
7-2	Conditional Assembly	156
7-3	Definite Iteration Assembly	159
7-4	Concatenation	161
7-5	Numeric-Valued Parameters	161
7-6	List-Valued Parameters	163
7-7	Created Symbols	166
7-8	Subconditionals	167
7-9	Testing Arguments	170
7-10	Indefinite Iteration Assembly	172
7-11	Nested Macro Definitions	172
7-12	Recursive Macro Calls	173
7-13	System Macros and Listing Control	173
7-14	Macros and Subroutines	174
<b>Chapter 8</b>	<b>Assembler Construction</b>	<b>180</b>
8-1	Two-Pass Assembly	181
8-2	One-Pass Assembly	183
8-3	Designing an Assembler	184
8-4	Interpretation and Translation	192
8-5	Symbol Table Organization	193
8-6	Linear Storage	194
	8-6.1 Linear Insertions	194
	8-6.2 Linear Searching	194
	8-6.3 Sorting a Linear Table	195
	8-6.4 A Sort Procedure	204
	8-6.5 Two-Pass Assembly with Linear Storage	207
8-7	Hashing	209
	8-7.1 Insertions with Hashing	210
	8-7.2 Searching with Hashing	211
	8-7.3 Sorting a Hash Table	214
	8-7.4 Two-Pass Assembly with Hashing	214
8-8	Tree Storage	214
	8-8.1 Tree Insertions	215
	8-8.2 Tree Searching	216
	8-8.3 Sorting a Tree	217
	8-8.4 Two-Pass Assembly with Tree Storage	218
8-9	Packing Symbols	218
<b>Chapter 9</b>	<b>Linking, Loading, and Interpretation</b>	<b>222</b>
9-1	Absolute Loaders	223
9-2	Relocating Loaders	226

9-3	Linkers	232
9-3.1	Relocatable Load-Module Output	234
9-3.2	Absolute Load-Module Output	238
9-3.3	Object-Module Output	239
9-4	Translate-and-Go Systems	240
9-5	Interpretation	240
<b>Chapter 10</b>	<b>Concurrent Input-Output</b>	<b>243</b>
10-1	I/O Instructions	243
10-2	Busy-Waiting I/O	245
10-3	Interrupts	247
10-4	Enabling and Disabling Interrupts	250
10-5	Interrupt Return	251
10-6	Masking Interrupts	252
10-7	Interrupt Priorities	255
10-8	Noninterrupt Systems	258
10-9	Classification of Devices	258
10-10	Direct Memory Access	259
10-11	Buffering	260
10-12	Input-Output Processors	263
10-13	PDP-11 I/O Devices	264
10-13.1	Paper-Tape Reader-Punch PC11	265
10-13.2	Alphanumeric Display-Keyboards VT52	266
10-14	Real-Time Systems	268
<b>Chapter 11</b>	<b>Supervisors and Traps</b>	<b>270</b>
11-1	Error Traps	271
11-2	Program Traps	272
11-3	Interrupt Handling	274
11-4	Supervisor State	275
11-5	Alternative Architectures	277
<b>Chapter 12</b>	<b>Systems Languages</b>	<b>279</b>
12-1	Pascal	279
12-1.1	Data Types	280
12-1.2	Expressions	285
12-1.3	Statements and Control Structures	285
12-1.4	Program Units	286
12-1.5	Pascal Example	286
12-2	The Language C	291
12-2.1	Data Types	291
12-2.2	Expressions	293
12-2.3	Statements and Control Structures	294
12-2.4	Program Units	295
12-2.5	C Example	296

<b>Chapter 13</b>	<b>Operating Systems</b>	<b>302</b>
13-1	RT-11	304
	13-1.1 The Monitor	304
	13-1.2 Keyboard Commands	306
	13-1.3 Program Requests	306
	13-1.4 Command String Interpretation	307
	13-1.5 Systems Programs	307
13-2	The UNIX System	308
	13-2.1 The File System	308
	13-2.2 The Shell and Programs	309
	13-2.3 File Transfers	309
	13-2.4 Systems Programs	311
 <b>Chapter 14</b>	 <b>Auxiliary-Memory Devices and Physical Files</b>	 <b>312</b>
14-1	Economic Considerations	312
14-2	Auxiliary-Memory Devices	313
	14-2.1 RX11 Floppy Disk System	318
14-3	Physical Files	319
14-4	Sequential Files	321
14-5	Indexed Files	323
14-6	File Directories	325
14-7	RT-11 Files	326
14-8	UNIX Files	327
14-9	Access Methods and Logical Files	328
 <b>Chapter 15</b>	 <b>Multiprogramming</b>	 <b>331</b>
15-1	Performance Considerations	331
15-2	Reentrant Programs	333
15-3	The Multiprogramming Supervisor	333
15-4	Processor Scheduling	336
	15-4.1 Short-Term Scheduling	336
	15-4.2 Nonpreemptive Scheduling	337
	15-4.3 Preemptive Scheduling	339
15-5	Memory Management	342
	15-5.1 Fixed Allocation	343
	15-5.2 First-Fit	344
	15-5.3 Best-Fit	346
	15-5.4 Buddy Systems	349
	15-5.5 Compaction	350
15-6	RT-11 Multiprogramming	350
15-7	UNIX System Multiprogramming	351
 <b>Chapter 16</b>	 <b>Memory Mapping and Protection</b>	 <b>357</b>
16-1	Protection Keys	358
16-2	Memory Mapping	359

16-3	Base and Bound Registers	360
16-4	Paged Allocation	361
16-5	Segmented Allocation	363
16-6	Further Study in Memory Mapping	364
16-7	PDP-11 Memory Mapping	365
16-8	UNIX System Use of Memory Mapping	367
<b>Chapter 17</b>	<b>Software Design Specification and Testing</b>	<b>370</b>
17-1	Program Quality	370
17-2	Abstraction	371
17-3	Design Methods	372
	17-3.1 Top-Down Programming	373
	17-3.2 Information-Hiding Modules	373
17-4	Specifications	374
17-5	Abstract Data Types	378
17-6	Structural Design	381
17-7	Formal Semantic Specifications	388
	17-7.1 Axiomatic Specifications	388
	17-7.2 Abstract Model Specifications	390
17-8	Error Handling	393
17-9	Testing	395
	17-9.1 Objectives of Testing	396
	17-9.2 Test Data Selection	396
	17-9.3 Test Management	397
<b>Chapter 18</b>	<b>The MITE System</b>	<b>400</b>
18-1	MITE Functional Specifications	400
18-2	MITE Structural Design Using Abstract Data Types	402
18-3	Implementation Notes	434
	<b>Appendixes</b>	<b>438</b>
A	Floating-Point Numbers	438
B	Summary of PDP-11/03 Instructions	443
C	Pascal Syntax	446
	<b>Bibliography</b>	<b>452</b>
	<b>Index</b>	<b>457</b>

## INTRODUCTION

The power of a digital computer derives from its ability to store a large quantity of information, in a way permitting ready access, and to process that information. In the following chapters we first consider methods of representing information for computer storage. These methods reflect the fact that the physical devices used for storage, called *memory devices*, consist essentially of a set of switches, each switch being in one of two states: on or off. All information to be stored must, then, be transformed into a representation that can be mirrored by a set of switches.

Having a basic understanding of information representation, we will move on to consider the basic mechanisms within a computer that enable the machine to process information. These mechanisms form a set of basic operations which may then be combined to form an algorithm. Algorithms, based on the operations provided by the machine, are specified by writing a program in machine language. A computer can execute a program expressed in machine language in a way analogous to that of a person carrying out the actions expressed in a list of instructions printed on a piece of paper. An interesting feature of most computers is that machine-language programs are stored in memory devices in exactly the same way as the data to be processed.

A programmer need not write programs in machine language; it is a very tedious process to specify an algorithm by setting a vast number of switches. Most computer manufacturers supply customers with a program called an *assembler* that translates programs written in a symbolic programming language into machine language. Such symbolic languages, called *assembly languages*, mirror the internal structure of the computer in their design. Assembly language is used in situations where a programmer needs precise control over the internal functioning of a computer and the programs are small enough to be

manageable without more elaborate languages. In our case, assembly-language programming provides an excellent means of gaining an understanding of the internal functioning of computers. The basic hardware computer, manifested by circuits and wires, needs a vast amount of program, its software, to transform it from an executor of machine instructions into an executor of commands by human users. It is primarily with this large amount of software that this text is concerned. Along the way of course it is necessary to understand the various hardware structures that exist in support of the software.

Although assembly language is a useful vehicle for explaining the details of hardware structure, it is not convenient for writing software. For writing well-structured, reliable software it is not at all suitable.

A class of high-level languages called *systems languages* are more appropriate for writing large programs. If necessary, assembly language can be used in critical parts of the software. Systems languages allow precise control over the hardware while encouraging the programmer to use well-structured design in the programs. Most newer software systems are now written in systems languages.

As we proceed in the study of hardware and software structures, there will be much preoccupation with detail, especially with regard to our example system, the PDP-11. The reader should try to remember that this detail will vary somewhat from machine to machine and that our primary purpose is to study fundamental concepts. Computing is a subject in which one can learn concepts well only by doing a lot of computing, and for this one needs an example system.

## 0-1 ALGORITHMS AND LANGUAGES

Computers exist primarily to facilitate the construction and execution of algorithms. The study of computers is then largely a study of algorithms and of the machine structures that support their implementation. An *algorithm* is a precise set of unambiguous instructions that can be followed and carried out by some execution mechanism (a computer) such that the execution eventually terminates. The language used to express the instructions of an algorithm will vary with the intended use. In our case we wish to describe algorithms for the purpose of reading and understanding them. It follows, then, that some language close to English will be most comfortable for doing this. Unfortunately, English itself is a very ambiguous medium for communicating precise meaning. On the other hand, the language used to describe instructions directly to a computer is far too detailed for easy reading. Besides, one objective here is to explain machine language, so we can hardly use it as a vehicle of explanation.

The compromise commonly adopted to informally explain algorithms is what can be called *structured English*: Ordinary English statements are imbedded in statements of a high-level programming language, and thus we obtain the readability of English with the precision of a programming language. On the assumption that the reader has been exposed to at least one high-level program-

ming language, the following chapters present algorithms based on a particular language, Pascal, without detailed explanation of the meaning of the statements involved. Pascal is a readable enough language that experience with programming in another language should yield sufficient understanding.

To illustrate structured English, here is a set of instructions for reading this text.

```
VAR i,j: INTEGER;
IF  you have no programming background THEN
    take a course in programming;
FOR i := 1 TO 18 DO BEGIN
    WHILE chapter i is not fully understood DO
        read chapter i;
    FOR j := 1 TO number of exercises in chapter i DO
        complete exercise i-j END
```

English statements are written in lowercase, as are program-variable names. In this program, *i* and *j* are declared in the first line to be integer variables. As can be seen, Pascal keywords are written in uppercase. The semicolon character (;) is used to separate statements to be executed sequentially. In this program, for example, there are, following the single declaration, two statements. There may appear to be more than two statements, but this is because a compound statement can be formed by enclosing a sequence of statements with a *BEGIN . . . END* pair of keywords. Such a compound statement can be used in a program wherever a simple statement can appear. The first statement starts with *IF* and the second with *FOR*. Notice that the indentation is carefully done to indicate the statement structure.

The first statement is a *conditional* with the meaning that the reader without a background of programming should obtain this background before attempting the material in the text. The semantics of this form of the *IF* statement are simply that the statement following the *THEN* is executed only if the condition following the *IF* is true.

The second statement is an *iteration* to be carried out 18 times. That is, the statement following the *DO* is carried out once for each one of a set of values to be assigned, in sequence, to *i*. The statement which is to be repeated is, in this example, a compound statement itself comprising two statements.

The first of these nested statements is an *indefinite iteration*, that is, a loop to be repeated an indefinite number of times depending on some specified condition. The *WHILE* is an indefinite iteration in which the looping condition is evaluated before each execution. Thus, if the condition is true, then the governed statement is executed; if it is false, looping terminates and execution proceeds to the statement following the *WHILE* statement. Obviously, execution of the statement governed by the *WHILE* had better affect the result of evaluating the condition or the iteration will never terminate!

The second nested statement is another *FOR* in which the number of iterations varies with the value of *i* determined in the outer loop.

Although there are many other kinds of statements in Pascal, this example



illustrates three fundamental kinds of control structures required to express algorithms: sequential execution, indefinite iteration, and conditional execution. In the case of conditional execution, we have actually shown only a special case of the general conditional structure, the IF . . . THEN . . . ELSE. To illustrate this general form, a slightly different set of instructions than those above for reading this text are

```
IF you have no programming background THEN
    take a programming course
ELSE proceed with this text as indicated above
```

The difference here is that the statement following the THEN is executed if the condition is true and the statement following the ELSE is executed if it is false. In either case control passes to the following statement if one is supplied. The reader without a background, therefore, does not get to read the text even if the required background is obtained!

It is not the intention to teach Pascal in detail here. Various features will be introduced and used throughout, and a later chapter will summarize the language. Enough information will be given to enable the reader to read the text, but if programs are to be written in Pascal then a language manual for the system in use is required.

## 0-2 ASSEMBLY LANGUAGE

Machine-level programming in assembly language is characterized by very simple, primitive operations. In general, much more effort is required of the assembly-language programmer to accomplish a task than would be required using a high-level language. More significant, however, is the fact that assembly language gives the programmer much less assistance in detecting errors and thereby increasing reliability. Consequently, much greater care and discipline are required in using assembly language.

Although later chapters explore assembly language in more detail, we can preview some of what is to come by looking at some simple cases. With the reader's appetite satisfied somewhat, we can then progress more systematically.

One of the primitive data types available in assembly language is the *character*. Much assembly-language programming deals with character manipulation. A character value is stored in a memory cell which can be given a name, much as a high-level language variable has a programmer-assigned name that is associated with its value. In the case of character storage let us just take as an illustration the following statement:

```
Q:  .ASCII  /?/
```

This statement specifies that there is to be a cell named Q containing the character value “question mark.” The cell could have been given a longer