THE OPEN UNIVERSITY
TM222 The digital computer
An Inter-faculty Second Level Course
Technology/Mathematics



HEKTOR USER MANUAL

TM222 Course Team

Dr P.I. Zorkoczy (Chairman)
Dr G. Alexander
Mr C.C. Bissell
Mr G. Hammond
Ms P. Higgins
Dr D.C. Ince
Ms A. Jones
Mr R. Loxton
Mr G. Martin
Dr W.S. Matheson
Mr M.A. Newton
Dr D.N. Pim
Mr P.D. Wilson

Support Staff

Mr G. Bellis
Mr R. Coles
Mr J. Garne
Mr D. Jones
Mr N. Stephenson
Mr M. Story

The Open University Press, Walton Hall, Milton Keynes, MK7 6AA
First published as the *PT502 HEKTOR User Manual*, 1981.
Revised and republished with present title, 1982. Second edition, 1983.
Reprinted 1984, 1986.
Copyright © 1982 The Open University.
All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from the publisher.
Designed and illustrated by the Graphic Design Studio of the Open University.
Filmset by Filmtype Services Limited, and printed in England by The Garden City Press Ltd, Letchworth, Herts SG6 1JS.

ISBN 0 335 17125 7

This text forms part of an Open University Second Level Course. For general availability of other material in this course, please write to: Open University Educational Enterprises, 12 Cofferidge Close, Stony Stratford, Milton Keynes, MK11 1BY, Great Britain.
Further information on Open University courses may be obtained from the Admissions Office, The Open University, P.O. Box 48, Milton Keynes, MK7 6AB, Great Britain.
2.3

CONTENTS

Study guide 6

PART I INTRODUCTION AND SET-UP

1 Introduction to the User Manual 8

2 Overview of the HEKTOR system 9

2.1 The structure of a computer 9
2.2 The HEKTOR microcomputer board 9
2.3 Inside the processor 10
2.3.1 The arithmetic/logic unit 11
2.3.2 The program counter 11
2.3.3 The control unit 12
2.4 Coding of instructions and data 12
2.5 Subroutines and the stack 13
2.6 Introduction to system software 15

3 HEKTOR system set-up 16

3.1 Parts checklist 16
3.1.1 Components supplied 16
3.1.2 The television set 16
3.1.3 The cassette recorder 16
3.1.4 Cassettes 16
3.1.5 Mains supply and plug 16
3.2 Handling precautions 18
3.3 Connecting HEKTOR components 18
3.3.1 Connecting a mains plug 18
3.3.2 Connecting the power-supply unit 18
3.3.3 Connecting the TV set 18
3.3.4 Connecting the cassette recorder 19
3.3.5 Connecting the peripheral board 19
3.4 HEKTOR self-testing 19
3.4.1 Switch-on tests 20
3.4.2 User RAM test 20
3.4.3 TV interface test 20
3.4.4 Keyboard test 20
3.4.5 Recording a test tape 20
3.4.6 Reading the test tape 20
3.4.7 Peripheral board tests 21
3.5 Hardware malfunctions 21
3.5.1 Missing or damaged components 21
3.5.2 Apparent loss of power 21
3.5.3 Memory faults 21
3.5.4 Unsatisfactory TV display 21
3.5.5 Difficulties with the cassette recorder 21
3.5.6 Keyboard faults 22
3.5.7 Peripheral board faults 22

PART II USING HEKTOR'S MAIN SOFTWARE FACILITIES

4 Using the monitor 24

4.1 The monitor structure 24
4.2 Monitor behaviour 24
4.3 Monitor command format 26
4.4 Monitor commands 27

4.4.1 Break-point command 27
4.4.2 Copy command 27
4.4.3 Editor entry command 27
4.4.4 Fill command 27
4.4.5 Go command 27
4.4.6 High-level language (BASIC) command 27
4.4.7 Load-from-tape command 28
4.4.8 Memory modify command 28
4.4.9 Print command 28
4.4.10 Query command 28
4.4.11 Rewind command 29
4.4.12 Save-on-tape command 29
4.4.13 Test command 29
4.4.14 Verify tape command 29
4.4.15 BASIC re-entry command 29
4.4.16 Register examine/modify command 29
4.4.17 Single-step command 30

5 Using the editor 31
5.1 The editor structure 31
5.2 Editor behaviour 32
5.3 Editor command format 33
5.4 Editor commands 34
5.4.1 Assembler entry command 34
5.4.2 Delete command 34
5.4.3 Edit line command 34
5.4.4 Insert command 34
5.4.5 'Kill' command 35
5.4.6 Load-from-tape command 35
5.4.7 Monitor entry command 35
5.4.8 Print command 35
5.4.9 Query command 36
5.4.10 Rewind command 36
5.4.11 Save-on-tape command 36
5.4.12 Verify tape command 36

6 8085 assembly language 37
6.1 Overview of assembly-language programming 37
6.2 Assembly-language statements 37
6.2.1 Comment lines 37
6.2.2 Assembly-language instructions 38
6.2.3 Assembler directives 38
6.3 Operand specification 39
6.3.1 Specifying 8-bit operand values 39
6.3.2 Specifying 16-bit operand values 40
6.3.3 Specifying operand addresses 40
6.3.4 Specifying registers 41
6.4 Opcode specification 42
6.4.1 Program counter (PC) 42
6.4.2 Stack pointer (SP) 42
6.4.3 Flag register (F) 46
6.4.4 Description of instruction set 46
6.5 Use of the assembler 52
6.5.1 User symbol table 52
6.5.2 Machine-code program 53
6.5.3 Program listing 53
6.5.4 Error messages 53
6.5.5 Assembler options 54

7 Using HEKTOR's BASIC language 55
7.1 Introduction 55
7.1.1 Short index of BASIC keywords 55

7.2 The BASIC command mode 55
7.2.1 Introduction: What is the BASIC command mode? 55
7.2.2 An introductory example 55
7.2.3 Storing and editing programs 56
7.2.4 Running programs 56
7.2.5 Debugging: Finding program errors 57
7.2.6 Format and character control 57
7.2.7 Saving and loading programs on cassette tape 58
7.3 Writing BASIC programs 58
7.3.1 The structure of a BASIC program 58
7.3.2 A line of BASIC 59
7.3.3 BASIC variables 59
7.3.4 BASIC operators and expressions 61
7.3.5 BASIC functions 61
7.3.6 Flow-of-control statements 62
7.3.7 Input/output instructions 63

PART III FURTHER TECHNICAL INFORMATION

8 HEKTOR system hardware 68
8.1 The microcomputer bus 68
8.2 The microprocessor 68
8.3 The memory subsystem 69
8.4 The keyboard interface 70
8.5 The TV interface 71
8.6 The cassette/serial line interface 73
8.7 The power supply 74
8.8 Connection to the bus 74
8.9 Connection to the serial line 74

9 HEKTOR system software 75
9.1 System data structure 75
9.2 Interrupt structure 75
9.3 I/O structure 78
9.4 TV handlers 78
9.4.1 TV output 78
9.4.2 Special messages 79
9.4.3 Output with hex conversion 79
9.5 Keyboard handlers 79
9.5.1 Accepting characters 79
9.5.2 Accepting a line 80
9.6 Cassette handlers 80
9.6.1 Saving data 81
9.6.2 Loading data 81
9.7 Serial line handlers 82
9.8 Processing utilities 82
9.9 Tune machine 83
9.10 SORT routine 84

APPENDICES
Appendix A: HEKTOR memory maps 86
Appendix B: Hexadecimal conversion tables 88
Appendix C: 8-bit ASCII codes 89
Appendix D: HEKTOR graphics symbols 90
Appendix E: HEKTOR BASIC error messages 91
Appendix F: 8085 operation codes 92
Appendix G: HEKTOR command lists 96

STUDY GUIDE

The HEKTOR User Manual is a reference manual, rather than part of the main teaching material of TM222. You will not need to read it cover-to-cover. Only those parts which are specifically referred to in the course units will be assessed. Moreover, parts of it go beyond the scope of TM222. You will be directed to those parts you will need at appropriate points in the course. However, you may find it interesting to browse through, as a self-contained description of a microcomputer, or as a second presentation of some material also covered in the course units.

The User Manual may serve various functions for you:

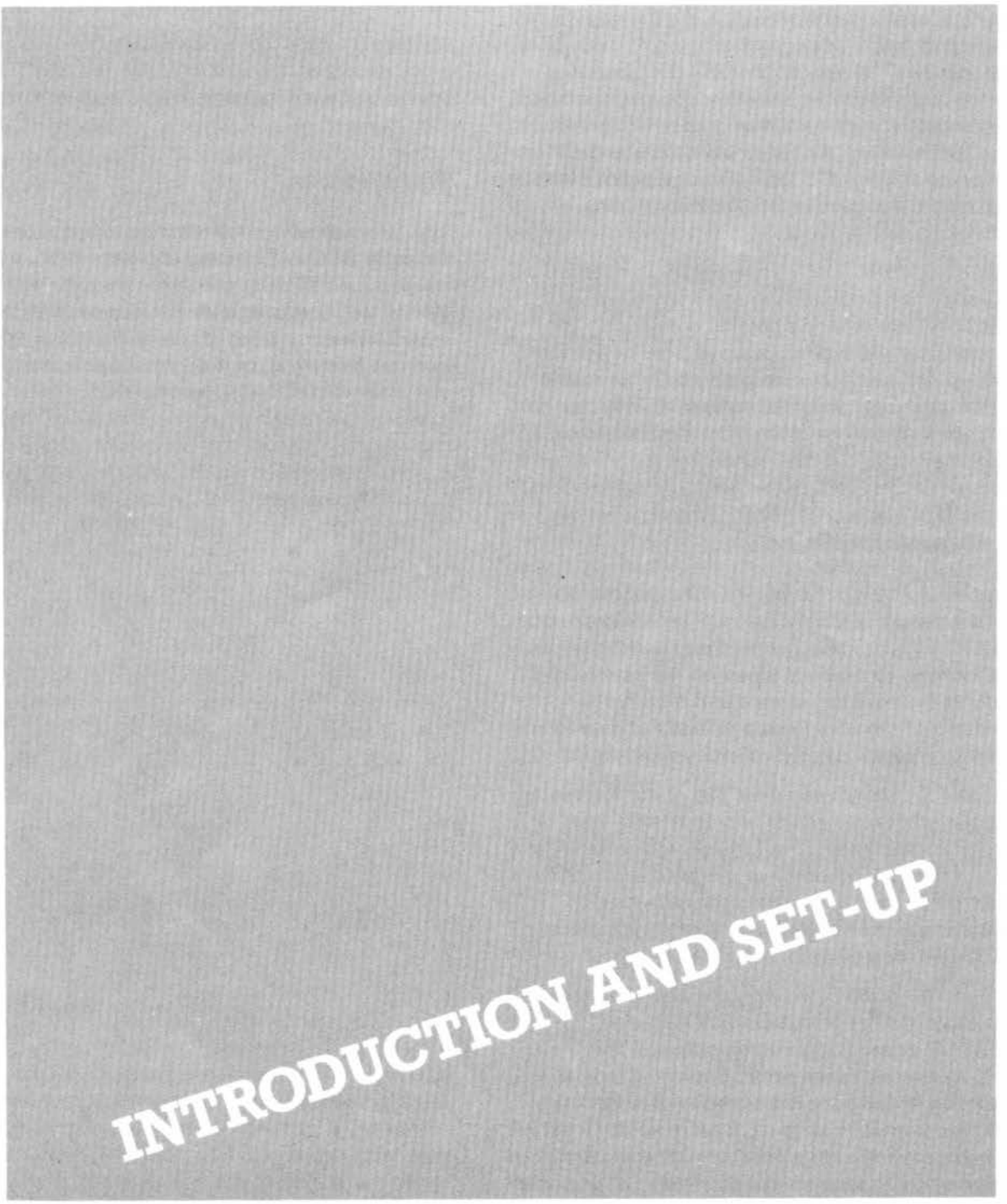
1 *Set-up and testing* — Section 3 will help you to connect the various parts of HEKTOR, and test that it is operating correctly.

2 *How to use HEKTOR* — Sections 4–7 describe the main facilities built into the machine. You will need to use these sections to learn to use HEKTOR and will then use them as reference.

3 *For reference* — The appendices contain conversion tables and summaries of HEKTOR's commands, which you may find useful.

4 *Further technical information* — Sections 2, 8 and 9 are optional reading for TM222.

P A R T I



1 INTRODUCTION TO THE USER MANUAL

Welcome to the HEKTOR microcomputer! This manual has been written to make your use of HEKTOR as easy as possible and to provide a description of its facilities.

The contents of this manual are divided into three parts – *Part I* Introduction and set-up; *Part II* Using HEKTOR's main software facilities; *Part III* Further technical information.

In more detail, the contents of this manual are:

Part I Introduction and set-up

Section 2 Overview of the HEKTOR system This section describes the structure and operation of computers in general, and relates this general structure to the particular components and operation of the HEKTOR microcomputer. There is also an introduction to HEKTOR's system software.

Section 3 HEKTOR system set-up In addition to information on connecting and setting-up your HEKTOR, this section contains handling precautions and a description of how to use the built-in self-test program to check for any system malfunction. There is also a subsection to aid diagnosis of, and recovery from, hardware faults.

Part II Using HEKTOR's main software facilities

Section 4 Using the monitor The monitor is a system program which controls the resources of HEKTOR (principally memory and interfaces) in response to commands typed on the keyboard. It enables the user to examine and modify the contents of memory locations, transfer data to and from tape cassettes, and to execute programs.

Section 5 Using the editor The editor is a system program that enables the user to handle lines of text. Its commands permit the user to store lines of text in memory, examine or modify them, and save them on or load them from tape cassettes. In particular, it is of use when writing programs in HEKTOR's assembly language.

Section 6 8085 assembly language This section describes the set of instructions built into HEKTOR's type 8085 microprocessor. The instructions are given both in machine language form (i.e. in hexadecimal code) and in assembly-language form. These descriptions will be a useful reference when you are programming in assembly language. The section also describes the use of the assembler system program, which translates programs from assembly language to machine language, ready for execution.

Section 7 Using HEKTOR's BASIC language This section gives a concise summary of the commands and facilities available when you write programs in BASIC on HEKTOR. It is not intended as a first introduction to BASIC, or as a textbook.

Part III Further technical information

Section 8 HEKTOR system hardware This section contains a more detailed description of the HEKTOR hardware than is contained in Section 2.

Section 9 HEKTOR system software This section gives details of the way in which the HEKTOR system software interacts with the hardware.

Appendices

The appendices contain memory maps, conversion tables of ASCII characters and hexadecimal numbers, HEKTOR's graphics symbols, HEKTOR BASIC error messages, and tables of all of HEKTOR's commands. These include the 8085 operation codes, monitor, editor and assembler commands, and BASIC keywords.

2 OVERVIEW OF THE HEKTOR SYSTEM

This section contains a review of the basic structure and principles of operation of digital computers. These general principles are illustrated with respect to the HEKTOR microcomputer system. The section also serves to introduce some of the terminology that will be used in the later sections of the manual.

2.1 The structure of a computer

The overall structure of a computer can be described in terms of five principal subsystems:

- The processor (or processing unit).
- The memory.
- The interfaces.
- The peripherals.
- The computer bus.

These subsystems are connected together electrically, as shown schematically in Figure 2.1. The *bus* is a common highway, to which the processor, interfaces, and memory are connected. It consists only of a set of electrical connections, but these are organized so as to allow the orderly communication of *data* between any of the other subsystems connected to it. The bus is important because it offers flexibility; extra memory or additional interfaces can be added to a computer simply by connecting them to the bus – the existing interconnections do not have to be modified.

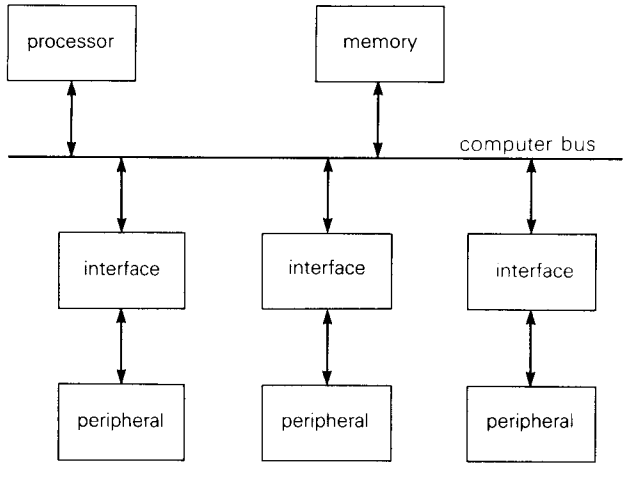


Figure 2.1 Computer structure

The *processor* organizes the fetching of data via the bus, performs operations on it, and despatches the results. It is directed in these tasks by a list of encoded *machine instructions* called a *program*. One of the jobs of the *memory* is to hold this list of instructions which the processor will fetch, interpret, and *execute*, one at a time. The memory also holds data that will be operated on by instructions, and the results of these operations can be stored in memory for later use. The memory, therefore, is used for several different purposes. It can retain data which is to be operated on by machine instructions, as well as the list of instructions itself. There is no distinction between the way in which data is stored and the way in which instructions are stored; it is the responsibility of the program designer to ensure that data in memory representing numbers to be arithmetically processed, say, is not treated by the processor as instructions in a program, and vice versa.

If the computer is to be generally useful, not only must it be able to process data held in memory, but it must also be able to communicate data to and from equipment external to the memory and processor. Various displays, knobs, switches, sensors and actuators can be controlled or interrogated by the processor. When connected to the computer system, these devices are called *peripherals*. Because the data they supply or expect to receive is seldom in the precise electrical form required for transmission via the bus, additional electronic circuits are required to provide the necessary conversions. These pieces of circuitry are connected between the bus and the peripherals as shown in Figure 2.1, and are called *interfaces*.

In summary, the processor (or processing unit) controls the operation of the computer, by a sequence of operations resulting from its execution of machine instructions held in memory. These operations involve processing data held in memory or supplied, via an appropriate interface, by an input peripheral. The data resulting from these processing operations can be returned to memory or sent to an output peripheral. All data transfers between the processor and memory or peripheral interfaces are via the computer bus.

2.2 The HEKTOR microcomputer board

The microcomputer board layout inside HEKTOR can be viewed by removing the lid in the top of the case.

Figure 2.2 is a plan view of the HEKTOR microcomputer *printed-circuit board*, showing the main components grouped into six subsystems. The principal peripherals are a *cassette recorder*, a *TV set*, and a *keyboard*.

Of these, only the keyboard is mounted on the HEKTOR board itself. In Figure 2.2, the keyboard is at the bottom. The line around the keyboard shows the area of the board that includes the keyboard and its interface. The keyboard interface consists primarily of the component labelled IC14, and it connects both to the individual keys on the keyboard and to

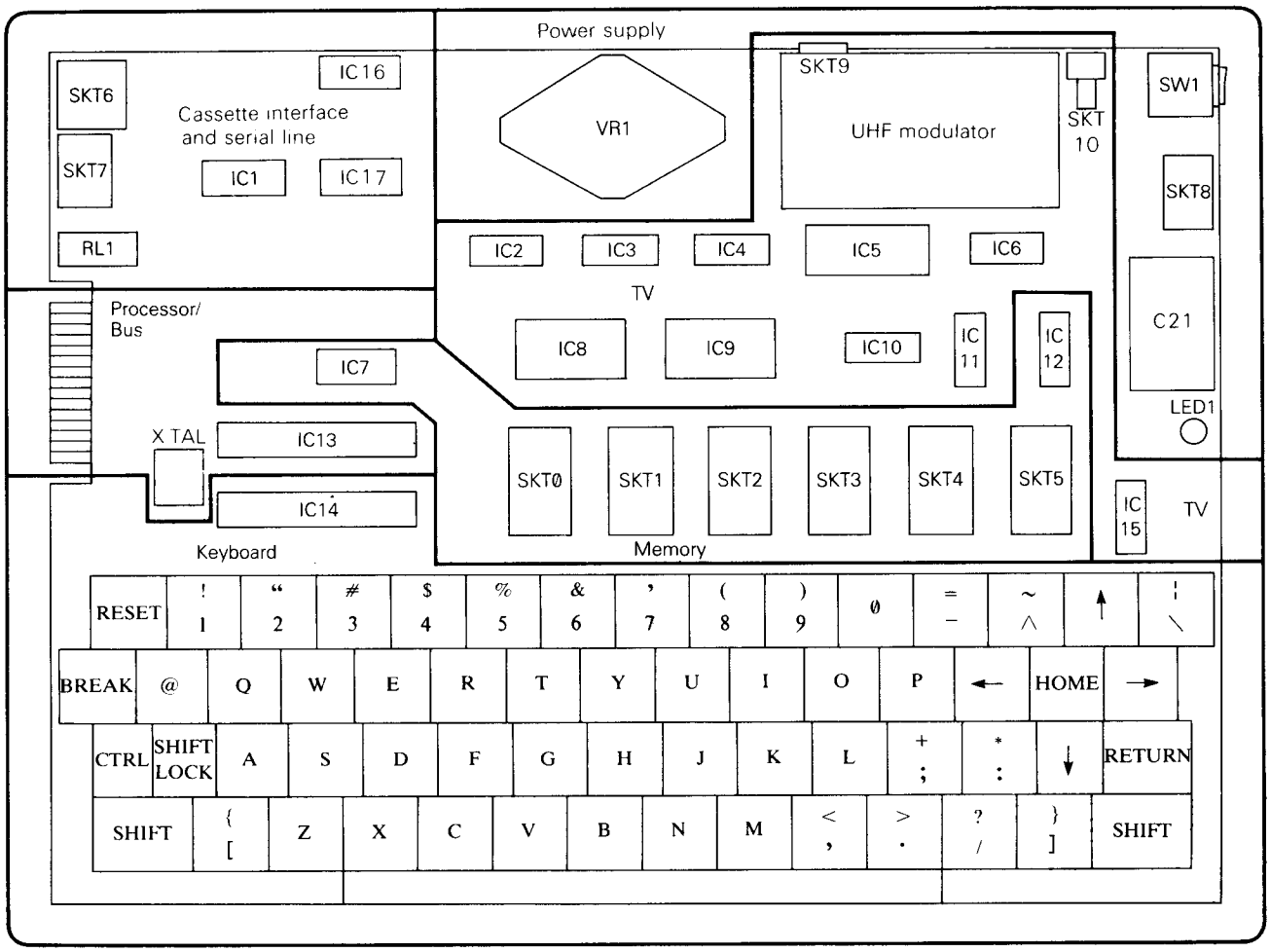


Figure 2.2 HEKTOR microcomputer board layout

the microcomputer *bus*, which consists of some of the copper printed-circuit tracks in the area of the board labelled 'processor/bus'.

The *processor* is IC13 in Figure 2.2, and it requires a few extra electronic components around it to enable it to operate. But most of this area of the board is taken up with the bus tracks which go to the *edge connector* (between the slots at the left-hand side of the processor/bus area). This edge connector enables peripheral interfaces or memory external to the main HEKTOR board to be connected to the microcomputer bus.

The *memory* subsystem includes IC7, IC12, and SKT0 to SKT5, in Figure 2.2.

The interface components for the *cassette recorder* are shown in the top left-hand corner. The recorder itself is connected to its interface by means of the socket labelled SKT7. There is also a general-purpose *serial line* interface in this area of the board, which enables a wide range of peripherals (such as printers) to be connected via SKT6.

Above the memory subsystem, in Figure 2.2, is the *TV interface*, which connects via SKT9 to the aerial socket of a standard UHF TV receiver.

The remaining components on the board, at the top and right, are a part of the *power supply* for HEKTOR. Mains power, transformed down to a lower voltage, connects to SKT8, from where it is converted into the stable D.C. electrical supply required by the electronic components on the HEKTOR board.

2.3 Inside the processor

Figure 2.3 is a simplified diagram of the internal structure of a processor (of which the 8085 microprocessor in HEKTOR is a typical example).

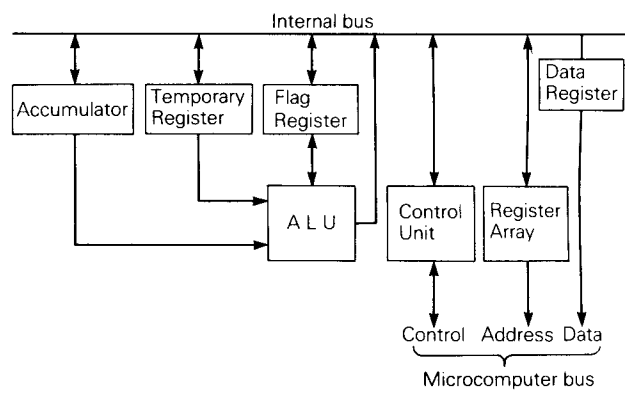


Figure 2.3 Structure of the processor

The processor contains four main groups of components:

- The registers.
- The arithmetic/logic unit (ALU).
- The control unit.
- The internal bus.

Of these components, the *internal bus* (like the microcomputer bus of the microcomputer as a whole) acts as a common intercommunication facility for the other components.

The *registers* hold data within the processor. Some hold data for a specific purpose related to the operation of the computer, and others are available for general use by the program designer as a small but convenient memory.

2.3.1 The arithmetic/logic unit

The arithmetic/logic unit is shown in Figure 2.3 as the *ALU*. It is the ALU that performs an actual processing operation on data, as directed by a machine instruction following its fetching into, and interpretation by, the control unit.

Suppose two data items, each representing numbers, are to be added together – a typical processing operation. One of the numbers to be added is assumed to be stored in the *accumulator* register; the second number may be in one of the general registers in the *register array*. (The machine instruction calling for the addition operation will specify which general register.) In interpreting this instruction, the control unit will first cause the second number to be copied into the *temporary register*, via the internal bus. The ALU will then perform the addition, usually returning the result (via the internal bus) to the accumulator, where it replaces the original number.

The result of the operation is available (in the accumulator), but it is also useful to have a summary of the features of this result. This is the purpose of the *flags* in the flag register. There is a flag (a simple yes/no indicator) which indicates whether the result was *zero* or not, another which indicates the *sign* of the result (positive or negative), and so on. Other machine instructions can test individual flags and behave accordingly. Thus it is possible to execute one sequence of instructions if the result of a previous addition is positive, and a different sequence if the result was negative.

The ALU can be instructed to perform a variety of arithmetic operations such as addition and subtraction, and logical operations such as AND and OR. There are, of course, separate machine instructions for each type of operation, and different processors have different *instruction sets*. But these simple operations available in the instruction set can be combined by the programmer so as to provide arbitrarily complex processing operations. (Multiplication, by repeated addition, is one example.)

2.3.2 The program counter

One of the registers in the register array of Figure 2.3 has a single, special purpose. It is called the *program counter*, and it holds information specifying the location in memory of the next machine instruction to be executed by the processor.

The memory consists of a collection of identical *memory locations*, each of which can hold a piece of data or part of a machine instruction. (In most microcomputers, a machine instruction occupies one, two or three adjacent locations.) Each location is identified by a unique number called its *address*, and consecutive locations in memory have consecutive addresses, as shown in Figure 2.4.

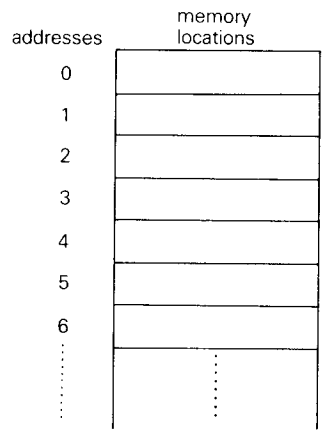


Figure 2.4 Memory locations and addresses

Suppose that the list of instructions forming a program is stored in an area of memory, and that the first instruction in the program (that is the one occupying the location with the lowest address) occupies the single location with address 100. In order to execute this program, the program counter (or *PC* register) must contain the address 100. (How this address can be stored in the PC will be discussed later.) The first action of the processor in executing this program, is to communicate the address in the PC to the memory subsystem, via the *address* part of the computer bus (see Figure 2.3). The memory responds by communicating the information stored in the addressed location back to the processor via the *data* part of the computer bus. This information is stored within the control unit of the processor as the machine instruction it is to execute. This *fetching* of an instruction is summarized schematically in Figure 2.5.

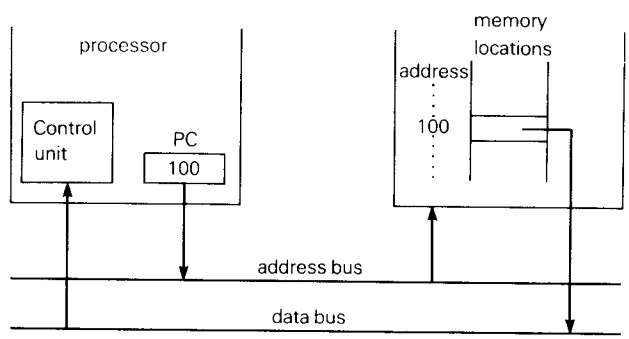


Figure 2.5 Fetching an instruction

The instruction is then ready for interpretation and *execution* by the control unit of the processor (possibly using its ALU). But the control unit, before executing the instruction, increases by one the number stored in the PC register. Then, having completed the instruction execution, the PC will be 'pointing' to the next location in memory, which in this example has the address 101 and contains the second instruction in the program. The second *fetch/execute* sequence can then begin automatically.

Some machine instructions may occupy two or three memory locations. In these cases, there are extra 'fetches', with the PC contents increased by one each time, before the 'execute' phase of the sequence.

For much of the time, the instructions in a program are fetched and executed in the order that they are stored in memory. But sometimes, the programmer will want to interfere with this natural sequence. (An example already mentioned is that of using the value of a flag to decide which of two sections of program should be executed next.) A simple instruction which enables the program sequencing to diverge from the natural order is the *jump* type of instruction. This instruction includes a memory address. When the instruction is executed, the address part of the instruction is simply stored in the PC register. Then, as the processor moves on automatically to the next fetch/execute sequence, the new instruction will be fetched from the location whose address was specified in the jump instruction.

2.3.3 The control unit

The control unit (see Figure 2.3) of the processor has already been mentioned in the context of the interpretation of machine instructions. The control unit issues the sequence of electronic signals which trigger the transfer of data of all types both inside and outside the processor. For example, within the processor, they cause the transfer of data from a general-purpose register to the temporary register. Other control signals define which operation, of the several available, is to be performed by the ALU during the execution of a processing instruction.

Similarly, the control unit issues control signal sequences on the *control* part of the computer bus (Figure 2.3), which enable properly timed data transfers between the processor and memory (or interfaces). This *timing* aspect of the control signals is achieved by the use of a *clock* signal, a regularly spaced sequence of electrical pulses or 'ticks'. Consider again the instruction fetch sequence of Figure 2.5. At one clock tick, the control unit will cause the PC's contents to be made available on the address bus. Clearly it will take some time before the memory can respond by placing the required machine instruction on the data bus. So the control unit will wait for one or more clock ticks before accepting the instruction from the data bus.

Besides timing this data transfer, the control unit will have to define the *direction* of data transfer. One of the signals making up the control bus indicates to the memory whether the addressed location is to have data *read* from it or *written* into it. That is, is the direction of data transfer to the processor from the memory, or vice versa?

Besides the control bus signals which allow for the orderly transfer of data via the bus, there is another group of signals, which enable events outside the computer to influence the normal sequence of program execution. These are called *interrupt* signals. When the control unit of the processor receives an interrupt signal, it suspends program execution (after completing the execution of the current instruction) and performs some special action, depending on which of several types of interrupt is received. Typically, the interrupt will be a request for attention from some peripheral device, and the effect of the interrupt is similar to that of a 'jump' instruction; that is, the processor starts executing a separate piece of program called an *interrupt handler* which will perform the operations required by the peripheral device.

A special type of interrupt is the *reset* signal. This signal is generated automatically when the computer is first switched on (and there also may be a reset switch that the computer operator can use). In HEKTOR, the reset signal sets the PC contents to zero, thus enabling the program which is stored in memory at address zero to be executed. This solves the problem of how to set the program counter to a known address, a necessary preliminary to allowing the automatic fetch/execute sequence of the processor to take control of the computer.

2.4 Coding of instructions and data

When information in a computer is transmitted from one place to another by means of electrical connections, the information is represented in the form of *binary voltages* on each of the connecting tracks or wires. That is, the voltage is either *high* (a few volts, usually) or *low* (zero volts). For a single wire, therefore, the transmitted data can have only one of two values. By convention, the symbols 1 and 0 are used to represent these two values. Two-valued data is sufficient for some purposes; for example, the *flag* data (Subsection 2.3.1), or the *read/write* control line (Subsection 2.3.3). But for most purposes, there is a need for a more extensive range of values; there are more than two addresses in the memory, for example.

If the combination of binary values carried by a *set* of connecting wires is used, the desired expansion of values occurs. In HEKTOR, for example, the data bus consists of eight wires, each of which carries a binary signal. There are, therefore, $2^8=256$ possible patterns of high and low voltages on this set of eight wires. Using the symbols 1 and 0 for the values on each wire, the range of patterns is from 00000000 to 11111111 on the *set* of eight wires. These symbolic patterns are called *binary codes*, and if there are eight binary values used, the code is an *8-bit binary code*. ('Bit' is a contraction of 'binary digit'.)

Not surprisingly, if the data bus carries N-bit codes, the memory also holds an N-bit code in each location. That is, each memory location consists of N binary storage cells, each of which can store a 1 or a 0. The value of N for a particular computer is called its *word length*, and data is stored, transmitted and processed as N-bit *words*. For the special case of 8-bit words (which includes HEKTOR and many other microcomputers), the term *byte* is used.

The *interpretation* placed on each of the 256 codes available with an 8-bit word length depends on the context. For example, the *unsigned integer numbers* from 0 to 255 are often represented by the binary codes from 00000000 to 11111111, respectively. With this interpretation, addition and subtraction can be performed by the processor, with the operands and the result all represented as binary codes.

But binary codes are required for other purposes. For example, *memory addresses* have to be specified. Now, in most microcomputers there is a need for more than 256 memory locations, so 8-bit codes are inadequate as memory addresses. In fact, 16-bit addresses are usually used, allowing up to $2^{16}=65536$ memory locations to be individually addressed. The *address bus* in HEKTOR therefore consists of sixteen wires, and the PC register is a *16-bit register*.

The *machine instructions* are also represented as binary codes. As was mentioned earlier, HEKTOR instructions are represented as one, two or three bytes. For all instructions, the first byte fetched from memory is the *operation code* for the instruction. For example, 10000000 is the binary code for one of HEKTOR's addition instructions, and 11000011 is a 'jump' operation code. The control unit of the processor decides, from the operation code, whether there are additional bytes to be fetched for the instruction in question.

If there are additional bytes in the instruction, they supply *operand* information. For example, the operand of the 'jump' instruction is an address in memory. This address is represented as a 16-bit code, which is split into two 8-bit codes when stored in memory as part of the jump instruction. Therefore, the jump instruction is a three-byte instruction: one byte of operation code, and two bytes of operand information.

The operand may be a 16-bit address, as for the jump instruction, or it may represent other information. A single-byte operand may, for example, represent a number to be added to the data in the processor's accumulator. The versatility of a processor's instruction set depends not only on the range of basic operations available, but also on the variety of ways in which operand information can be specified in instructions.

2.5 Subroutines and the stack

In Subsection 2.3.3, the concept of *interrupts* was introduced. When the processor is interrupted, it suspends the execution of the current program, in order to execute a separate piece of program in response to the interrupt – the interrupt service routine. Having executed this routine, what happens next? Clearly, the programmer would like the execution of the original program to continue from the point at which it was interrupted. For this to be possible, the computer must 'make a note' of the address of the next instruction to be executed in the original program, before starting execution of the interrupt service routine. Having completed the service routine, the processor can then use this saved address information to resume execution of the original program.

This facility of being able to break off from the normal sequence of a program, in order to perform some subsidiary task and then return to the main program, is a very powerful one, not simply restricted to interrupt handling. Suppose that there is a task that has to be performed several times, at different points throughout a program. The list of instructions which perform this task can be stored in memory as a *subroutine*, separate from the main program. Then whenever this task is required in the main program, the machine instruction which *calls* this subroutine is inserted (instead of the list of instructions making up the subroutine). The 'call' instruction is similar to the 'jump' instruction, in that it loads the program counter with the address specified by the operand part of the instruction. But, in addition, the 'call' instruction stores the old program counter's contents in memory; that is, it 'makes a note' of where to return to in the main program after the subroutine is executed. The subroutine is then automatically executed. Its last instruction is a 'return' instruction, whose function is to restore the program counter's contents so as to enable the main program to be resumed.

The return instruction has no operand – the return address is whatever was stored when the previous

matching call instruction was executed. Therefore the subroutine can be called, as and when required, by the main program. This is shown in Figure 2.6. Further, subroutines can themselves call other subroutines, as shown in Figure 2.7, when they are said to be *nested*.

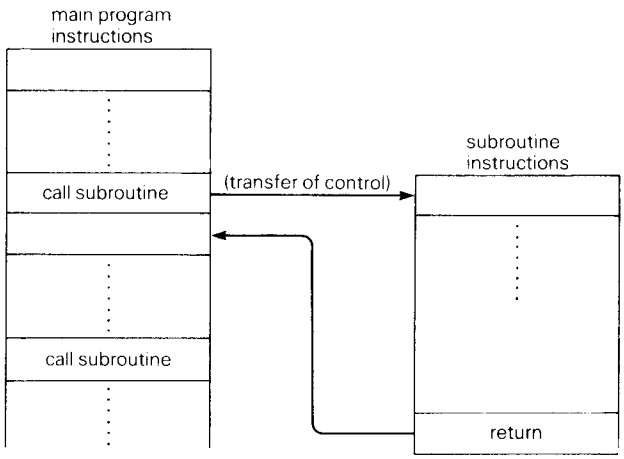


Figure 2.6 Execution of a subroutine

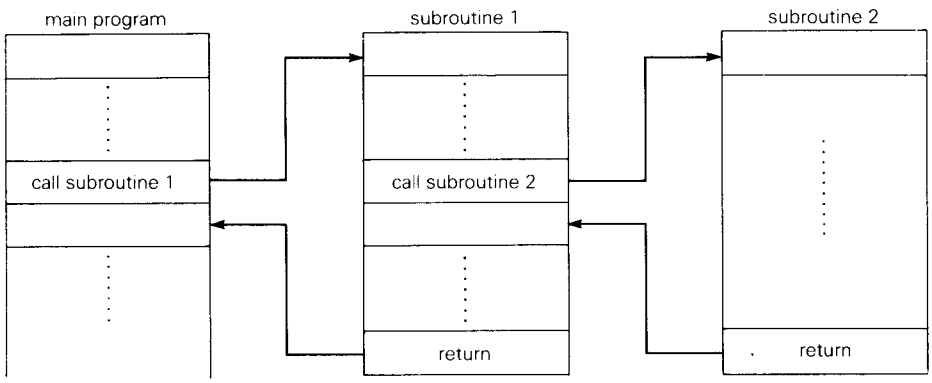


Figure 2.7 Nested subroutines

However, there is a problem. With possibly several return addresses stored in memory, depending on the 'depth' of the nested subroutines, how is a return instruction to locate the appropriate one to 'return' to? The usual method for the orderly storage and retrieval of return addresses involves an area of memory called the *stack*. This area is not different from any other area of memory – it consists of storage locations, each with its own address. But the *method of access* to this area is different. There is a special register in the processor called the *stack pointer* which contains a memory address, and access to the stack area is made only by reference to the stack pointer. The address currently in the stack pointer is called the *top of stack*. Figure 2.8 shows the situation where the stack contains four bytes of data (shaded), with the top of stack being the stack location which has the lowest address.

To store an additional byte on the stack (an operation called *pushing* data onto the stack), the address

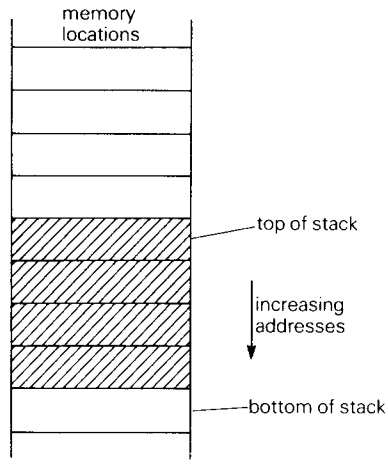


Figure 2.8 The stack

in the stack pointer is first reduced by one, to point to the unused location just above the stack. The data byte is then stored in this location, defining a new top of stack. This situation is shown in Figure 2.9. To retrieve a byte from the stack, the reverse operation is performed. That is, the required byte is read from the location pointed to by the stack pointer, and the

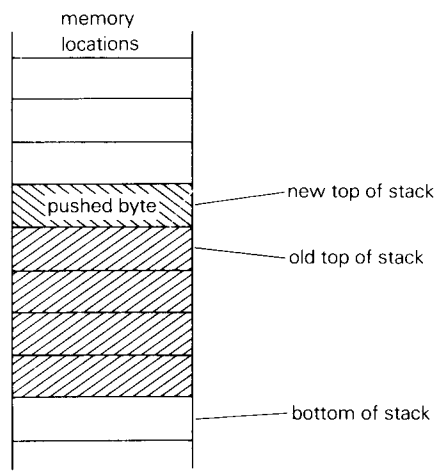


Figure 2.9 Pushing data onto the stack

address in the stack pointer is increased by one. This operation is called *popping* data off the stack. Note that after popping a byte off the stack, the location which contained it is now considered to be an 'unused' location above the stack – subsequent

pushes will overwrite whatever data was in that location.

This symmetric pair of push/pop operations is just what is required for the storage and retrieval of return addresses in the nested calling of subroutines. When a subroutine is called, the current PC contents (the return address) are pushed onto the stack as two bytes. The return instruction simply pops the address off the stack and into the PC register to effect the required return. Note that the 'last-in/first-out' nature of the stack ensures that the successive pushes of nested subroutine calls can be followed by the successive pops of their return instructions, supplying return addresses in the right order.

2.6 Introduction to system software

System software is the name given to the programs, subroutines, interrupt service routines, etc., which are supplied with the hardware of the computer. The 'naked' hardware is of little use; consider what happens when the computer is switched on. In the case of HEKTOR, the *reset* control line will be activated, which simply causes the processor to execute the 'program' stored in memory beginning at address zero (Subsection 2.3.3). If there is no meaningful program stored there, the processor will not perform any useful operations. (It will not simply 'stop' however; it will treat the random data as 'instructions', and operate randomly as a result.)

HEKTOR has been designed so that an area of memory starting at address zero contains a useful program, called the *monitor*. The electronic memory devices used for this area are *read-only* devices (*ROM*), which means that their stored contents cannot be changed even when the machine is switched off. Therefore, the monitor program is always executed when HEKTOR is switched on. This program contains subroutines which enable the keyboard and TV peripherals to be used for the communication of data between a human user and the microcomputer. For example, the machine instructions of a *user program* can be keyed in on the keyboad, and the monitor program will store them in memory.

The monitor enables a human user to use the keyboard and TV screen for a variety of functions related to the storing, examination, and step-by-step testing of user programs. These various functions are initiated by *commands* keyed in at the keyboard. Section 4 of this manual describes how to use the monitor. The monitor also performs a number of 'housekeeping' functions, such as defining an area of memory to be used as the stack (Subsection 2.5).

The system software of HEKTOR includes other programs besides the monitor. (There are monitor commands which cause these programs to be executed.) The *editor* program allows the user to describe programs in a more convenient way than as binary-coded instructions. Programs written in *assembly language* (described in Section 6) can be prepared, stored and modified by using the editor commands (Section 5).

The third major component of the system software is the *assembler* program, which translates an assembly language program description into its binary-coded equivalent, ready for execution.

The fourth major component is the *BASIC interpreter*. It enables you to write, correct and execute programs in the BASIC language. Finally, there is a group of *application* programs, which control particular types of external equipment connected to HEKTOR via its edge connector. One such piece of equipment is the HEKTOR peripheral board, with its switches and lights, etc.

It was mentioned, at the beginning of this subsection, that the system software resides in ROM memory, with addresses starting at zero. Clearly, some of HEKTOR's memory must be *read/write memory* (known as *RAM*), in order that user-defined programs can be stored. There will also be an area of memory used for the stack.

So some of memory is ROM, some is RAM, and for some valid memory addresses there is no actual physical memory installed! A useful way of summarizing what memory exists, what type it is, and how it is used by the system software, is to construct a *memory map*. An outline memory map is shown in Figure 2.10. Note that the size of each area is described using the symbol 'K'. 'K' is a convenient shorthand for the number 1024 (which is 2^{10}). There are therefore 2^{16}=64K different memory addresses. Note that some addresses are used to access peripheral devices (through their interfaces), rather than to address conventional memory locations.

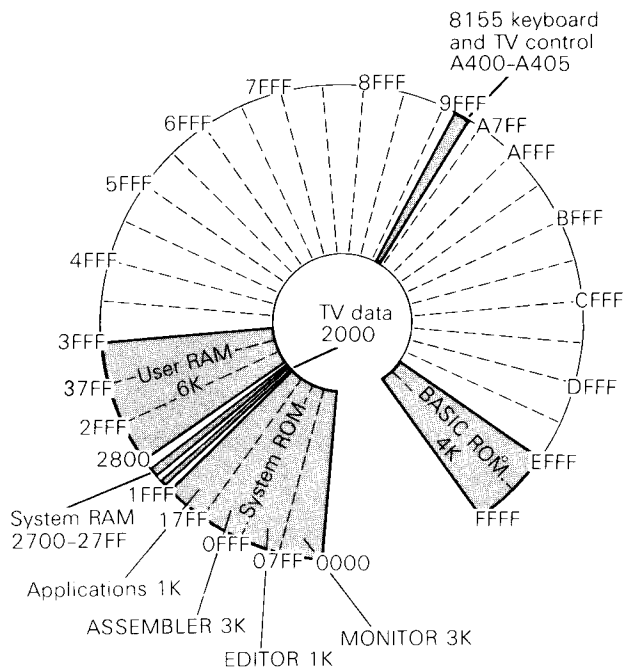


Figure 2.10 HEKTOR memory and I/O map

3 HEKTOR SYSTEM SET-UP

The purpose of this section is to describe the components of the HEKTOR system which are supplied within the package, and those which you will have to provide yourself. The interconnection of these components so as to enable HEKTOR to operate is also described, together with some initial adjustment procedures.

The section then describes the HEKTOR self-test procedure, in which a system program checks the operation of each of HEKTOR's main electronic subsystems.

Finally, there is a subsection which is designed to help diagnose the cause of hardware faults. These faults are often due to incorrect interconnection or adjustment of the equipment. If this is the case, you can remedy the fault yourself. If the fault is due to mechanical or electrical damage to HEKTOR, it may be returned, using the return slip enclosed with the equipment.

3.1 Parts checklist

The HEKTOR system consists of several component parts. Most are included in the package, but you will need to supply a standard UHF TV set and a mains plug for the HEKTOR power supply. You will also need to use the cassette recorder for data and program storage. Each of these components is described below.

3.1.1 Components supplied

In the package(s) you receive, you should have the following components:

- The HEKTOR microcomputer in its case.
- The HEKTOR power supply, consisting of a box with two cables protruding from it. One is a mains cable, and the other terminates in a 6-pin DIN plug.
- A cable for connecting HEKTOR to a TV set. It is a single cable with a coaxial connector on one end and a 'phono' type connector on the other end (see Figure 3.1).
- Cables for connecting HEKTOR to a cassette recorder. There is a single 5-pin DIN plug at one end and 3 plugs at the other end (see Figure 3.1).
- The HEKTOR peripheral board, which has 8 switches and lights on it and a flat ribbon cable attached to it.
- A cassette recorder.
- Cassette tapes. (These may be mailed separately.)

3.1.2 The television set

The user-supplied television set should meet the following requirements:

- It is able to accept standard British television signals that are broadcast in the UHF band to the 625-line standard.
- It has provision for the connection of an external aerial through a standard coaxial socket.
- It has controls for tuning over the UHF band.
- It has brightness and contrast controls.

Some television receivers may require adjustment of the horizontal stability or 'horizontal hold' to give a readable display. On some models, there is an external control for horizontal stability, but on others this control is only accessible through a hole in the cover, or with the cover off. *You should not operate the set yourself with the cover off*; a TV dealer will usually make the necessary simple adjustment (see Subsection 3.3.3).

Best results are achieved using the smaller black-and-white sets with a twelve to fourteen-inch screen. Larger sets may have to be positioned too far away from HEKTOR for convenient reading. As HEKTOR does not generate a colour display, a colour set is not needed, and its display may be poorer than that of a black-and-white set.

3.1.3 The cassette recorder

The cassette recorder provided should enable you to save your own programs on tape and also use pre-recorded programs we have sent you on cassette. If you have your own tape recorder you may use it if you wish. However, the variations in detail between makes of tape recorders means that there is no guarantee that you will be successful.

3.1.4 Cassettes

Of the cassette tapes sent to you, one or more will be blank, for use in storing your own programs. If you need more than these, the ideal cassettes are the 10 to 20-minute cassettes sold specifically for use with computers. However, most branded ferric cassettes are satisfactory, and once a particular cassette has been proven, it can be re-used with confidence many times. There is no need for long cassettes – C60 or shorter types are adequate. Avoid the cheapest cassettes, as they may not record satisfactorily.

3.1.5 Mains supply and plug

The mains plug you use must obviously match the mains power sockets available. The HEKTOR power supply is designed to operate from the usual 240 volt 50 Hz mains supply. Either the supply or the plug should be fitted with a 3 amp fuse. Note that to operate HEKTOR you will need *three* mains sockets: for HEKTOR itself, for the TV and for the cassette recorder. A multiple outlet or adapters may be useful.

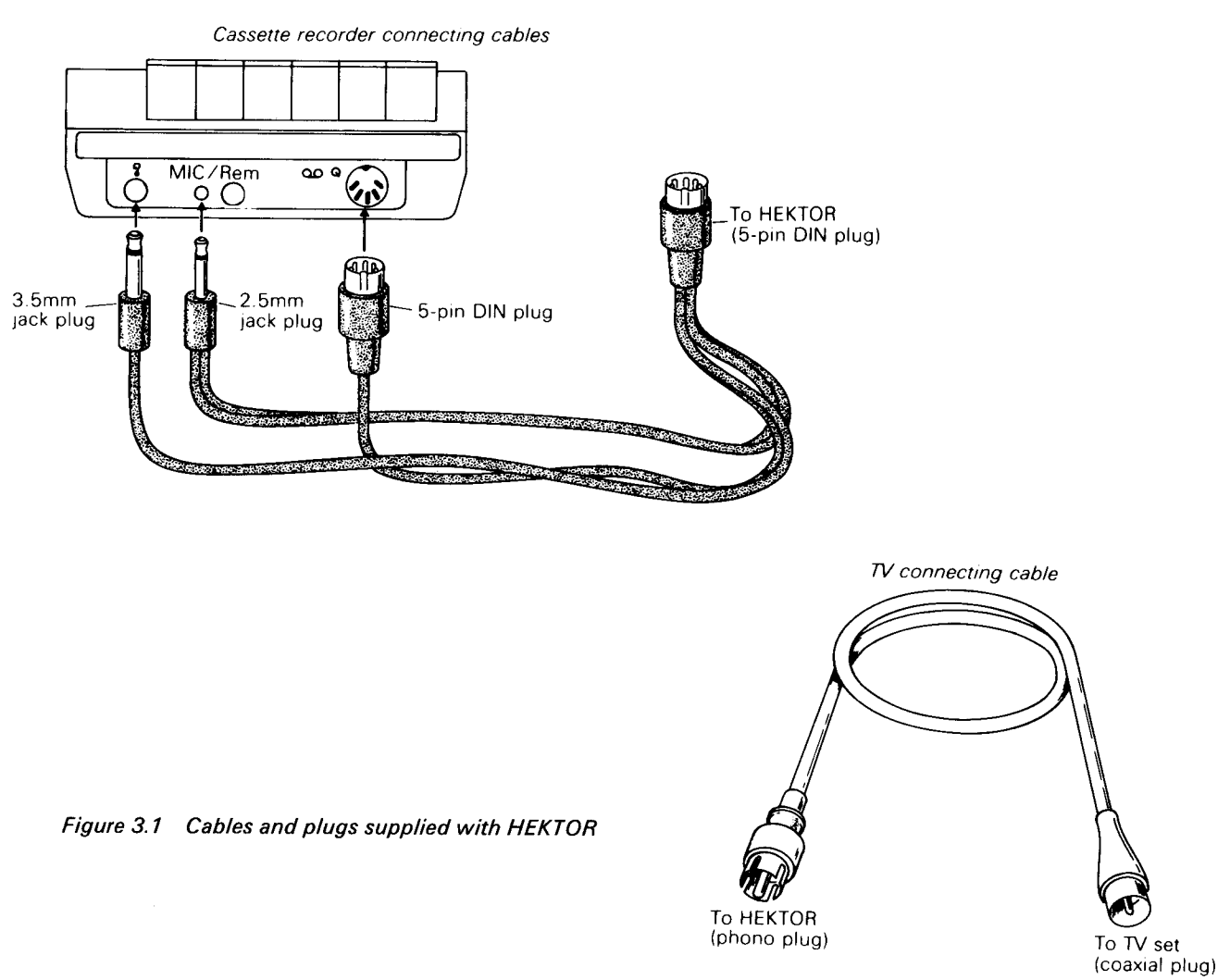


Figure 3.1 Cables and plugs supplied with HEKTOR

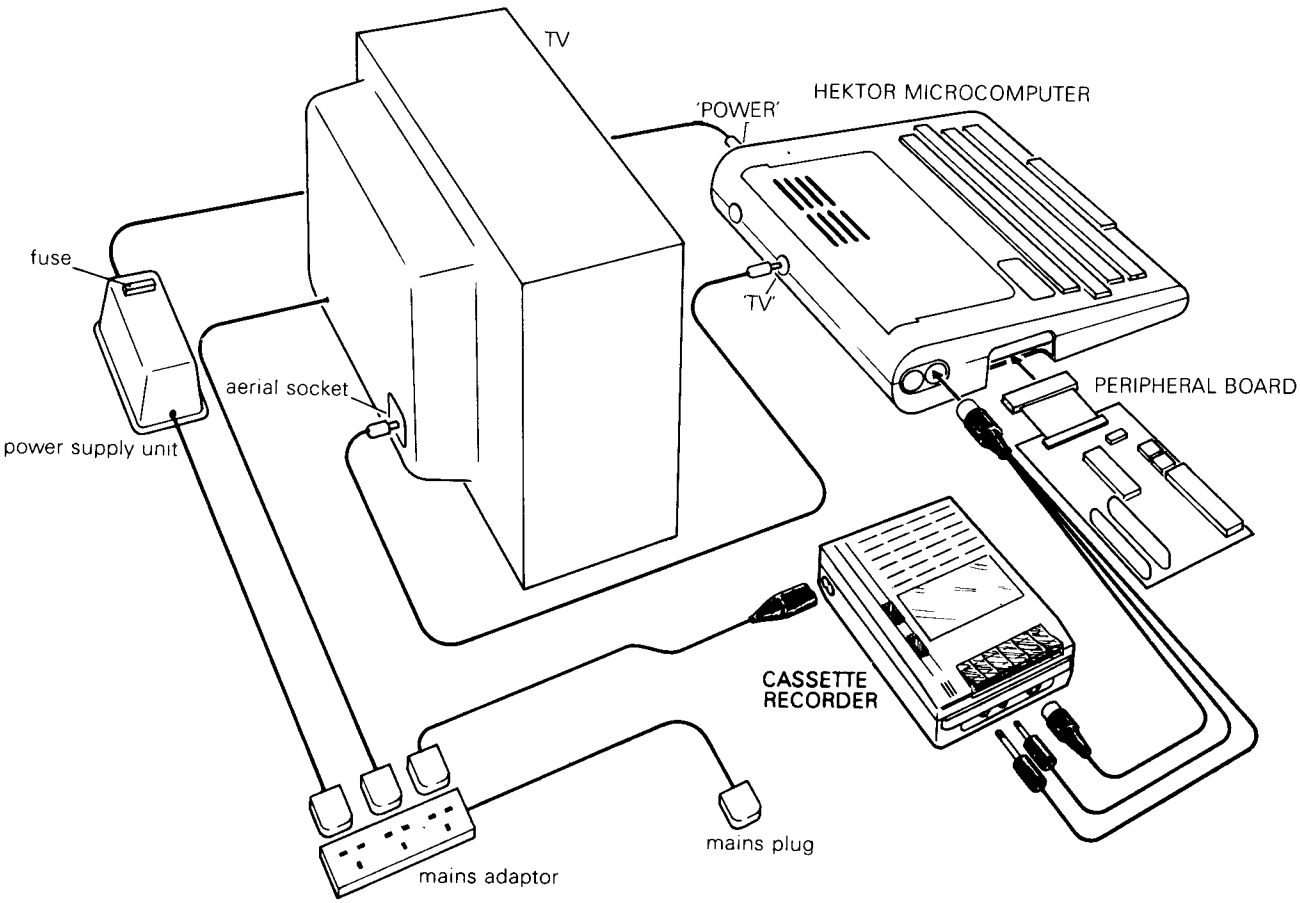


Figure 3.2 Connections for HEKTOR system components

3.2 Handling precautions

The power supply for the HEKTOR system is provided by an enclosed unit which contains a transformer. The transformer reduces the mains voltage and provides electrical isolation, so that HEKTOR's exposed parts operate at safe voltages. There is therefore no risk of electrical shock if these parts are touched. The top-rear panel of HEKTOR slides open, to enable you to examine its construction, but touching some of the electronic components, while in operation, can cause temporary malfunctions.

The HEKTOR microcomputer should stand on a flat surface, so that it is supported by its feet; otherwise pressure on the keys could damage the printed-circuit board. When the case is open keep metal objects away from the board to avoid accidental electrical shorts. Equally, moisture, such as spilt drinks, damp tablecloths, etc, can also cause at least temporary malfunction.

Some components in HEKTOR become warm in use, but not dangerously hot unless the normal heat dissipation in free air is grossly interfered with. So HEKTOR should not be covered whilst in operation.

3.3 Connecting HEKTOR components

This subsection explains how the various HEKTOR components are connected together. Figure 3.2 shows a suggested layout with suitable interconnections. (Note that the cassette recorder and the peripheral board need not be connected when they are not being used.)

Do not connect or disconnect the HEKTOR peripheral board while the power is switched on. Serious damage can result.

3.3.1 Connecting a mains plug

There are two wires in the mains cable lead from HEKTOR. These have to be connected to the pins of the plug as shown in Figure 3.3, as follows:

- The brown live wire to the terminal marked L.
- The blue neutral wire to the terminal marked N.

A 2-pin plug should not be used, and the mains socket or plug should be fitted with a 3 amp fuse.

3.3.2 Connecting the power-supply unit

Place the microcomputer on a flat surface, ensuring that there is plenty of room below it for air to circulate through the ventilating slots.

Plug the 6-pin DIN plug from the power-supply unit into the matching socket in the far right-hand corner of HEKTOR, next to the power switch. DO NOT FORCE THE PLUG. There is a matching key and

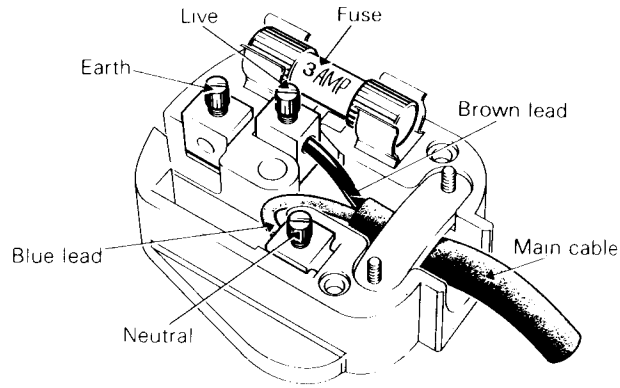


Figure 3.3 A correctly wired 3-pin plug

groove to prevent insertion with incorrect alignment. Then plug the power supply into the mains, and switch on both the HEKTOR power switch and the mains outlet switch. The red indicator on the right-hand side of the case should illuminate. If it does not, switch off immediately and consult Subsection 3.5.

Further checks can only be carried out with the TV set connected.

3.3.3 Connecting the TV set

Check that the TV you intend to use with HEKTOR will receive the normal broadcast signals. Switch off, and disconnect the usual aerial cable from the aerial socket on the TV. With HEKTOR also switched off, plug the coaxial plug (see Figure 3.1) on the TV cable supplied into the aerial socket on your TV, and plug the phono plug on the other end of the cable into the socket marked 'TV' on the back of the HEKTOR case. (The second socket on the case is for a TV monitor, not a conventional receiver.)

Turn the TV on and allow up to a minute for it to warm up. Then turn HEKTOR on, so that the red indicator on the HEKTOR case illuminates. Now adjust the tuning of the TV set until a stable pattern of white letters on a dark background appears. Adjust the tuning, brightness and contrast controls for the best picture.

HEKTOR contains facilities for producing sounds through the TV loudspeaker. The tuning which produces the best sound may be slightly different from that which produces the best picture. It is best to leave the TV tuned to the best picture (and the volume control turned completely down) except when actually using the sound generator.

If the picture does not stabilize, it may be necessary to adjust the horizontal hold control (see Subsection 3.1.2). However, first check that no other tuning position gives a better picture. HEKTOR is designed to give the best picture on Channel 36 (592.25 MHz), but other tuning positions will also produce a picture.

If you have a TV monitor, a higher-quality picture can be obtained by connecting the direct video coaxial socket to the TV monitor. This socket is labelled 'VIDEO'. However, if you use a TV monitor you will not have access to the sound generation facilities of HEKTOR.

3.3.4 Connecting the cassette recorder

First connect the appropriate one of the two 5-pin DIN plugs (Figure 3.1) to the DIN socket on the left-hand side of HEKTOR labelled 'TAPE'. The three remaining plugs on the cable go to the cassette recorder for (a) recording, (b) playback and (c) remote control of the motor. Use Figure 3.1 to identify the appropriate plug for each purpose, and then Figure 3.2 to identify the corresponding socket on the cassette recorder.

A quick test will indicate if the power connections for the cassette recorder are correct. Switch HEKTOR on, and switch on the mains supply to the recorder. Now press the PLAY switch on the recorder – nothing should happen. Now use HEKTOR's keyboard and press the [R] key followed by the [RETURN] key. The cassette recorder motor should now start, rotating one of the spindles which wind the cassette tape. If this does not happen, switch off and check the connections. If necessary, consult Subsection 3.5. Switch off HEKTOR.

3.3.5 Connecting the peripheral board

On the left-hand side of the HEKTOR microcomputer, there is an opening labelled 'peripheral board'. The forty connecting pads (twenty above and twenty underneath) within this opening enable external equipment to connect to the bus of the microcomputer (Section 2.1). The required connector is an edge connector attached to a forty-way flat ribbon cable.

To connect the HEKTOR peripheral board, proceed as follows. With HEKTOR switched off, gently ease the edge connector attached to the peripheral board between the slots on the microcomputer board, so that the edge connector's internal metal wipers make contact with the connecting pads on the microcomputer board. The edge connector is symmetrical, so it is physically possible to connect it upside down. This could cause severe damage. The correct orientation of the edge connector is obtained when there is no twist in the flat connecting ribbon cable and the peripheral board components (indicator lights, switches, etc.) are facing upwards.

A quick test will indicate if the edge connector's power connections are making good contact. Switch HEKTOR on. Press either or both of the buttons on the peripheral board marked 'PL' or 'PD'. The little red indicator lights (LEDs) on them should light up. If not, switch off, check the connections, and consult Subsection 3.5, if necessary.

3.4 HEKTOR self-testing

The system software of HEKTOR includes a self-testing program, which individually exercises each of the physical subsystems of HEKTOR, including the cassette recorder, and peripheral board (if fitted). This subsection describes how to use the keyboard and TV display to cause these tests to be executed in sequence.

Figure 3.4 shows the overall testing scheme. There are seven groups of tests and, once started up, the user can cause the next test group in the sequence to be performed by pressing the [BREAK] key. So the user need not repeat all tests in order, say, just to test the peripheral board. After starting the test sequence, four [BREAK] key pressings will bring the system to the peripheral board tests. Note also that the [RESET] key can be used at any stage to leave the self-test sequence.

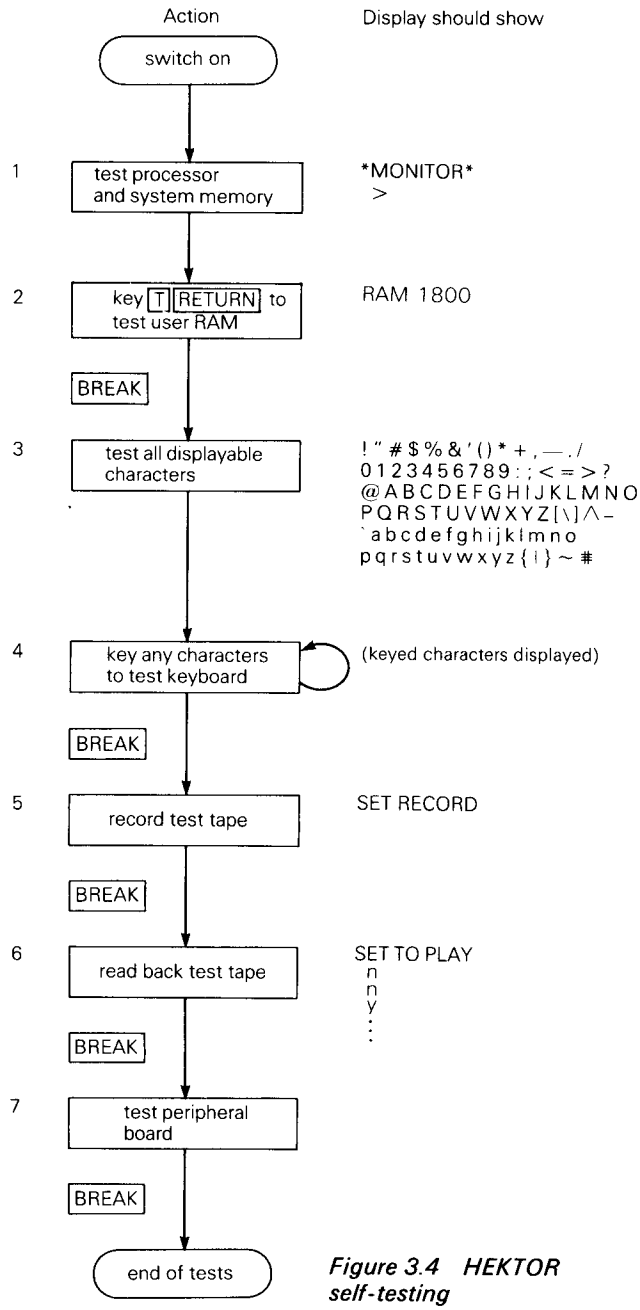


Figure 3.4 HEKTOR self-testing

Your HEKTOR system passed these tests before being sent to you, so any immediate faults revealed are likely to be due to damage in transit or (more likely!) due to the equipment being incorrectly connected or operated. If faults become apparent after a period of successful use, the self-test procedure, together with the diagnostic information in Subsection 3.5, will help you track down the cause.

You should refer to Figure 3.4 in the description of the individual tests which follows, in order to keep track of the sequence of tests.

3.4.1 Switch-on tests

When HEKTOR is switched on, a red indicator on the microcomputer should illuminate. This shows that electrical power is reaching the microcomputer.

If the display shows *MONITOR* after switch on, the system has already passed a number of automatic tests. There can be considerable confidence that the following subsystems are functional:

- Microprocessor.
- System RAM (stack, etc.).
- System ROM.
- Much of the TV interface.
- The TV itself.
- The power supply unit.

If no display occurs, consult Subsection 3.5. If the message includes RAM ERROR or ROM ERROR, there is a system memory fault (Subsection 3.5).

3.4.2 User RAM test

As shown in Figure 3.4 the special self-test sequence is initiated as follows. Following switch-on, or use of the [RESET] key, type [T] [RETURN] in response to the monitor *prompt* character '>'. (The monitor system program is described in Section 4.)

The first special test is to measure the amount of usable RAM memory, and display this number. The message displayed should therefore be RAM 1800. This number is in hexadecimal (Appendix B) and represents the amount of memory that HEKTOR finds to be correctly functioning for both reading and writing. There are 1800 (hex) bytes of user memory installed in the microcomputer board. Therefore, if the displayed number is less than 1800, there is likely to be a memory fault.

3.4.3 TV interface test

A full test of the TV interface and TV occurs after starting the self-test sequence, and having pressed the [BREAK] key once. HEKTOR attempts to display all the displayable characters, in the format shown in Figure 3.4. If this display occurs, the TV interface is fully functional.

3.4.4 Keyboard test

This test routine is entered automatically after the character display (Subsection 3.4.3). Any keys on the keyboard (except the special [BREAK], [RESET], and [CTRL] keys) can be used, and the key should be 'echoed' on the TV display. Any keys not appropriately echoed may indicate a keyboard fault (one of the more likely faults, as the keyboard includes mechanical as well as electronic components). Note the effects of the following *cursor movement* keys, which affect the position on the screen of the blinking *cursor*:

- [↑] – up one line.
- [↓] – down one line.
- [→] – right one position.
- [←] – left one position.
- [RETURN] – extreme left.
- [HOME] – top left, with clear screen.

3.4.5 Recording a test tape

Starting the self-test sequence, and keying [BREAK] twice, enables a special data pattern to be recorded on cassette tape. This cassette can then be used in the next test to adjust the cassette recorder's volume control to the most suitable position for future use of the recorder with HEKTOR.

The message 'SET RECORD' will be displayed. You should place a blank rewound cassette tape in the recorder, ensure that all connections are correctly made, and then set the recorder controls to record (that is, press both REC and PLAY buttons together).

Keying [RETURN] on the keyboard will now start the recorder and, after ten seconds, a repeated test data pattern will be recorded. After a minute or two, use the recorder controls to rewind the cassette (ready for reading). Then key [BREAK] to move on to the next test.

3.4.6 Reading the test tape

In this test, HEKTOR will endeavour to read the repeated short blocks of test data that have been recorded on cassette using the procedure in Subsection 3.4.5 above.

On entering this section of the test sequence, the display will show 'SET TO PLAY'. You should place the (rewound) test-data cassette in the recorder, set its controls to PLAY, and set the volume control to about half way (if you have no other information about the likely 'best' position). Set the tone control to half way also. Then key [RETURN]. The tape should start moving, and after the ten-second delay to allow for the blank leader, the letters 'y' or 'n' should start appearing on the screen as each block of data is played back. A 'y' indicates that HEKTOR has recognized the block as data, and the 'n' that it has failed to recognize it as correctly recorded data. If no

character at all appears, HEKTOR has not even recognized the recording as data. While the tape is running, adjust the recorder's volume control until the letter 'y' appears consistently. It is useful to mark the position where this occurs. Unless you change your brand of tape, this setting should be satisfactory for future use with HEKTOR. If your first few blocks of data produced 'n's, it is worthwhile repeating the test with the new volume control setting to see if they are recognized. (The first few blocks are the ones least likely to be reliably recorded.)

3.4.7 Peripheral board tests

This final test, entered after four [BREAK] key pressings in the self-test sequence, tests the individual hardware subsystems on HEKTOR's peripheral board. The tests should only be performed if the peripheral board is already connected to the microcomputer. (If not, *switch off* before connecting and run the entire test program again.)

There is no indication on the TV screen that this test has started, but some of the eight light-emitting diodes (LEDs) on the peripheral board might light up. The pattern of illumination of the LEDs should match the settings of the switches adjacent to them. Change the settings of some of the switches and then press and release the button marked 'PD'. The pattern of lighted LEDs should change to match the new switch settings.

Now press the button marked 'PL'. The pattern on the LEDs should change to its opposite: the lighted LEDs should go off, and vice versa.

This completes HEKTOR's self-test procedure. Keying [BREAK] one more time returns control to the monitor, which should respond with the prompt '>'.

3.5 Hardware malfunctions

The most likely cause of apparent malfunction of HEKTOR is that it is either not connected up correctly, or that its behaviour has been misunderstood by the user! So make sure that these causes are eliminated before going to the inconvenience of returning the equipment.

That having been said, hardware faults can occur. The most likely causes of failure are *mechanical*, rather than electrical. Connections may become worn or dirty, for example. The electronic components do not 'wear out', but electro-mechanical ones can – the keyboard switches, for example.

There can be *temporary* faults with the system. Besides the obvious example of a poor electrical contact, electrical interference may occur in extreme cases. Switching off heavy electrical equipment may generate mains-borne interference which could cause HEKTOR to apparently 'hang-up'. Again, try to eliminate these causes before deciding that HEKTOR has suffered a hardware fault.

In the following subsections, a number of possible fault conditions are discussed.

3.5.1 Missing or damaged components

If, when you first receive your HEKTOR system, there are any missing or obviously damaged components, you should return it.

3.5.2 Apparent loss of power

If the red LED indicator on the microcomputer board fails to illuminate when HEKTOR is switched on, first check that it is not simply the indicator; that is, does HEKTOR still provide a display, and otherwise appear to be working?

If not, check the mains socket; does another electrical appliance work using that socket? Next, try replacing the fuses. If the new fuse blows, or HEKTOR still fails to respond, return it for repair.

3.5.3 Memory faults

If any of the messages RAM ERROR, ROM0, 1 or 2 ERROR, appear after switch-on, HEKTOR has detected an apparent memory fault. The same applies if the memory self-test (Subsection 3.4.2) displays an unexpected number. This fault may be temporary, so switch off, check all connections, and try again. If the fault persists, it may be due to a socketed integrated-circuit package not seating properly. So *switch off*, slide open the panel on the top of the case, and press gently but firmly on the socketed components. Then try again. If the fault still persists, return the system for component replacement.

3.5.4 Unsatisfactory TV display

If you know that the TV is functional, but tuning slowly across its range produces no change in display (at normal brightness/contrast levels), then the TV is not connected properly, HEKTOR has lost power, or there is a hardware fault in HEKTOR.

If a picture can be tuned in, but it is a static but random patchwork of light and dark, try switching HEKTOR on and off a few times.

If a picture can be tuned in, but it appears to be skewed across the screen, adjust the horizontal stability (see Subsection 3.3.3), or try another TV set. This is not a hardware malfunction on the HEKTOR board.

3.5.5 Difficulties with the cassette recorder

Subsections 3.1.3 and 3.1.4 dealt with the use of the cassette recorder and cassette tapes, Subsection 3.3.4 with the recorder's connection to HEKTOR, and Subsections 3.4.5 and 3.4.6 with its setting up.

You can check if the test data has been recorded, by playing this test tape with the recorder disconnected from HEKTOR. It should sound like a scratchy buzz, with a rhythm that repeats about once per second.

Remember that faults in this area are much more likely to be due to the recorder, to poor contacts in its connection to HEKTOR, or to the cassette itself, rather than to an otherwise working HEKTOR system.

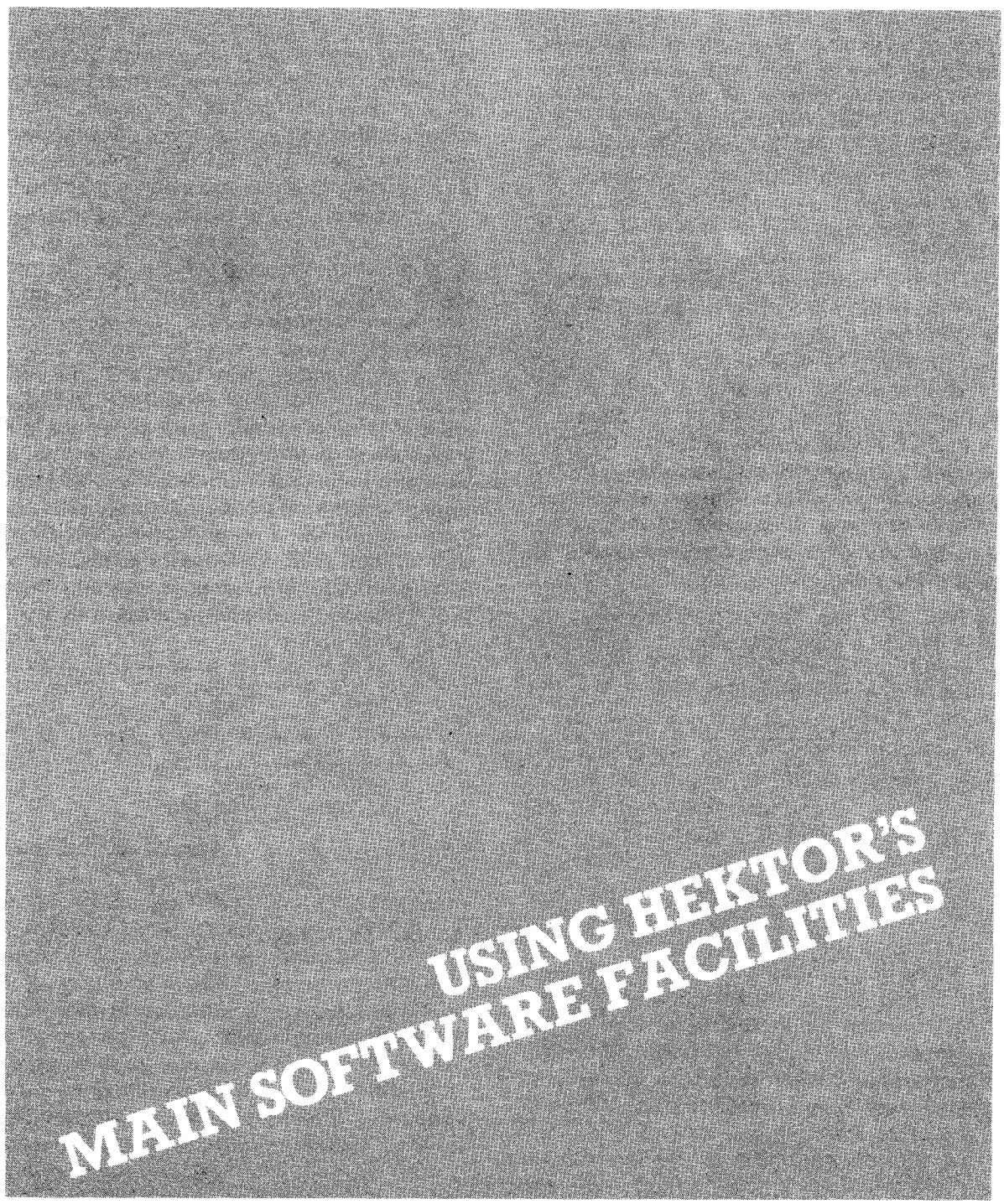
3.5.6 Keyboard faults

If you suspect keyboard faults, check out the keyboard using the test of Subsection 3.4.4. If you type too fast or press the keys too gently, they may not register with HEKTOR. But if you cannot get any response from just one or two keys, there may be a fault with the key mechanism and it can be replaced.

3.5.7 Peripheral board faults

The tests of Subsection 3.4.7 should reveal any faults on this board. The multiway edge connector is likely to be the prime suspect, so try wiping the connection pads on the microcomputer board with a clean, dry cloth (not a finger!). If some of the tests with the peripheral board work, but others do not, and you have eliminated setting-up or connection faults, then a hardware fault with this board is likely, and it should be returned for repair.

P A R T II



4 USING THE MONITOR

The monitor is a system program which provides a convenient interface between a human user and the HEKTOR microcomputer. The monitor controls the resources of HEKTOR (principally memory and interfaces), in response to *commands* typed on the keyboard. These commands enable the user to examine and modify the contents of memory locations, to save data on cassette and load data from cassette into memory, and to cause user-designed programs to be executed under controlled conditions which facilitate program testing.

Section 9 of this manual gives a technical description of the monitor, but in order to enable the best use of the monitor facilities to be made, an overview of the monitor's structure and behaviour is also given here in this section. Extensive use is made of *hexadecimal numbers* in monitor commands, for specifying memory addresses and data values (Appendix B contains tables for converting hexadecimal numbers to and from decimal numbers.) As an example of the interaction between the HEKTOR user and the monitor, consider the following task. Suppose the user wishes to know the contents of a particular memory location in HEKTOR. Without special equipment, the only way of acquiring this information is to cause HEKTOR to execute a program which will display the contents of the location on the TV display. Now the monitor is such a program, but one which is sufficiently general purpose that it will display the contents of *any* memory location, and perform a number of other, similarly useful, tasks as well.

As the monitor is general purpose, how does the user specify *which* memory location the monitor is to display the contents of? Indeed, how is the 'memory examine' task selected from among the range of monitor tasks? This is achieved by designing the monitor program so that it performs tasks in response to *commands* typed at the keyboard. In this example, the user will specify the 'memory examine' task by keying [M], and following it by the *address* of the memory location whose contents the monitor is to display. (This address is an *argument* of the M command.) On receipt of this command, the monitor will display the required information, and then await a further command.

This is shown schematically in Figure 4.1, where the participating components of HEKTOR are shown, together with the *data* (broad arrows) which passes between them. The monitor program (which starts executing automatically when HEKTOR is switched on) first signifies its readiness to accept a command by displaying the 'prompt' character, >, on the screen. It then awaits a *keyboard command*. In this case, the user types M135C, followed by the [RETURN] key. This is interpreted by the monitor as a 'memory examine' command, with reference to the memory location whose address is 135C (hexadecimal). As shown in Figure 4.1, the memory of HEKTOR consists of a mixture of ROM and RAM types of memory device. Each location in memory is identified by a unique four-digit hexadecimal code (its *address*), and contains a piece of data which is a two-digit hexadecimal code. The fixed codes in the ROM area of memory are the machine instructions making up the system software, including the monitor program itself, whereas the RAM locations contain variable data. Note that for some addresses, there is no physical memory installed; HEKTOR has about 12000 (denary) ROM locations, and about 6000 (denary) RAM locations, whereas there are 65536 (denary) addresses available with four-digit hexadecimal codes.

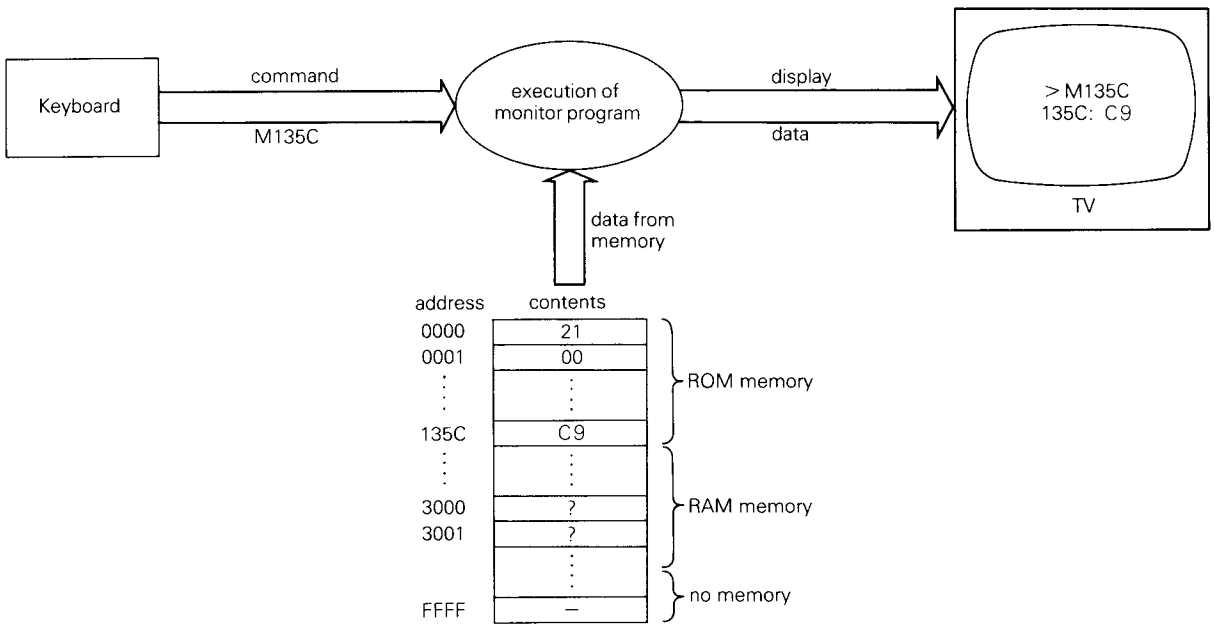


Figure 4.1 Example of a monitor command

In this example, therefore, the execution of the monitor program, after receiving the M135C command, causes the contents of the location with address 135C (hex) to be displayed on the TV screen. As this is a ROM location, its contents are fixed and happen to be C9 (hex).

In summary, Figure 4.1 is a diagram in which components of HEKTOR (memory and its contents, keyboard and display subsystems) are controlled during the execution of the monitor program so as to perform tasks in response to the users' keyboard commands. Note that the instructions making up the monitor program feature twice in the diagram. As codes are stored in memory, they are available for inspection, using the M command; when collectively *executed* by the microprocessor, they cause the behaviour described above.

4.1 The monitor structure

Figure 4.2 shows part of the HEKTOR *memory map* (described in Subsection 2.6), concentrating on the areas which are particularly relevant to the monitor. The monitor program itself can be considered as comprising four main sections stored together in the lowest-address area of ROM:

- command acceptance;
- command execution;
- utilities;
- interrupt handler.

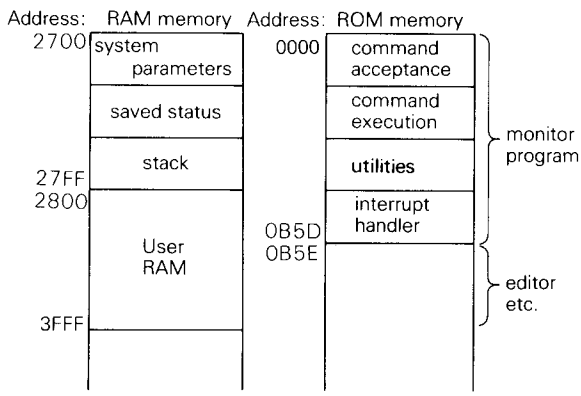


Figure 4.2 Monitor structure

Command acceptance is a section of the monitor which displays a 'prompt' message on the TV screen, and waits for a command to be typed in at the keyboard. When a valid monitor command is received, the appropriate part of the *command execution* section of the monitor takes over, to perform the requested action. The *utilities* are a set of subroutines which are frequently required by the monitor, other system programs, and indeed are available for use by user programs (see Section 9 for details). There are utility subroutines to detect when a key is pressed, to display a character on the TV screen, etc. Finally, the *interrupt handler* (introduced in Subsection 2.3) facilitates the testing of programs by enabling the user to interrupt the execution of a program in an orderly way to check its status.

The second area of memory of primary interest to the monitor user is the RAM memory, shown in Figure 4.2. Again, this can be considered as comprising four sections:

- system parameters;
- saved status;
- stack;
- user RAM.

The *system parameter* area is the RAM 'workspace' used by the monitor. The data stored here is not of great interest during normal monitor usage, but it can be modified by the advanced user to extend the versatility of the utility subroutines (see Section 9).

The *saved status* area is used to maintain a record of the state of the system when certain events occur. The 'status' of the system is effectively the contents of the microprocessor registers, including the program counter and stack pointer. When a program is *interrupted*, for example, if the register contents are saved, and then restored again after dealing with the interrupt, resumption of execution of the original program can occur as though no interrupt had occurred. The user can interrupt a program in a variety of ways, and use monitor commands to examine or modify the status of the system at the time of the interrupt, before allowing execution to continue.

The *stack* area of RAM is defined by the monitor when it is first executed following switch-on. That is, the *stack pointer* is initialized with the bottom-of-stack address (2800 hexadecimal), which allows a stack to form using locations with addresses less than this. In normal usage, therefore, the programmer need not be explicitly concerned with providing a stack area for user programs; the monitor-defined stack is adequate for most purposes.

Finally, the area of read/write memory with addresses in the range 2800–3FFF (hexadecimal) is not used by the monitor, and so is entirely available for *user* programs and data.

4.2 Monitor behaviour

Figure 4.3 shows schematically how the main sections of the monitor program work together, in response to external events such as keyboard commands, to achieve the various monitor functions. When HEKTOR is first switched on, or if the RESET key is pressed at any time, the monitor program starts executing. After initializing the system parameters (including definition of the stack area) the *command acceptance* routine is executed.

The command acceptance routine displays a '>' *prompt* character on the screen and waits for a valid

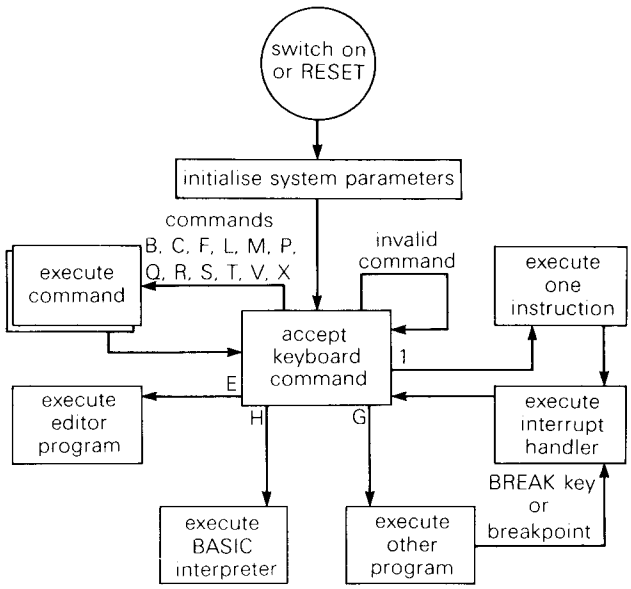


Figure 4.3 Monitor behaviour

monitor command to be keyed in. The format of these commands is discussed in detail later, but there are sixteen basic commands, each identified by a single character, as shown in Figure 4.3. An unrecognizable or invalid command causes 'ERROR' to be displayed, and another command to be awaited. The sixteen valid types of command split into two groups. The larger group (of twelve commands) cause the appropriate *command execution* routine to be performed, after which the command acceptance routine is re-entered automatically for the next command. The commands in this group involve operations such as the display and modification of data in memory, and the handling of data storage on cassette tape.

The four remaining commands each involve the execution of programs other than the monitor, as shown in Figure 4.3. In one case, E, the *editor* system program is executed (with its own set of keyboard commands, as described in Section 5). In the next case, H, the BASIC interpreter is executed (see Section 7). The two other commands, G and 1, enable any program stored in memory to be executed, whether it is a system program or a user program. These programs can be *interrupted*, and the processor status stored in the *saved status* area, after which the monitor is re-entered. In the case of the '1' command, the interrupt occurs automatically after one instruction is executed. In the case of the G command, the user can interrupt by using the BREAK key or by using the *break-point* facility (see Subsection 4.4.1).

As well as saving the processor register's contents in the saved status area, the interrupt handler also displays this data on the screen. A typical display is:

```
PC   SP   AF   BC   DE   HL   I
3104 27EC 1294 0000 AC01 380A 00B1
```

The interpretation of the display is that, at the time the interrupt occurred, the program counter (PC) contained the (hexadecimal) address 3104, the stack pointer (SP) contained the address 27EC, the accumulator (A) contained the byte 12 (hexadecimal), and the flag register (F) the byte 94. The six general-purpose registers (B, C, D, E, H, L) contained 00, 00, AC, 01, 38, 0A, respectively. Finally the interrupt mask register (I) contained B1 (even though this is displayed as 00B1).

Because the processor registers are loaded with the saved status information immediately before the G and 1 commands are executed, user programs can be tested easily. Program execution can be interrupted at any stage, the state of the processor and memory examined (using other monitor commands), and execution resumed, as though no interrupt had occurred.

4.3 Monitor command format

The command acceptance routine displays a prompt (the '>' character) and then awaits a monitor command. The command, as typed by the user, must conform to the general format:

Command character Argument list Terminator

The *command character* must be one of B, C, E, F, G, H, L, M, P, Q, R, S, T, V, W, X, or 1. This identifies the command type; for example, 'C' is the command character which causes data to be *copied* from one area of memory to another.

The *argument list* may be omitted for some commands, but if present, consists of up to three numbers separated by commas. The numbers are interpreted by the monitor as *hexadecimal* numbers (see Appendix B) and so the digits 0–9, A–F can be used. For example, the command C3000,300F,3100 is a copy command with three arguments. For this command the three numbers refer to 16-bit memory addresses, and the effect of the command is to copy the sixteen bytes of data stored in the area of memory whose addresses are 3000 to 300F inclusive, to the area of memory whose lowest address is 3100.

The arguments refer either to 16-bit or 8-bit codes, depending on the command, and the string of digits typed are interpreted according to whether an 8-bit or a 16-bit argument is expected. Normally, four hexadecimal digits are used to specify a 16-bit quantity, and two for an 8-bit quantity, but if *fewer* digits than required are given, leading zeros will be automatically implied. If more digits than necessary for an 8-bit argument are supplied, the leading digits will be ignored. (More than four digits will cause an ERROR display, in any case.) For example, C0,10,3000 is an acceptable equivalent to C0000,0010,3000, and F3000,3FFF,1234 will produce the same effect as F3000,3FFF,34.

The third element of a keyed-in command is the *terminator*. The command is accepted upon the receipt of a terminator, and the command executed

immediately (assuming that it is a valid command). The terminator can be supplied in any of four ways:

- the [RETURN] key;
- the [↑] key;
- the [↓] key;
- the [P] key, while the [CTRL] key is held down.

For most commands, any of these terminators can be used with no alteration to the effect of the command. (The [RETURN] key is probably easiest to use.) The [CTRL]–[P] terminator has a special effect only for the P command (see below), and the [↑] and [↓] terminators only for *subsidiary commands* associated with the M and X commands (see below).

Finally, typing errors can be corrected while typing in a command, by using the [←] key. Characters are 'echoed' on the TV screen as they are typed in, and the [←] key causes the previous character to be 'rubbed out'. Note that no further corrections to the command are possible after the terminator key is pressed.

4.4 Monitor commands

In the description of individual commands given below, the symbols <A1>, <A2>, etc., are used to denote hexadecimal arguments, and <T> to denote the use of one of the terminator characters listed in Subsection 4.3.

4.4.1 Break-point command

B <T> removes any break-point previously set; B <A> <T> sets a break-point at address <A>. A *break-point* can be inserted in a user program at the address of the operation code of any instruction. The break-point behaves as an interrupt. That is, when execution of the program reaches the break-point, execution is suspended (and the monitor re-entered) just as though the [BREAK] key has been pressed at that moment. Only one break-point can be inserted at a time; a second use of the B command will remove any previous break-point before inserting the new one. The break-point is automatically removed when the break occurs.

Because the insertion of a break-point involves the replacement of the byte at the specified address by a one-byte restart instruction, care must be taken to insert a break-point only at a location containing the operation code part of an instruction, and not one of its operand bytes. By combining the use of the B command with the use of the G and 1 commands, arbitrarily large or small segments of a user program can be executed, and the effects examined, segment by segment.

4.4.2 Copy command

C<A1>, <A2>, <A3> <T> copies data from one area of memory to another.

The data stored in the area of memory whose addresses run from <A1> to <A2> inclusive is copied, byte by byte, into an area of RAM memory of identical size, but whose lowest address is <A3>.

For example: C3800,3808,3801 'shifts' a block of nine bytes, stored in the area 3800–3808, along into the area 3801–3809. C3800,3808,37FF 'shifts' the block in the opposite direction.

4.4.3 Editor entry command

E<T> causes execution of the editor system program. The editor system program is described in Section 5.

4.4.4 Fill command

F <A1>, <A2>, <A3> <T> fills all locations in an area of memory with a particular byte.

All locations in an area of RAM memory, whose addresses run from <A1> to <A2> inclusive, will have the data byte <A3> stored in them. Note that <A1> and <A2> will be four-digit hexadecimal numbers, representing 16-bit addresses, whereas <A3> is a two-digit hexadecimal number, representing the 8-bit data.

For example: F2800,3FFF,00 will fill the whole of the user RAM area with the data byte 00.

4.4.5 Go command

G <T> causes execution of instructions starting at the address specified by the saved PC status.

G <A> <T> causes execution of instructions starting at <A>.

The saved processor status was described in Subsections 4.1 and 4.2. Execution of the G command involves setting all the processor registers to the values specified by the saved status. If an argument <A> is specified in the G command, it is this address which is loaded into the program counter register. The processor then begins executing instructions, starting with the one stored in the location now addressed by the program counter. Execution will continue automatically until interrupted by a break-point, the [BREAK] key, or the [RESET] key.

For example: G3000 will begin execution of a user program stored in RAM, beginning at address 3000. G1B38 will execute one of the application programs of HEKTOR's system software (stored in ROM).

4.4.6 High-level language (BASIC) command

H <T> causes execution of the BASIC interpreter. It will be followed by the display 'HEKTOR BASIC INTERPRETER', and the BASIC prompt '*'. HEKTOR will expect any subsequent commands to come from the BASIC command mode (see Section 7).

4.4.7 Load-from-tape command

L <T> causes data stored on cassette to be loaded into RAM memory. The immediate effect of this command is to cause the message 'SET TO PLAY' to be displayed on the screen. At this point, the user should ensure that the appropriate cassette (fully rewound) is placed in the properly connected cassette recorder, and the recorder controls set to the playback position. When this setting up of the cassette recorder has been completed, pressing the [RETURN] key causes HEKTOR to start the recorder and start searching for data recorded on the cassette. If data is found, it is loaded into memory (the addresses of the locations into which the data is to be loaded are also recorded on the cassette). Command execution then ceases with the power removed from the cassette recorder.

If data is found, but a tape-reading error was detected, the loading is aborted and 'ERROR' is displayed. (Data is recorded on tape by a method which enables errors to be detected.)

If no data is found, searching will continue indefinitely unless aborted by the use of the [BREAK] or [RESET] keys.

4.4.8 Memory modify command

M <A> <T> enables the contents of memory locations to be examined and modified.

This command displays the contents of the memory location whose address is <A>.

For example, if the byte 56 (hexadecimal) is stored in the location whose address is 3000 (hexadecimal), the command M3000 will cause the display: 3000: 56. After displaying the contents of the addressed location, the monitor re-executes its command acceptance routine, to accept a *subsidiary command* from the user. This subsidiary command is expected to be of the form: <T>, by itself; or <A> <T>.

If an argument is specified, this is interpreted as a replacement data byte for the addressed location, and the appropriate replacement is performed. What happens next depends on which terminator <T> was used in the subsidiary command. If the terminator is [RETURN] (or [CTRL]-[P]), this indicates the end of the whole M command. If the terminator is [↓] the contents of the location with the *next higher* address is displayed, and a new subsidiary command awaited, as before. If the terminator is [↑], the same thing happens, but with respect to the *next lower* address.

Thus, using a single M command, and a number of subsidiary commands, the user can scan up and down over an area of memory, examining the contents of individual locations and changing some of them if desired. An example of the use of the M command is given in Table 4.1, where the keys typed are shown on the left, the effects on the display are in the middle, and a comment is made on the right.

4.4.9 Print command

P <A1>, <A2> <T> displays the contents of the range of memory locations specified.

This command displays the contents of the locations in memory, whose addresses run from <A1> to <A2> inclusive. The display format is compact, with up to sixteen bytes per line, enabling the contents of up to 256 locations to all appear on the screen at once. Each line of data bytes is preceded by the address of the location of the first byte on that line.

For the P command only, if the [CTRL]-[P] terminator is used, the data is not displayed on the screen, but sent to the *serial line* interface instead. This is useful if a printer, for example, is connected to the serial line socket (see Sections 8 and 9 for details).

4.4.10 Query command

Q <A1>, <A2>, <A3> <T> searches an area of memory, seeking a match with a particular data byte.

<A3> is an 8-bit code, and the area of memory whose addresses run from <A1> to <A2> inclusive is searched in that order, seeking a stored byte to match <A3>.

Table 4.1 Use of the M command

Keys	*Display*	*Comment*
[M][3][0][0][0][RETURN]	>M3000 3000:56	Contents of addressed location = 56(hex)
[A][B][↓]	3000:56 AB 3001:00	Subsidiary command: contents become AB(hex); move to next location
[↓]	3001:00 3002:F3	No change to 00 byte; move to next location
[2][4][↑]	3002:F3 24 3001:00	Change F3 to 24; move back to previous location
[F][F][RETURN]	3001:00 FF >	Change from 00 to FF; exit from M command

If a match occurs, the search terminates with a display of the address of the first location containing <A3>. If no match occurs within the specified area, the search terminates with no address display.

4.4.11 Rewind command

R<T> connects power to the cassette recorder, to allow rewinding of cassettes.

The cassette recorder controls are normally disabled, as HEKTOR uses the 'REM' cable to switch off the power to the cassette recorder. Following the R command (as for the other cassette commands: L, S, V), the power is switched on to the recorder. In the case of the R command, however, no other action occurs; the user is free to use the recorder controls as desired. The main use of the R command is to *rewind* cassettes following the saving or loading of data.

After the desired rewinding has occurred, keying [RETURN] will remove the power from the recorder again, to complete the R command execution.

4.4.12 Save-on-tape command

S<A1>,<A2><T> causes the data in the specified area of memory to be saved on cassette tape.

The immediate effect of this command is to cause the message 'SET RECORD' to be displayed on the screen. At this point, the user should ensure that the fully rewound cassette which is to receive the data is placed in the properly connected cassette recorder, and that the recorder controls are set to the record position. When this setting up of the cassette recorder has been completed, pressing the [RETURN] key causes HEKTOR to start the recorder and send to it the data stored in the area of memory whose addresses run from <A1> to <A2> inclusive. When all the data has been sent, power is removed from the recorder and command execution is complete.

Details of the format of data storage on cassettes are given in Section 9, but a summary is as follows.

HEKTOR starts the tape moving, but waits ten seconds before recording data. This is to take account of the blank 'leader' tape in most cassettes. Then the addresses <A1>, <A2> are recorded to enable subsequent L commands to load the data into the appropriate area of memory automatically. The data bytes themselves are then recorded. Finally an error-detection code is recorded, called a *check sum*.

It is good practice, immediately after saving data on cassette, to *rewind* the cassette (using the R command) and then *verify* it (using the V command). In this way, any recording errors will be detected before the data which is to be saved is lost. The data can then be re-recorded if necessary.

4.4.13 Test command

T<T> causes HEKTOR to test each of its main subsystems.

These tests are described in Subsection 3.4.

4.4.14 Verify tape command

V<T> verifies that the data recorded on a cassette tape is capable of being loaded into memory.

Command execution for the verify command is very similar to that for the load-from-tape command (L). The only difference is that any data read from tape is not loaded into memory. The purpose of this command is to check that any subsequent L commands will have a high probability of success.

A successful verification is indicated by the display of the monitor prompt, and by the *absence* of any error messages.

4.4.15 BASIC re-entry

The BASIC re-entry command W<T> causes a transfer to the BASIC interpreter *without* deleting any BASIC programs previously stored. (This is called a 'warm start', hence the 'W'.) It is principally intended to help recovery from any error occurring while using BASIC which results in a return to the monitor. In such circumstances, returning to BASIC using the monitor's H command would delete any BASIC program stored.

4.4.16 Register examine/modify command

X<T> causes a display of the saved status, and enables modification of this data.

The saved status is described in Subsections 4.1 and 4.2.

The X command causes a display of the saved register contents in the same format as that which occurs following the interrupting of a user program.

After displaying the saved status, the display of the first register's contents (those of the PC register) is repeated, for example: PC: 3010.

The monitor then re-executes its command acceptance routine, to accept a *subsidiary command* from the user (as for the M command). This subsidiary command is expected to be of the form: <T>, by itself, or <A><T>.

If an argument is specified, this is interpreted as a 16-bit value, which replaces the old value in the saved program counter. What happens next depends on which terminator <T> was used in the subsidiary command. If the terminator is [RETURN] (or [CTRL]-[P]) this indicates the end of the whole X command. If the terminator is [↓], the contents of the *next* saved register are displayed, and a new subsidiary command awaited. (The next register after the

Table 4.2 Use of the X command

Keys used	*Effect on display*	*Comment*
[X] [RETURN]	>X PC SP AF BC DE HL I 3104 27EC 1294 0000 AC01 380A 00B1 PC: 3104	Display of saved status; repeat of contents of PC register.
[3] [1] [8] [8] [↓]	PC: 3104 3188 SP: 27EC	Subsidiary command; new contents are 3188; move to next saved register.
[↓]	SP: 27EC AF: 1294	No change to contents of SP register; move to AF register pair
[F] [F] [9] [4] [↑]	AF: 1294 FF94 SP: 27EC	Change saved A register contents from 12 to FF; no change to F register; move back to SP register.
[2] [7] [F] [E] [RETURN]	SP: 27EC 27FE >	Change saved SP value to 27FE; exit from X command.

PC is the stack pointer SP.) If the terminator is [↑], the same thing happens, but with respect to the *previous* register. Note that as there is no register previous to the PC, and no next register after the I register, the use of [↑] and [↓] respectively simply repeats the display of the current register's contents, in these cases.

Thus, using a single X command, and a number of subsidiary commands, the user can scan backwards and forwards across the saved status data, examining the contents of each, and changing some of them if desired. Note that the contents of the 8-bit registers (A, F, B, C, D, E, H, L, and I) cannot be individually changed; the argument <A> is treated as a 16-bit code (4-hexadecimal digits), which is to replace the saved contents of a *register pair*. For example, typing 12AB [↓] following the display, BC: 00FF, will change the saved B register's contents from 00 to 12 (hex) and the saved C register's contents from FF to AB (hex).

An example of the use of X command is given in Table 4.2, where the keys typed are shown on the left, the effects on the display are in the middle, and a comment is made on the right.

4.4.17 Single-step command

1 <T> causes execution of the single instruction stored at the address specified by the saved PC status.

1 <A> <T> causes execution of the single instruction stored at <A>.

Command execution for the single-step command is very similar to that for the go command (G). The only difference is that, for the 1 command, execution of the specified program is automatically interrupted after one instruction has been executed. (For the G command, execution continues until a user-supplied interrupt or a break-point is encountered.) The interrupt handler, as for other interrupts, causes the new saved status to be displayed, to round off the single-step operation.

5 USING THE EDITOR

The editor is a system program which acts as an interface between a human user and the HEKTOR microcomputer, like the monitor program. It, too, responds to *commands* typed on the keyboard. But, unlike the monitor, it deals with *lines of text*, rather than hexadecimal data. Lines of text stored in memory can be examined or modified, and saved on or loaded from cassette tape. If the lines of text represent an *assembly language* program (the main use for the editor), they can be translated by the *assembler program* (Section 6) into a program of machine instructions, which can then be executed using the testing facilities of the monitor program (Section 4).

Figure 5.1 is an example of the use of the editor to store lines of text in the text buffer. The I command (for 'insert') is used in response to the editor's prompt character '#'. The subsequent characters keyed in are not only echoed on the screen, but are stored in the *text buffer* as a line of text. The command I0 means 'insert after line number zero', and so the line of text is line number 1. The RETURN key signifies the end of line, and the editor is then ready to accept a second line, and so on. Other editor commands enable the user to manipulate whole lines of text, by referring to them by their line numbers.

The individual characters are stored in the text buffer, using one RAM location per character, with the ASCII coding scheme (Appendix C) being used to represent characters as two-digit hexadecimal codes. This is the main difference between the editor and the monitor. The actual hexadecimal codes and the actual memory addresses used for the storage of character data do not have to be known to the editor user; the editor program performs all the necessary conversions and allocation of storage. The user is concerned only with lines of characters and the associated line numbers (which, for added convenience, are *decimal* numbers).

5.1 The editor structure

Figure 5.2 shows part of the HEKTOR *memory map* (described in Subsection 2.6), concentrating on the areas which are particularly relevant to the editor.

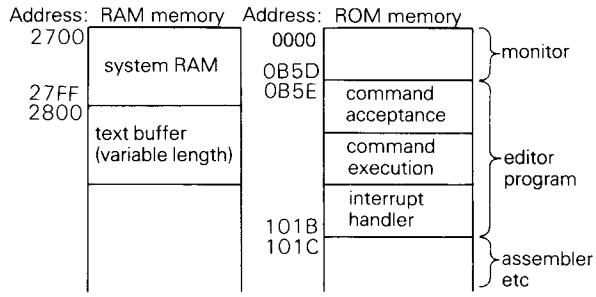


Figure 5.2 Editor structure

The editor program consists of three main sections, stored together in the area of ROM just 'above' the monitor program:

- command acceptance;
- command execution;
- interrupt handler.

Command acceptance is a section of the editor which displays the editor 'prompt' message on the TV screen, and waits for a command to be typed in at the keyboard. When a valid editor command is received, the appropriate part of the *command execution* section of the editor is activated, and it performs the requested action. The *interrupt handler* (introduced in Subsection 2.3) enables the BREAK interrupt key to be used to return to the command acceptance routine; it is therefore a different routine

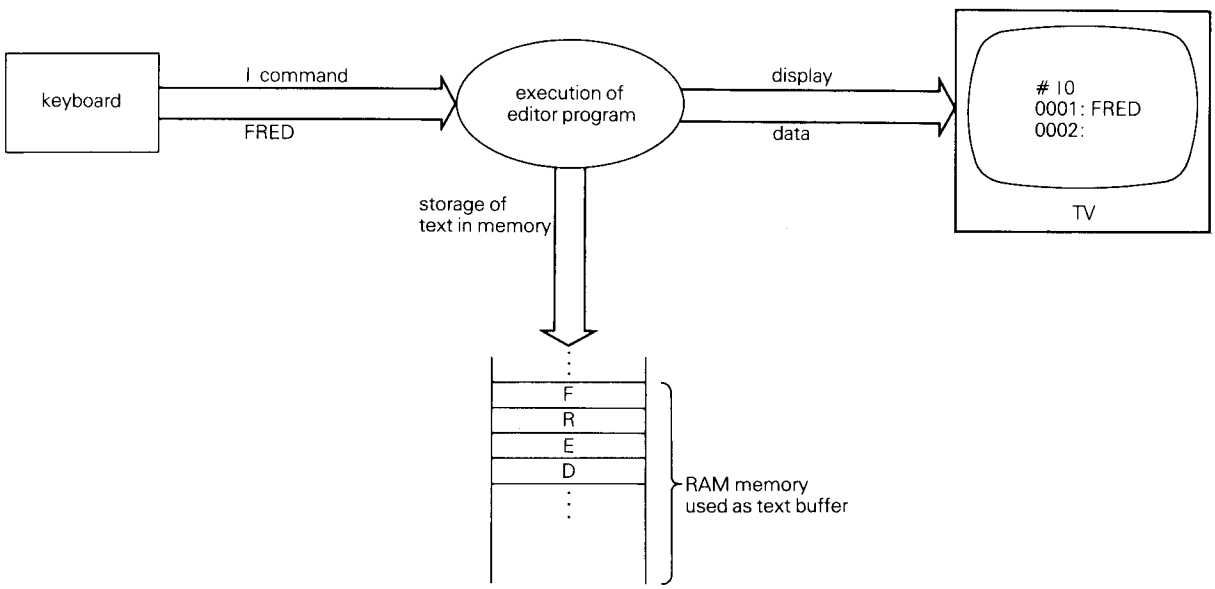


Figure 5.1 Example of editor command

from the one used by the monitor program. The editor also uses some of the monitor's *utility* routines (see Subsection 4.1), those which handle the TV, keyboard, and cassette interfaces, for example.

Figure 5.2 also shows the way in which the editor program makes use of the RAM memory. The *system RAM* is used as a 'workspace' by the editor, and as a *stack* area. But of main interest to the editor user is the *text buffer*, which is the area of RAM where the lines of text are stored and operated on by the editor, in response to editor commands.

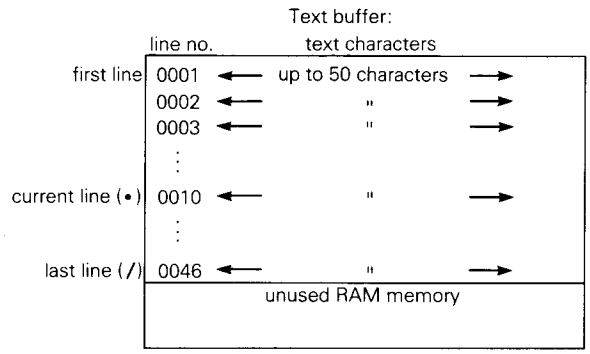


Figure 5.3 The text buffer

Figure 5.3 shows the structure of the text buffer in more detail. The text buffer will contain a variable number of *lines of text*, each of which consists of a line number followed by a variable number of *character* bytes. For example:

0010 THIS IS A LINE

is a line of text, whose line number is 10 (denary), and which contains fourteen characters of text, including three 'space' characters. When stored in RAM, this line of text will occupy seventeen consecutive locations. Two locations are used to contain the numeric code for the line number, followed by the fourteen locations containing character codes, and the last location contains the zero byte to indicate the end of the line. (The character codes used are the standard ASCII codes – Appendix C.)

The lines of text are themselves stored consecutively in the RAM area whose lowest address is 2800 (hexadecimal), and the line numbers of consecutive lines increase consecutively, as shown in Figure 5.3. Note that, although line numbers are shown as four-digit numbers, these are *denary numbers*, unlike the four-digit hexadecimal addresses of the monitor. The line numbers, therefore, run from 1 to a theoretical maximum of 9999.

The editor commands enable the character text in a line or sequence of lines to be displayed, modified, saved on cassette, etc., and the user specifies *which* lines by reference to the decimal line numbers. As an added convenience, the editor keeps track of two special lines, the *last* line and the *current* line, as shown in Figure 5.3. The user may refer to these lines by the special symbols '/' and '.' respectively, instead of by their actual line numbers. For example, the editor command P1,/ causes a display of all lines, from line number 1 to the last line in the text buffer. Because the number of lines in the text buffer is continuously changing as new lines are inserted or existing lines deleted (with a consequent renumbering of lines to keep the line numbers consecutive), the ability to use '.' and '/' is useful.

5.2 Editor behaviour

Figure 5.4 shows, schematically, the relationship between the components of the editor program (command acceptance, command execution, and interrupt handler), and the twelve types of editor command. The editor is entered from the monitor program (using the monitor's E command) and the *command acceptance* routine is activated. An important point to note is that the editor does not 'clear' the text buffer when entered from some other system program. Instead, it works out how much of the text buffer area of RAM contains data which fits the 'lines-of-text' format described in Section 5.1 above. This enables the user to use the monitor or assembler system programs and return to the editor with the text in the text buffer undisturbed (provided the user has not deliberately altered the contents of memory in the text buffer area by use of the monitor M command, for example).

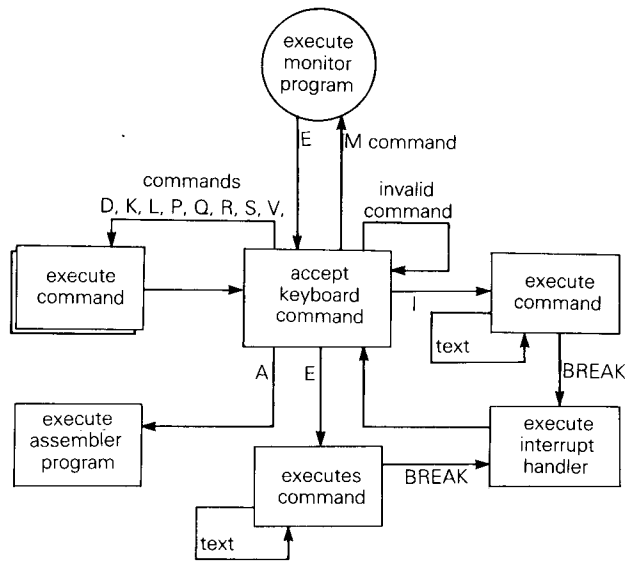


Figure 5.4 Editor behaviour

The command acceptance routine displays the editor *prompt* character '#' and awaits a command from the user. (The format of commands, and their detailed descriptions, are given later.) The twelve types of command split into two groups. The larger group, the D, K, L, P, Q, R, S, V commands, as shown in Figure 5.4, cause the appropriate *command execution* routine to be executed, after which the command acceptance routine is re-executed

automatically. These commands enable existing lines of text to be deleted, displayed, and saved on or loaded from cassette.

Of the remaining four commands, the M and A commands cause transfers to the *monitor* (Section 4) and the *assembler* (Section 6) system programs. The I and E commands are the only ones which enable new text to be typed into the text buffer. The I command enables complete new lines of text to be inserted between any two existing lines in the text buffer, whereas the E command allows modification to existing lines on a character-by-character basis. The command execution routines for these two commands accept text information as typed on the keyboard, and so the [BREAK] interrupt key is used to indicate when command execution should cease, as shown in Figure 5.4.

The command acceptance routine displays the editor prompt '#', but it also displays one of three additional messages in certain circumstances. 'ERROR' indicates that the command just typed was unrecognizable or invalid. 'NO TEXT' is simply informative; the text buffer is 'empty'. 'NEARLY FULL' is a warning message that the size of the text buffer is approaching that of the available RAM memory. It first appears when there is room for about 150 additional characters of text.

5.3 Editor command format

Each of the editor commands, as typed by the user on the keyboard in response to the editor prompt '#', has the same general form, namely:

Command character Argument list Terminator

The *command character* identifies the type of command and must be one of A, D, E, I, K, L, M, P, Q, R, S, V. The *argument list* is omitted for some commands, but if present, consists of up to three arguments, using a comma separator. The arguments supply additional information about the command; for example the command S2,10 has two arguments, each of which refers to a line number. This command causes the lines of text in the text buffer, whose line numbers range from 2 to 10 (decimal) inclusive, to be saved on cassette.

The *terminator* information, which indicates that the user has finished typing the command, can be supplied in any of four ways:

- keying [RETURN];
- keying [↓];
- keying [↑];
- keying [P], with the [CTRL] key held down.

For many commands, any of these terminators can be used, and [RETURN] is probably most convenient. For some commands, however, the different terminators have slightly different effects. For example, using [CTRL]-[P] with the P command causes lines of text to be 'printed' via HEKTOR's serial line interface rather than on the TV screen. Where differences between [RETURN], [↓], and [↑] exist, [RETURN] refers to the current line, [↓] to the *next* line, and [↑] to the *previous* line in the text buffer. (These differences are explained with respect to the individual commands, in Subsection 5.4.)

Returning to the *arguments* in the command, what meaning is attached to them? Usually, they refer to line numbers, specifying either a single line of text (one argument used) or a range of consecutive lines (two arguments used). For example, D3 causes deletion of line number 3 only, whereas D3,7 causes deletion of all lines in the range 3 to 7 inclusive.

The special symbols '.' and '/' can be used in arguments, where they refer to the *current* and *last* line respectively, as described in Subsection 5.1. The actual line numbers that these symbols refer to will change, as editor commands are executed. For example, if there are 12 lines currently in the text buffer and a new one is added, using the I command, to make 13 lines, the editor itself will change the value assigned to '/' from 12 to 13. Similarly, the current line '.' is updated after the execution of each command to be the line number of the line of text last processed. For example, P1,10 will cause '.' to be set to 10. (The detailed effects on the current line number are discussed below, for each command.) Note that, whenever lines are inserted or deleted, the lines are renumbered automatically by the editor to preserve the consecutive-numbering scheme. The values of '.' and '/' are updated accordingly.

As an additional convenience, a limited amount of *arithmetic* is allowed with line-number arguments. The current or last line can be specified, *offset* by a number of lines. For example, D.–1, .+1 has the effect of deleting three lines: the previous line, the current line, and the next line.

When line numbers are specified which are outside the range of lines currently in the text buffer, the editor automatically *truncates* them appropriately. For example, P1,9999 is equivalent to P1,/.

If a line-number argument is omitted where one would normally be expected, the editor takes the current line to be the implied argument. Thus D. is equivalent to D by itself. This facility is convenient, but should be used with caution.

For two commands, arguments other than line numbers are required. In these cases the argument begins with the '\' character, used as a separator instead of a comma. For the A command, which causes the *assembler* system program to be executed, there are a number of *options* available. These are discussed in Subsection 6.5.5, but an example of this command is A\L\S. For the Q command, which searches lines of text for the occurrence of a particular *string* of characters, the string is specified as an argument thus: Q1,10\FRED.

Finally, typing errors can be corrected at any stage of keying in a command up to the keying in of the terminator. The characters keyed in are 'echoed' on the screen, and use of the [←] key will delete the previous character.

5.4 Editor commands

Each of the twelve editor commands is described below, with the symbols <A1>, <A2>, etc., denoting *arguments*, and <T> denoting one of the *terminators* discussed in Subsection 5.3 above. The effect on the *current* line value, symbolically '.', is also included.

5.4.1 Assembler entry command

A<A1><A2>...<T> causes execution of the assembler program.

This command can have several arguments, each of which is an assembler option. Each argument is of the form '\' followed by a single character which specifies the option. (For details, see Subsection 6.5.) The assembler usually leaves the text in the text buffer intact, so re-editing of the existing text, following assembly, is possible.

For example, A\L\S\T\M\W is the command with all options specified.

5.4.2 Delete command

D<A><T> deletes the line specified by <A>. D<A1>,<A2><T> deletes the lines in the range <A1> to <A2>.

This command deletes whole lines from the text buffer, following which all remaining lines are renumbered to retain consecutive numbering. The current line number is updated so as to refer to the line of text following the block of lines deleted.

For example, if the text buffer contains:

```
0001  TOM
0002  DICK
0003  HARRY,
```

the command D1,2 will leave the text buffer containing 0001 HARRY, with the current line number having the value 1.

5.4.3 Edit line command

E<A><T> edits individual characters in the line specified by A.

Execution of this command enables individual characters to be changed, inserted or deleted in the specified line of text. The line is first displayed in its current state, with the *cursor* (the blinking underline) pointing to the first character in the line.

The *command acceptance* routine is then re-entered to accept a *subsidiary command*, which may be any of the following:

- [BREAK] indicates termination of the E command, and any changes made to the current line since it was last saved are ignored.
- [RETURN] (or [CTRL]-[P]) saves the line in its current state, and displays it again for any further editing.
- [↑] saves the line and displays the previous line ready for editing.
- [↓] saves the line and displays the next line, ready for editing.
- [→] moves the cursor to the next character on the right (unless it is currently pointing to the fiftieth character).
- [←] deletes the previous character to the left, and the cursor moves left to remain pointing to the current character (unless it is currently at the start of the line).
- Any other character keyed in is inserted at the cursor position, and the cursor moves right to remain pointing to the current character.

After accepting a subsidiary command and performing the requested action, a new subsidiary command is awaited (unless [BREAK] is keyed). The current line is set to be that of the last line displayed.

It is not easy to provide a compact example of line-editing, because of the close and dynamic relationship between the subsidiary commands and the display. It is recommended that users gain an appreciation of the power of this command by experimentation for themselves!

5.4.4 Insert command

I<A><T> enables insertion of lines of text *after* the line specified by A.

Command execution begins with a prompt message which is the line number of the lines of text to be inserted. This first line number will therefore be one more than that specified by <A>.

A *subsidiary command* is then awaited, which is expected to be of the form: <text characters><T>.

The typed text is 'echoed' on the screen, and typing corrections (using the [←] key) can be made. When a terminator <T> is keyed, the line of text is inserted into the text buffer, and the current line number becomes the line number of this line. Then a new prompt message (the line number of the next line) is displayed, and a new subsidiary command awaited.

If the user does not wish to insert any more lines of text, keying [BREAK] will terminate the I command execution immediately, without inserting the line corresponding to the last prompt.

Note that because the I command inserts lines *after* the line specified by <A>, there has to be special

provision for inserting lines at the start of the text buffer. For this command only, the form I0<T> is allowed, and this command enables line-insertion before line 1.

It should also be noted that line lengths may not exceed fifty text characters. Any additional text characters typed are simply ignored.

An example of insertion is shown in Table 5.1. After this command is executed, the text buffer will have had two lines of text inserted, and, after the automatic renumbering, these will be lines 21 and 22. The current line (symbolically '.') will be 22, and the value of '/' will have been increased by two.

5.4.5 'Kill' command

K<T> deletes all lines of text in the text buffer.

This command sets the current and last line indicators to zero. This means that following a K command, only the I0 and L commands will be meaningful.

The use of the K command is recommended when first using the editor during a session. It is conceivable that, by a freak, the random data in the text buffer area could appear to be valid lines of text. Remember that the editor never 'kills' the text buffer automatically.

5.4.6 Load-from-tape command

L<T> appends lines of text saved on cassette to the text buffer. Command execution for this command begins with the prompt message 'SET TO PLAY'. The user should place the rewound cassette, containing the lines of text to be loaded, in the cassette recorder and set its controls to PLAY.

Keying RETURN will cause the tape to start moving, and any lines of text found on the tape will be loaded into the text buffer *after* the current last line. If the tape does not appear (to HEKTOR) to contain valid text data, it will continue to search the tape indefinitely. (Use of the BREAK key will abort the search, if necessary.)

If the loading is successful, there is an automatic return to the command acceptance routine. If some loading has occurred, but a reading error is detected, 'ERROR' will be displayed. In these circumstances the text buffer may include some, but not all, of the lines stored on tape. The L command leaves the current line '.' unchanged, but the last line '/' is updated according to how many lines of text were successfully loaded.

Note that the lines loaded from tape will be renumbered automatically, to take account of the fact that they have been *appended* to the text buffer. Their line numbers at the time they were saved on tape are ignored.

Note also that attempting to use the *editor's* L command with cassettes which were recorded using the *monitor* program's S command (or vice versa) will cause unpredictable results!

5.4.7 Monitor entry command

M<T> returns to the monitor system program.

This command causes execution of the monitor program (Section 4). The response to the command is, therefore, the display of the monitor prompt character '>'.

Note that, on subsequent re-entry to the editor, the existing lines of text in the text buffer will be intact, unless monitor commands to alter the data stored in this area of memory have explicitly been used.

5.4.8 Print command

P<A><T> displays the specified line of text. P<A1>,<A2><T> displays the lines in the range <A1> to <A2>.

This command causes a display of the line(s) of text specified by the line-number arguments. For this command, the different terminators <T> have different effects.

If <T> is RETURN, there is simply a display of the

Table 5.1 Use of the I command

Keys used	*Display*	*Comment*
I20 RETURN	#I20 0021	Execute I command: line-number prompt
EDITING RETURN	0021 EDITING 0022	Subsidiary command: insert 'EDITING' line; move to next line
IS FUN RETURN	0022 IS FUN 0023	Subsidiary command: insert 'IS FUN' line; move to next line
BREAK	0023 # 	End of insertion: exit from I command

lines specified. If <T> is [CTRL]-[P], the textual information is sent to the serial line interface, instead of the TV interface.

If <T> is [↑], the line(s) displayed are those with line numbers which are one *less* than those specified in the arguments. If <T> is [↓], the lines displayed are those with line numbers which are one *more* than those specified. This enables the compact commands P[↓] and P[↑] to display the *next* line and the *previous* line respectively, with respect to the current line.

The P command affects the current line indicator, and the current line becomes the last one displayed before execution of the P command terminates.

5.4.9 Query command

Q<A1>, <A2>, <A3> <T> searches a range of lines, seeking a match with a specified character string.

For this command, <A3> consists of the '\' character followed by a *string* of text characters. <A1> and <A2> are line-number arguments. The sequence of lines of text from <A1> to <A2> is searched, seeking the first line whose text contains the specified string. (Note that the line number is not considered part of the text of a line.)

If a match is found, the relevant line is displayed and the current line becomes this line. If no match is found, the current line becomes that specified by <A2>, but execution terminates with no display.

For example, the command Q1,5\G IS could cause the display:

```
    0004 EDITING IS FUN
```

with this line becoming the current line.

5.4.10 Rewind command

R<T> connects power to the cassette recorder, to allow rewinding of cassettes.

The cassette recorder controls are usually disabled, because of HEKTOR's control of the recorder's power supply. Following the R command, power is made available to the recorder, and the user can operate the recorder controls to rewind cassettes ready for the saving or loading of lines of text.

Keying [RETURN] will remove the power from the recorder again, to complete execution of this command.

5.4.11 Save-on-tape command

S<A1>, <A2> <T> saves the specified lines of text on tape.

The immediate effect of this command is to cause the message 'SET RECORD' to be displayed on the screen. The user should ensure that a rewound cassette is placed in the recorder, and the recorder controls set to the record position. Then, keying [RETURN] will cause the lines of text specified by the range <A1> to <A2>, inclusive, to be sent to the recorder. When all the text has been sent, power is removed from the recorder, and command execution is complete.

It is good practice, immediately after saving text on tape, to use the R and V commands to rewind the cassette and verify that recording was successful. If this is done, any recording errors will be detected before the lines of text in the text buffer are lost. The lines can be re-recorded if necessary.

5.4.12 Verify tape command

V<T> verifies that the text recorded on tape is capable of being loaded into the text buffer.

Command execution for the verify command is very similar to that for the load-from-tape command. The only difference is that any text read from tape is not actually appended to the text buffer. The purpose of this command is to check that any subsequent L commands will have a high probability of success.

A successful verification is indicated by the display of the editor prompt, and by the *absence* of any error messages.

6 8085 ASSEMBLY LANGUAGE

6.1 Overview of assembly-language programming

The individual instructions in a microcomputer program must be stored as *machine-code* in the microcomputer's program memory: that is, as 8-bit binary codes which indicate the *operation* to be performed and any additional *operand* information required (such as the address of the location in memory which contains the data to be operated on). Not only are there different operations available in a microprocessor's *instruction set*, but there are also several different ways of specifying operands. (There are said to be several *addressing modes* for many of the basic operations.)

An example of machine-code is the 8-bit binary code 10010000 (or 90 hexadecimal). When interpreted as an instruction by the 8085 microprocessor, this means: 'subtract the number stored in the microprocessor's B register from that in its A register'.

There is little about the code 10010000 which indicates this meaning to a human programmer, and so programming directly in machine-code is a tedious and error-prone activity. However, programming in *assembly language* allows *symbols* to be used which are easier for the programmer to remember and interpret. The example instruction above would be written in the 8085 assembly-language as 'SUB B', which is clearly more nearly related to the description of the instruction than the binary or hexadecimal code of *machine language*. In this example, 'SUB' and 'B' are *symbolic* representations of the operation and operand parts of the instruction. Notice the space separating the two symbols. This means that the operation and operand parts of the instruction can be separately represented by symbols. In the machine-code 10010000, these two parts of the instruction are much less clear.

In this example, the symbol 'B' indicates that the contents of register B are to be used as an operand. Other addressing modes involve specifying addresses of locations in memory which contain the operand data. In this case, using assembly-language, symbols can be chosen by the programmer to represent particular addresses in memory. For example, 'LDA COUNT' is an instruction which copies (or *loads*) into the A register the data from a particular memory location, which is identified by the symbol 'COUNT'. In the machine-code version of this instruction, the actual address which locates the operand has to be specified as a 16-bit code.

Writing a program in the symbolic assembly-language form is preferable, but the symbols have eventually to be translated into their machine-code equivalent to be executed by the microprocessor. This translation can be performed automatically by an *assembler program*, and the system software in HEKTOR's ROM memory contains such an assembler. Assembly-language programs can be prepared using the HEKTOR *editor*, and then translated into machine-code by the assembler. Figure 6.1 illustrates schematically the process of program development using the HEKTOR system software as an aid to development.

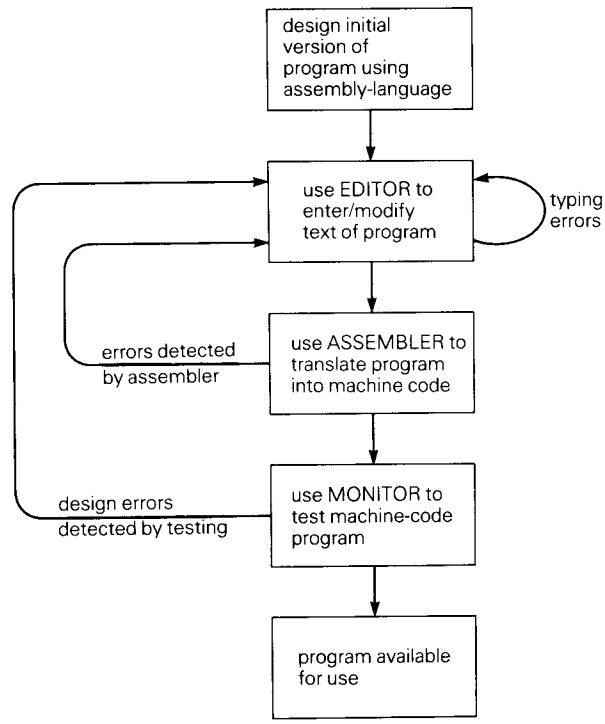


Figure 6.1 Program development

This section describes the structure of 8085 assembly-language programs, and how to use the assembler.

6.2 Assembly-language statements

A program in 8085 assembly-language consists of a number of lines of text (*source* lines). The assembler recognizes three types of source line:

- comment lines;
- assembly-language instruction lines;
- assembler directives.

6.2.1 Comment lines

Comment lines are ignored by the assembler. They are aids to program documentation that the programmer may wish to use. Comment lines are taken to be any lines which begin with a semicolon. For example:

```
;
; INPUT ROUTINE:
;
```

is a set of three comment lines.

6.2.2 Assembly-language instructions

Each *instruction line* is translated by the assembler into a machine-code instruction. These lines consist of up to four *fields*, not all of which are present in every instruction line. The line format, in terms of these fields, is:

[label field] [opcode field] [operand field] [comment field]

The *label field* is always optional. It consists of a label symbol followed by a colon. The label symbol consists of up to six characters. The label symbol is made up of letters or a mixture of letters and numerals. However, the first character must be a letter. The label symbol is a symbol which represents the address in memory where the machine-code instruction (as translated from the opcode/operand fields in the instruction line) will eventually be stored. Accordingly, each label in the program must use a unique symbol. These are examples of valid label symbols:

START
LINE27
M8085A
Z

The *opcode field* is always present, and consists of a single *mnemonic* symbol (mnemonic means 'serving to remind'). The symbol denotes one of the eighty basic operations in the 8085's *instruction set* (see Subsection 6.4).

An *operand field* is not required for some instructions (the 'halt' operation, for example). Other instructions involve either one or two operands. Each of the operands contains information identifying the location within the microcomputer of the data which is to be operated on, or the data itself. For example, in the instruction 'MOV A,B', the operands referred to are the contents of the microprocessor's A and B registers. The instruction means 'move the data from the B register into the A register'. Whenever there are two operands in the operand field, they are separated by a comma. (For details of operand specification, see Subsection 6.3.)

The *comment field* in an instruction line is always optional and, like the comment line, starts with a semicolon. Any text following the semicolon is ignored by the assembler, but is to help you understand the operation of the program at that point.

For the assembler to be able to detect where one field in an instruction line ends and the next begins, certain conventions regarding field *delimiters* exist. The colon in a label field is a sufficient delimiter between the label and opcode fields. But for readability, one or more spaces may be inserted after the colon. The assembler expects one or more spaces between the opcode and operand fields, if there is an operand field. The end of the operand field (or the opcode field if there is no operand field) is indicated by a space, by the semicolon which starts a comment field, or by the end of the line, whichever occurs first.

6.2.3 Assembler directives

The third type of source line is used to supply additional information about the program to the assembler. These lines do not translate into machine instructions, although they have a similar layout to the instruction lines; that is, there are up to four fields:

[label field] [pseudo-opcode field] [operand field] [comment field]

The label, operand, and comment fields have the same format as those in instruction lines, but the pseudo-operations, of which there are five types, have different functions.

Two pseudo-operations allow the programmer to define blocks of data that the program is to operate upon. They are:

- the DB pseudo-operation;
- the DW pseudo-operation.

The *data byte* pseudo-operation, DB, causes the assembler to store in memory the 8-bit data specified in the operand field. The operand may define a single byte (see Subsection 6.3 for the allowed forms of 8-bit operand), or a string of character data. In this latter case, the operand is specified as a sequence of characters enclosed in single quotes, and the assembler translates this into *character codes* (see Appendix C) for storage in memory. Examples of the DB directive are:

```
CODES:  DB 5          a list consisting of the
        DB 4          codes 05, 04, 03 (hex)
        DB 3
MESS3:  DB 'ERROR'    a list consisting of the
                      codes 45, 52, 52, 4F, 52
                      (hex)
```

Lists of data can be accessed by reference to their symbolic starting address. For example, the symbolic instruction 'LDA CODES+2' will be translated into a machine instruction which will cause the third item in the list of codes, which is the code 03 (hex) to be copied into the A register.

The *data word* pseudo-operation, DW, causes the assembler to store in memory the 16-bit data specified in the operand field. (See Subsection 6.3 for the allowed forms of 16-bit operand.) The data occupies two adjacent 8-bit locations in the memory, with the least significant 8 bits of the operand stored in the location with the lower address. For example:

```
POWERS: DW 1        a list consisting of the 8-
        DW 10       bit codes 01, 00, 0A, 00,
        DW 100      64, 00, E8, 03 (hex)
        DW 1000
```

There are three other pseudo-operations used in assembler directive lines. The *equate* pseudo-operation, EQU, is a direct way of assigning a value

to a symbol (rather than the indirect way of using a label symbol in other source lines). The directive 'COUNT: EQU 47' allows the symbol COUNT to be used elsewhere in the program whenever the constant data value 47 is required. The *origin* pseudo-operation, ORG, directs the assembler as to where in memory to store the machine-code it creates by translation of the subsequent lines of the assembly-language program. The binary codes are stored in consecutive locations beginning at the location specified by the address given in the operand field of the ORG directive. For example:

```
        ORG 12288
TABLE:  DB 1
        DB 2
        DB 3
```

has the effect of storing the specified table of data starting at the location with the address 12288 denary (that is, 3000 hexadecimal). The symbolic address 'TABLE' can, of course, be used in the assembly-language program when referring to the table of data. HEKTOR's assembler allows at most one origin directive in a program. It is used if the programmer wishes to specify exactly where in memory the machine-code version of the program should be stored for execution by the microprocessor. If there is no origin directive, the code will be assembled so as to be available for execution in an area of memory which is not used by the editor's text buffer (see Sections 5 and 6.5).

Finally, the *end-of-program* pseudo-operation, END, indicates to the assembler that this is the last line in the program. An optional operand field for this pseudo-operation allows the program's *entry point* to be defined. (The entry point is the first instruction in the program to be executed.) The example of Figure 6.2 shows a program structure, using each of the assembler directives.

```
;
;FIRST THE DEFINITIONS FOR DATA
;
        ORG 15000       ;ORIGIN AT ADDRESS 15000
MESG:   DB  'EXAMPLE'   ;STORED CHARACTER CODES
FACTOR: EQU 8000        ;DEFINE VALUE FOR FACTOR
CONST:  DW  FACTOR      ;16-BIT VALUE STORED
;
;NEXT THE INSTRUCTIONS
;
START:  LDA MESG        ;GET CODE FOR 'E'
          .
          .
          .
        END START       ;DEFINE ENTRY POINT
```

Figure 6.2 Use of assembler directives

6.3 Operand specification

The operand field of an instruction contains information about one or more of the *operands* involved in the instruction. If there is more than one operand specified, the specifications are separated by a comma. The comma acts as a delimiter.

For some instructions, the *operand value* itself is required to be specified (as an 8-bit or 16-bit value). For others, the *address* of the location holding the operand is specified (either a particular *register* in the microprocessor, or a particular address elsewhere in the microcomputer). Each type of operand specification is discussed below.

6.3.1 Specifying 8-bit operand values

The assembler accepts *symbolic, numeric* and some *mixed* specifications for 8-bit operands, namely:

- <symbol>
- <number>
- <symbol + number>
- <symbol − number>

For example, suppose the symbol COUNT has previously been defined (in an 'equate' directive) as having the value 47. Then the instruction 'MVI A,COUNT−1' has the meaning: 'copy the value of the second operand (COUNT−1) into the A register'. This value will be evaluated by the assembler as 47−1 =46, and the 8-bit code for 46 will form part of the machine-code instruction. Note that the 'I' in the opcode mnemonic MVI indicates an *immediate* operation; that is, the operand data itself is immediately available in the instruction, rather than the address of the data.

The value of operands can be specified in any of three ways:

- denary numbers;
- hexadecimal numbers;
- ASCII codes.

A string of *numeric* characters, by itself, is taken to be a denary number. A string of characters from the set 0–9, A–F, immediately followed by an H, and beginning with a numeric character, is interpreted as a hexadecimal number. For example, 0AFH will be evaluated as the hexadecimal number AF (which is 175 denary) but AFH will be treated as a symbol, not a number.

A character enclosed in single quotes will be translated by the assembler into the 8-bit ASCII code for that character (see Appendix C). Therefore, using the previous example in which COUNT has the value 47 denary, each of the following instructions has the same machine-code equivalent:

```
MVI A,COUNT−1     ;symbolic expression
MVI A,46          ;denary number
MVI A,2EH         ;hex number
MVI A,'.'         ;ASCII code
```

6.3.2 Specifying 16-bit operand values

Some instructions have an operand which is expressed as a 16-bit value in the machine-code instruction. The same forms of expression as for 8-bit operands can be used in the assembly-language version, with the assembler evaluating the operand in 16 bits rather than in 8 bits. In the machine-code instruction, the 16-bit operand is split into two 8-bit codes for processing and storage in the 8-bit microcomputer. As an example, the instruction LXI H,3 means 'load the value of the second operand (3 in this case) expressed as a 16-bit value, into the register *pair* consisting of the 8-bit H register and the 8-bit L register in the microprocessor'. Note that after executing this instruction the L register would contain 03 (hex) and the H register 00 (hex), as the 16-bit code for 3 is 0003 (hex). The machine-code for this instruction is three 8-bit codes (or bytes):

21 (hex) – opcode for 'LXI'
03 (hex) – least significant 8 bts of operand
00 (hex) – most significant 8 bits of operand

By convention, when a 16-bit value is stored in memory as two consecutive bytes, the least significant byte is stored in the location with the lower address, as shown in this example.

6.3.3 Specifying operand addresses

Some instructions require that the *address* of the location holding the data to be operated on is specified, rather than the value of the data itself. For some of these instructions, a 16-bit *memory address* is specified; for others, the memory address is *implied*. One instruction allows only a limited number of memory addresses to be specified, the *restart addresses*. Finally, two instructions have operands which refer to 8-bit *I/O addresses*, rather than memory addresses.

Table 6.1 summarizes these four types of addresses used as operands, and the ways in which they can be expressed in the 8085 assembly-language are described below.

6.3.3.1 16-bit memory address

Memory addresses used as operands (type A16 in Table 6.1) can be specified by the programmer using the symbolic, numeric, or mixed expressions for operands that are allowed for 16-bit data values (Subsections 6.3.1 and 6.3.2). For example, 'JMP START+3' means 'jump (or transfer control) to the section of program whose first instruction is found at the symbolic address START+3'. In this example, the assembler will evaluate START+3 as a particular 16-bit memory address, and this address will be stored as part of the machine-code instruction. 'START', of course, will have had to be specified, probably as a *label* attached to an instruction.

6.3.3.2 Implied memory address

There is a large group of instructions for which the memory address is not specified directly within the instruction, but is *implied*. The actual address is contained within the microprocessor, in its H and L registers.

The contents of these two 8-bit registers are strung together to give a 16-bit address. The H, or 'high', register contains the most significant 8 bits of the 16-bit address, while the L, or 'low', register contains the least significant part of the address. For example, if H contains 0A (hex) and L contains B1 (hex), the 16-bit address referred to by the H, L register pair is 0AB1 (hex). The symbol 'M' (see Table 6.1) is used in the operand field of an instruction which allows this addressing mode. Thus, INR M means 'increment (or add 1 to) the 8-bit number stored in memory at an address given by the contents of the H, L register pair'. Note that it is the programmer's responsibility to ensure, using other instructions, that the H and L registers are loaded with the required address before the instruction is executed.

6.3.3.3 Restart addresses

The 'restart' instruction (see Subsection 6.4) refers to only eight memory addresses where the restart can occur. These are specified, in a coded form, by a single digit in the range 0 to 7. (This is the A3 type

Table 6.1 Types of operand address

Type of operand address	*Description of address*	*Examples of symbolic specification*
A16	16-bit memory address	START+3 8765 3000H
A8	8-bit I/O address	DEVI+2 63 40H
A3	One of eight restart addresses	0 7
M	Implied memory address (the 16-bit address stored in the H,L register pair)	M

of operand in Table 6.1.) The actual addresses in memory are those given by multiplying the operand digit by 8 (namely the hexadecimal addresses 0000, 0008, 0010, 0018, 0020, 0028, 0030, 0038).

6.3.3.4 I/O addresses

The 'input' and 'output' instructions (Subsection 6.4) have operands which are *8-bit I/O addresses* (type A8 in Table 6.1). These addresses refer to the *I/O devices* in the microcomputer rather than memory. The programmer can specify these operand addresses using the same symbolic forms as for 8-bit data value operands (Subsection 6.3.1).

6.3.4 Specifying registers

The fourth type of operand address specifies addresses within the microprocessor; that is, it identifies *microprocessor registers*. Figure 6.3 shows these registers and their symbolic names. Two of the registers, the *stack pointer* (SP) and the *program counter* (PC) are 16-bit registers and their contents, being memory addresses, can only be operated on as 16-bit quantities. The remaining nine registers (A, F, B, C, D, E, H, L, I) are *8-bit registers*, but some instructions operate on the 16-bit values contained in particular *register pairs*. The possible register pairs are A and F, B and C, D and E, H and L.

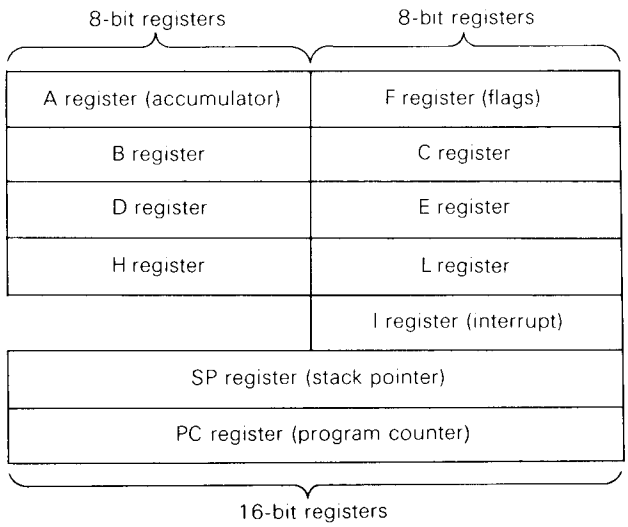


Figure 6.3 8085 microprocessor registers

Some instructions are limited in that they operate on data in only one particular register – there is therefore no need to specify the register as an operand.

For example, the 'CMA' instruction operates only on the A register contents, and the operand field is not required for this instruction.

For other instructions, some registers or register pairs are allowed, but not others. It is, therefore, necessary to classify the registers and register pairs into groups, so that the programmer can identify which registers or register pairs can be used as operands for particular instructions. These groups are shown in Table 6.2. In assembly-langauge instructions, the symbolic register name is used in the operand field.

Note that the symbol 'H', for example, is used to refer to either the 8-bit register H or the 16-bit register pair H, L. The assembler decides which is intended from the operation code – some instructions have 8-bit operands, others 16-bit operands, but none allow a choice. For example:

```
LXI  H,3  ;load H, L with 16-bit code for 3
MVI  H,3  ;load H with 8-bit code for 3
```

Table 6.2 Groups of register operands

Register group name	*Type of register(s) involved*	*Registers allowed in group*	*Symbol used in operand field*
R8	single 8-bit register	A register	A
		B register	B
		C register	C
		D register	D
		E register	E
		H register	H
		L register	L
R16a	16-bit register, or pair of 8-bit registers	B & C registers	B
		D & E registers	D
		H & L registers	H
		Stack pointer	SP
R16b	pair of 8-bit registers	A & F registers	PSW
		B & C registers	B
		D & E registers	D
		H & L registers	H
R16c	pair of 8-bit registers	B & C registers	B
		D & E registers	D

6.4 Opcode specification

The opcode field of an instruction contains a single mnemonic symbol, representing one of the types of operation in the 8085's instruction set. The assembler translates this symbol into an 8-bit operation code, using its *permanent symbol table*. If the instruction has one or more *register operands* (Table 6.2), then this information is also coded into the 8 bits of this first word of the machine-code instruction. The other types of operand, 8-bit values, 16-bit values, memory addresses, etc. (see Subsection 6.3), are coded into one or two additional 8-bit codes and are appended to the first word. Operand types are summarized in Table 6.3.

Table 6.3 Operand types

Operand Type	*Description*
D16	16-bit operand value
D8	8-bit operand value
A16	16-bit memory address
M	implied address (specified by H,L contents)
A8	8-bit I/O address
A3	One of eight restart addresses
R16a	16-bit register or register-pair (Table 6.2)
R16b	16-bit register or register-pair (Table 6.2)
R16c	16-bit register or register-pair (Table 6.2)
R8	8-bit register (Table 6.2)

In this subsection, the instruction set is presented in several ways. Table 6.4 gives a condensed interpretation of each instruction symbol, together with the types of operand that are used with it. Table 6.5 lists the machine-code opcodes of all instructions, in numeric order, with their assembly-language equivalents. Table 6.6 gives the same information, but in alphabetic order of opcode symbols.

Table 6.6 also gives information on the time taken to execute each of these instructions. This number is the number of microprocessor clock cycles required for the instruction to be fetched into the microprocessor and executed. On HEKTOR, the frequency is 3 024 000 Hz, giving a cycle time of approximately 0.328 microseconds. The ACI instruction therefore takes 2.297 microseconds to execute. The conditional instructions' execution time depends on whether the condition is met or not; the time is shorter if it is not met. The execution time for a sequence of instructions can be calculated simply by adding together the individual execution times.

In the remainder of this subsection, the instructions are described in detail, and presented in alphabetic order. Before explaining these instructions, however, it is necessary to review the function of three special microprocessor registers, namely:

- program counter (PC);
- stack pointer (SP);
- flag register (F).

6.4.1 Program counter (PC)

The purpose of this 16-bit register is to store the address of the memory location which holds the next machine-code instruction to be fetched and executed. The microprocessor supplies this address information, via the microcomputer address bus, at the start of the sequence of operations which make up the *fetch/execute sequence* (or *cycle*) for an instruction. The addressed memory location supplies the 8-bit data stored at that address via the data bus, and this data is interpreted by the microprocessor as a machine-code instruction. (If the instruction requires additional operand information, the program counter's contents are incremented, allowing the microprocessor to read the second, and possibly third, 8-bit code of the instruction.)

During the fetching of all instructions, the microprocessor automatically increments the contents of the program counter each time it accepts a machine-code byte. In this way, instructions are normally executed in the order that they are stored in memory.

Some instructions, however, achieve their effects by deliberately altering the contents of the program counter. The instruction 'JMP START' instructs the microprocessor to execute the instruction stored at address START. This is achieved by simply loading the operand information (the address START) into the program counter, and the next instruction executed will automatically be that stored at START.

6.4.2 Stack pointer (SP)

For the 8085 microprocessor, as with most computers, an area of read/write memory is used as a *stack*. The stack is used for the temporary storage of data, but information is stored in and retrieved from the stack in a unique, and useful, way. Its method of access is by *pushing* data onto the top of the stack or *popping* it off again, as discussed in Section 2.5. The top of the stack is the location in memory whose address is contained in the 16-bit *stack pointer* in the microprocessor.

In the 8085 microprocessor, push and pop operations involve only 16-bit data, so each 'push' operation is really two pushes, one for each byte of the pair that make up a 16-bit data item. The source of the 16-bit data for pushing, and the destination of popped 16-bit data, are the 16-bit registers (or pairs of 8-bit registers) in the microprocessor.

There are instructions which explicitly enable the pushing and popping of register pair data, but other instructions include these operations as part of their overall function. The most important of these are the *subroutine call* and *return* instructions.

Table 6.4 Summary of 8085 instruction types

Data Copy Group

MOV	Copy data between register/memory and register/memory
MVI	Copy operand data to register/memory
LDA, LDAX	Copy data from memory to A register
STA, STAX	Copy data from A register to memory
LHLD	Copy data from memory to HL register pair
SHLD	Copy data from HL register pair to memory
LXI	Copy operand data to register pair
XCHG	Exchange data between HL and DE register pairs
XTHL	Exchange data between HL register pair and top of stack

Arithmetic Group

ADD	Add register/memory contents to A register contents
ADI	Add operand data to A register contents
ADC	Add register/memory and Carry contents to A register contents
ACI	Add operand data and Carry contents to A register contents
SUB	Subtract register/memory contents from A register contents
SUI	Subtract operand data from A register contents
SBB	Subtract register/memory and Carry contents from A register contents
SBI	Subtract operand data and Carry contents from A register contents
INR	Increment register/memory contents by 1
DCR	Decrement register/memory contents by 1
INX	Increment register pair contents by 1
DCX	Decrement register pair contents by 1
DAD	Add register pair contents to HL register pair contents
DAA	Adjust A register contents for BCD result following addition

Logical Group

ANA	Logical AND register/memory contents with A register contents
ANI	Logical AND operand data with A register contents
ORA	Logical OR register/memory contents with A register contents
ORI	Logical OR operand data with A register contents
XRA	Logical EXCLUSIVE-OR register/memory contents with A register contents
XRI	Logical EXCLUSIVE-OR operand data with A register contents
CMP	Compare register/memory contents with A register contents
CPI	Compare operand data with A register contents
RLC	Rotate A register contents left, and into Carry
RRC	Rotate A register contents right, and into Carry
RAL	Rotate A register and Carry contents left
RAR	Rotate A register and Carry contents right
CMA	Complement A register contents
CMC	Complement Carry contents
STC	Set Carry contents to 1

Program Sequence Control Group

Jump if:	Call if:	Return if:	Condition is:
JC	CC	RC	Carry (Carry=1)
JNC	CNC	RNC	No Carry (Carry=0)
JZ	CZ	RZ	Zero (Zero=1)
JNZ	CNZ	RNZ	Not Zero (Zero=0)
JP	CP	RP	Plus (Sign=0)
JM	CM	RM	Minus (Sign=1)
JPE	CPE	RPE	Parity even (Parity=1)
JPO	CPO	RPO	Parity odd (Parity=0)
JMP	CALL	RET	Unconditionally

PCHL	Copy data from HL register pair to Program Counter
RST	Call routine at restart address

Stack Operation Group

PUSH	Push register pair contents onto the stack
POP	Pop top-of-stack data into register pair
SPHL	Copy data from HL register pair to Stack Pointer

Input/Output Group

IN	Copy data from I/O device to A register
OUT	Copy data from A register to I/O device

Machine Control Group

EI	Enable interrupt system
DI	Disable interrupt system
RIM	Copy Interrupt Status data to A register
SIM	Copy A register contents to Interrupt Control
HLT	Halt processor
NOP	No operation

Table 6.5 Opcodes in numeric order

OP CODE	MNEMONIC	OP CODE	MNEMONIC	OP CODE	MNEMONIC	OP CODE	MNEMONIC
00	NOP	40	MOV B,B	80	ADD B	C0	RNZ
01	LXI B,D16	41	MOV B,C	81	ADD C	C1	POP B
02	STAX B	42	MOV B,D	82	ADD D	C2	JNZ A16
03	INX B	43	MOV B,E	83	ADD E	C3	JMP A16
04	INR B	44	MOV B,H	84	ADD H	C4	CNZ A16
05	DCR B	45	MOV B,L	85	ADD L	C5	PUSH B
06	MVI B,D8	46	MOV B,M	86	ADD M	C6	ADI D8
07	RLC	47	MOV B,A	87	ADD A	C7	RST 0
08	-	48	MOV C,B	88	ADC B	C8	RZ
09	DAD B	49	MOV C,C	89	ADC C	C9	RET
0A	LDAX B	4A	MOV C,D	8A	ADC D	CA	JZ A16
0B	DCX B	4B	MOV C,E	8B	ADC E	CB	-
0C	INR C	4C	MOV C,H	8C	ADC H	CC	CZ A16
0D	DCR C	4D	MOV C,L	8D	ADC L	CD	CALL A16
0E	MVI C,D8	4E	MOV C,M	8E	ADC M	CE	ACI D8
0F	RRC	4F	MOV C,A	8F	ADC A	CF	RST 1
10	-	50	MOV D,B	90	SUB B	D0	RNC
11	LXI D,D16	51	MOV D,C	91	SUB C	D1	POP D
12	STAX D	52	MOV D,D	92	SUB D	D2	JNC A16
13	INX D	53	MOV D,E	93	SUB E	D3	OUT A8
14	INR D	54	MOV D,H	94	SUB H	D4	CNC A16
15	DCR D	55	MOV D,L	95	SUB L	D5	PUSH D
16	MVI D,D8	56	MOV D,M	96	SUB M	D6	SUI D8
17	RAL	57	MOV D,A	97	SUM A	D7	RST 2
18	-	58	MOV E,B	98	SBB B	D8	RC
19	DAD D	59	MOV E,C	99	SBB C	D9	-
1A	LDAX D	5A	MOV E,D	9A	SBB D	DA	JC A16
1B	DCX D	5B	MOV E,E	9B	SBB E	DB	IN A8
1C	INR E	5C	MOV E,H	9C	SBB H	DC	CC A16
1D	DCR E	5D	MOV E,L	9D	SBB L	DD	-
1E	MVI E,D8	5E	MOV E,M	9E	SBB M	DE	SBI D8
1F	RAR	5F	MOV E,A	9F	SBB A	DF	RST 3
20	RIM	60	MOV H,B	A0	ANA B	E0	RPO
21	LXI H,D16	61	MOV H,C	A1	ANA C	E1	POP H
22	SHLD A16	62	MOV H,D	A2	ANA D	E2	JPO A16
23	INX H	63	MOV H,E	A3	ANA E	E3	XTHL
24	INR H	64	MOV H,H	A4	ANA H	E4	CPO A16
25	DCR H	65	MOV H,L	A5	ANA L	E5	PUSH H
26	MVI H,D8	66	MOV H,M	A6	ANA M	E6	ANI D8
27	DAA	67	MOV H,A	A7	ANA A	E7	RST 4
28	-	68	MOV L,B	A8	XRA B	E8	RPE
29	DAD H	69	MOV L,C	A9	XRA C	E9	PCHL
2A	LHLD A16	6A	MOV L,D	AA	XRA D	EA	JPE A16
2B	DCX H	6B	MOV L,E	AB	XRA E	EB	XCHG
2C	INR L	6C	MOV L,H	AC	XRA H	EC	CPE A16
2D	DCR L	6D	MOV L,L	AD	XRA L	ED	-
2E	MVI L,D8	6E	MOV L,M	AE	XRA M	EE	XRI D8
2F	CMA	6F	MOV L,A	AF	XRA A	EF	RST 5
30	SIM	70	MOV M,B	B0	ORA B	F0	RP
31	LXI SP,D16	71	MOV M,C	B1	ORA C	F1	POP PSW
32	STA A16	72	MOV M,D	B2	ORA D	F2	JP A16
33	INX SP	73	MOV M,E	B3	ORA E	F3	DI
34	INR M	74	MOV M,H	B4	ORA H	F4	CP A16
35	DCR M	75	MOV M,L	B5	ORA L	F5	PUSH PSW
36	MVI M,D8	76	HLT	B6	ORA M	F6	ORI D8
37	STC	77	MOV M,A	B7	ORA A	F7	RST 6
38	-	78	MOV A,B	B8	CMP B	F8	RM
39	DAD SP	79	MOV A,C	B9	CMP C	F9	SPHL
3A	LDA A16	7A	MOV A,D	BA	CMP D	FA	JM A16
3B	DCX SP	7B	MOV A,E	BB	CMP E	FB	EI
3C	INR A	7C	MOV A,H	BC	CMP H	FC	CM A16
3D	DCR A	7D	MOV A,L	BD	CMP L	FD	-
3E	MVI A,D8	7E	MOV A,M	BE	CMP M	FE	CPI D8
3F	CMC	7F	MOV A,A	BF	CMP A	FF	RST 7

Note: D8 = 8-bit data
A8 = 8-bit I/O address
D16 = 16-bit data
A16 = 16-bit memory address

Table 6.6 Opcodes in alphabetical order

MNEMONIC	OP CODE	TIME
ACI D8	CE	7
ADC A	8F	4
ADC B	88	4
ADC C	89	4
ADC D	8A	4
ADC E	8B	4
ADC H	8C	4
ADC L	8D	4
ADC M	8E	7
ADD A	87	4
ADD B	80	4
ADD C	81	4
ADD D	82	4
ADD E	83	4
ADD H	84	4
ADD L	85	4
ADD M	86	7
ADI D8	C6	7
ANA A	A7	4
ANA B	A0	4
ANA C	A1	4
ANA D	A2	4
ANA E	A3	4
ANA H	A4	4
ANA L	A5	4
ANA M	A6	7
ANI D8	E6	7
CALL A16	CD	18
CC A16	DC	9/18
CM A16	FC	9/18
CMA	2F	4
CMC	3F	4
CMP A	BF	4
CMP B	B8	4
CMP C	B9	4
CMP D	BA	4
CMP E	BB	4
CMP H	BC	4
CMP L	BD	4
CMP M	BE	7
CNC A16	D4	9/18
CNZ A16	C4	9/18
CP A16	F4	9/18
CPE A16	EC	9/18
CPI D8	FE	7
CPO A16	E4	9/18
CZ A16	CC	9/18
DAA	27	4
DAD B	09	10
DAD D	19	10
DAD H	29	10
DAD SP	39	10
DCR A	3D	4
DCR B	05	4
DCR C	0D	4
DCR D	15	4
DCR E	1D	4
DCR H	25	4
DCR L	2D	4
DCR M	35	10
DCX B	0B	6
DCX D	1B	6
DCX H	2B	6
DCX SP	3B	6
DI	F3	4
EI	FB	4
HLT	76	5
IN A8	DB	10
INR A	3C	4
INR B	04	4
INR C	0C	4
INR D	14	4
INR E	1C	4
INR H	24	4
INR L	2C	4
INR M	34	10
INX B	03	6
INX D	13	6
INX H	23	6
INX SP	33	6
JC A16	DA	7/10
JM A16	FA	7/10
JMP A16	C3	10
JNC A16	D2	7/10
JNZ A16	C2	7/10
JP A16	F2	7/10
JPE A16	EA	7/10
JPO A16	E2	7/10
JZ A16	CA	7/10
LDA A16	3A	13
LDAX B	0A	7
LDAX D	1A	7
LHLD A16	2A	16
LXI B,D16	01	10
LXI D,D16	11	10
LXI H,D16	21	10
LXI SP,D16	31	10
MOV A,A	7F	4
MOV A,B	78	4
MOV A,C	79	4
MOV A,D	7A	4
MOV A,E	7B	4
MOV A,H	7C	4
MOV A,L	7D	4
MOV A,M	7E	7
MOV B,A	47	4
MOV B,B	40	4
MOV B,C	41	4
MOV B,D	42	4
MOV B,E	43	4
MOV B,H	44	4
MOV B,L	45	4
MOV B,M	46	7
MOV C,A	4F	4
MOV C,B	48	4
MOV C,C	49	4
MOV C,D	4A	4
MOV C,E	4B	4
MOV C,H	4C	4
MOV C,L	4D	4
MOV C,M	4E	7
MOV D,A	57	4
MOV D,B	50	4
MOV D,C	51	4
MOV D,D	52	4
MOV D,E	53	4
MOV D,H	54	4
MOV D,L	55	4
MOV D,M	56	7
MOV E,A	5F	4
MOV E,B	58	4
MOV E,C	59	4
MOV E,D	5A	4
MOV E,E	5B	4
MOV E,H	5C	4
MOV E,L	5D	4
MOV E,M	5E	7
MOV H,A	67	4
MOV H,B	60	4
MOV H,C	61	4
MOV H,D	62	4
MOV H,E	63	4
MOV H,H	64	4
MOV H,L	65	4
MOV H,M	66	7
MOV L,A	6F	4
MOV L,B	68	4
MOV L,C	69	4
MOV L,D	6A	4
MOV L,E	6B	4
MOV L,H	6C	4
MOV L,L	6D	4
MOV L,M	6E	7
MOV M,A	77	7
MOV M,B	70	7
MOV M,C	71	7
MOV M,D	72	7
MOV M,E	73	7
MOV M,H	74	7
MOV M,L	75	7
MVI A,D8	3E	7
MVI B,D8	06	7
MVI C,D8	0E	7
MVI D,D8	16	7
MVI E,D8	1E	7
MVI H,D8	26	7
MVI L,D8	2E	7
MVI M,D8	36	10
NOP	00	4
ORA A	B7	4
ORA B	B0	4
ORA C	B1	4
ORA D	B2	4
ORA E	B3	4
ORA H	B4	4
ORA L	B5	4
ORA M	B6	7
ORI D8	F6	7
OUT A8	D3	10
PCHL	E9	6
POP B	C1	10
POP D	D1	10
POP H	E1	10
POP PSW	F1	10
PUSH B	C5	12
PUSH D	D5	12
PUSH H	E5	12
PUSH PSW	F5	12
RAL	17	4
RAR	1F	4
RC	D8	6/12
RET	C9	10
RIM	20	4
RLC	07	4
RM	F8	6/12
RNC	D0	6/12
RNZ	C0	6/12
RP	F0	6/12
RPE	E8	6/12
RPO	E0	6/12
RRC	0F	4
RST 0	C7	12
RST 1	CF	12
RST 2	D7	12
RST 3	DF	12
RST 4	E7	12
RST 5	EF	12
RST 6	F7	12
RST 7	FF	12
RZ	C8	6/12
SBB A	9F	4
SBB B	98	4
SBB C	99	4
SBB D	9A	4
SBB E	9B	4
SBB H	9C	4
SBB L	9D	4
SBB M	9E	7
SBI D8	DE	7
SHLD A16	22	16
SIM	30	4
SPHL	F9	6
STA A16	32	13
STAX B	02	7
STAX D	12	7
STC	37	4
SUB A	97	4
SUB B	90	4
SUB C	91	4
SUB D	92	4
SUB E	93	4
SUB H	94	4
SUB L	95	4
SUB M	96	7
SUI D8	D6	7
XCHG	EB	4
XRA A	AF	4
XRA B	A8	4
XRA C	A9	4
XRA D	AA	4
XRA E	AB	4
XRA H	AC	4
XRA L	AD	4
XRA M	AE	7
XRI D8	EE	7
XTHL	E3	16

Note: D8 = 8-bit data
A8 = 8-bit I/O address
D16 = 16-bit data
A16 = 16-bit memory address

TIME = execution time in machine cycles; where two times given, longer time is if action is performed

6.4.3 Flag register (F)

The flag register can be treated as an 8-bit register, but is more usefully considered as a collection of 1-bit registers or *flags*. Of the 8 bits, only 5 are used in the 8085. Figure 6.4 shows the structure of the flag register. The purpose of the flags is to supply additional information about the effects of some of the machine instructions. This allows other instructions to perform different actions depending on the value of a particular flag. The five flags are:

- sign flag (S);
- zero flag (Z);
- carry flag (CY);
- auxiliary carry flag (AC);
- parity flag (P).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
S flag	Z flag	–	AC flag	–	P flag	–	CY flag

Figure 6.4 The flag register

The *sign flag* is set to the value 1 by some instructions if the result of that operation is a negative number.

These instructions set the sign flag to 0 if the result is a positive number and to 1 otherwise. Other instructions perform an operation only if the sign flag is found to be 1 or 0. For example, 'JP START' means 'jump to address START if and only if the sign flag is 0 (that is, indicating 'positive')'.

The *zero flag* is similar in that it is set to 1 by some instructions if the result is the 8-bit code for zero (that is, 00000000 binary).

The *carry flag* is used in arithmetic operations and is set to 1 or 0 as a result of arithmetic and some other processing instructions. Like the sign and zero flags, there are a number of instructions whose operation is conditional upon the value of the carry flag.

The *auxiliary carry flag* is similar to the carry flag but is useful only if the programmer wishes to perform arithmetic using the *binary-coded-decimal* representation for numbers, rather than the 2's complement representation.

The *parity flag* is set to 1 by some instructions if the resulting 8-bit code contains an even number of bits set at 1 (that is, the code is of *even parity*). If there is an odd number of bits set at 1 (*odd parity*) the parity flag is set to 0.

6.4.4 Description of instruction set

The behaviour of each type of instruction is described below, and Tables 6.4 to 6.6 provide a summary. Any operands required by an instruction will be indicated in terms of the types of operand listed in Table 6.3 and described in Subsection 6.3.

The effects on the flags in the flag register are also described. The following format is adopted for each instruction:

---

opcode symbol operand type(s)
expanded mnemonic
number of bytes in instruction
flags affected
description of instruction
examples

Note that those instructions having an 8-bit register as an operand (Type R8 in Table 6.3), also allow the implied address mode (Type M).

ACI D8

(add immediate, with carry)

2 bytes

Z, S, P, CY, AC flags

The value D8 and the value of the CY flag are added, using 2's complement 8-bit arithmetic, to the contents of the A register. The result is left in the A register, and flags are set according to this result (see Subsection 6.4.3).

Example: ACI 14H

Assuming the A register originally contained 42 (hexadecimal) and the CY flag was 1, the result is 57 (hexadecimal). The Z, S, P, CY and AC flags are all set to 0 in this example.

ADC R8

(add with carry)

ADC M

1 byte

Z, S, P, CY, AC flags

Identical in function to ACI, except that the data added are the contents of a register or the implied address.

ADD R8

(add)

ADD M

1 byte

Z, S, P, CY, AC flags

Identical in function to ADC, except that the value of the CY flag is not used in the addition.

ADI D8

(add immediate)

2 bytes

Z, S, P, CY, AC flags

Identical in function to ACI, except that the value of the CY flag is not used in the addition.

ANA R8

(logical AND with A register)

ANA M

1 byte

Z, S, P, CY, AC flags

Each bit in the A register that was previously 1 is set to 0 if the corresponding bit of the operand specified is 0. The result sets the Z, S, P flags appropriately and the CY flag is set to 0, and the AC flag to 1.

Example: ANA B

Assuming A contained 10101010 (binary) and B contained 00001111 (binary), A would be changed to 00001010 (binary), with Z, S, CY flags set to 0 and P, AC set to 1.

ANI D8

(AND immediate with A register)

2 bytes

Z, S, P, CY, AC flags

Identical in function to ANA, except that the operand data is specified in the instruction, rather than being a register's contents.

CALL A16

(call a subroutine)

3 bytes

no flags affected

The current contents of the PC register are pushed on the stack (see Subsection 6.4.2). The address A16 is then loaded into the PC register. This allows transfer to a subroutine (at A16) in such a way that, when completed, a return can be made to the calling routine. This is performed by one of the 'RETURN' group of instructions. By using the stack to save the return address, the called subroutine can itself call other subroutines without any confusion of return addresses (see also RET instruction).

CC A16

(call if carry)

3 bytes

no flags affected

Identical to the CALL instruction, except that the action is performed if and only if the CY flag is set to 1.

CM A16

(call if minus)

3 bytes

no flags affected

Identical to the CALL instruction, except that the action is performed if and only if the S flag is 1 (indicating a negative or 'minus' condition).

CMA

(complement A register)

1 byte

no flags affected

Each bit in the A register is altered to its complementary value (1 to 0, 0 to 1). The result is said to be the 'one's complement' of the original value.

CMC

(complement carry)

1 byte

CY flag only

The CY flag is complemented (1 becomes 0, 0 becomes 1).

CMP R8

(compare with A register)

CMP M

1 byte

Z, S, P, CY, AC flags

The instruction is used to determine whether the contents of register R8 are greater than, equal to, or less than the contents of the A register. The contents of register R8 are subtracted from the contents of A. The operands are left unchanged, but the flags are set according to the result of the subtraction.

The S flag is set to the most significant bit of the result. Thus if the result is taken to be a 2's complement number, the S flag gives its sign. The Z flag is set to one only if the result is zero (i.e. the accumulator and register contents are equal). The CY flag is set according to the following table, with the register contents taken as *unsigned integers* (0–255 denary).

CY flag Set to	(A) < (R8) 1	(A) = (R8) 0	(A) > (R8) 0

Example: CMP C

If the A register contains 7F (hex) and the C register contains 93 (hex), the Z flag will be set to 0, and the S and CY flags set to 1.

CNC A16

(call if no carry)

3 bytes

no flags affected

Identical to the CALL instruction, except that the action is performed if and only if the CY flag is set to 0.

CNZ A16

(call if not zero)

3 bytes

no flags affected

Identical to the CALL instruction, except that the action is performed if and only if the Z flag is 0 (indicating a non-zero result in a previous instruction).

CP A16

(call if positive)

3 bytes

no flags affected

Identical to the CALL instruction, except that the action is performed if and only if the S flag is set to 0 (indicating a 'positive' result).

CPE A16

(call if parity even)

3 bytes

no flags affected

Identical to the CALL instruction, except that the action is performed if and only if the P flag is set to 1 (even parity).

CPI D8

(compare immediate)

2 bytes

Z, S, P, CY, AC flags

Identical to the CMP instruction, except that the value compared with the A register contents is D8, rather than a register's contents, or contents of a memory location.

CPO A16

(call if parity odd)

3 bytes

no flags affected

Identical to the CALL instruction, except that the action is performed if and only if the P flag is set to 0 (odd parity).

CZ A16

(call if zero)

3 bytes

no flags affected

Identical to the CALL instruction, except that the action is performed if and only if the Z flag is 1 (indicating a zero result in a previous instruction).

DAA

(decimal adjust A register)

1 byte

Z, S, P, CY, AC flags

This instruction adjusts the contents of the A register when binary-coded decimal (BCD) arithmetic is being used, instead of the more generally useful 2's complement arithmetic. It uses the values of the CY and AC flags, following an add operation, to produce the effect of BCD addition.

Example: ADI 73H; add 73 (BCD)
DAA ; adjust to BCD

Assuming the A register originally contained 08 (hex), then following the ADI, the A register would contain 7B (hex). The DAA would then convert this to 81 (hex), which is the correct result in terms of BCD numbers.

DAD R16a

(double register add)

1 byte

CY flag only

This instruction adds the 16-bit value in the specified register pair to the 16-bit contents of the H, L register pair, leaving the result in H, L. The CY flag reflects the result of this 16-bit 2's complement addition.

Example: DAD B

Assuming H, L contained FFFF (hex) and B, C contained 0001 (hex), the result would be 0000 (hex), and the CY flag would be set to 1.

DCR R8

(decrement register)

DCR M

1 byte

Z, S, P, AC flags only

The contents of the specified operand are decremented (1 is subtracted). Note that the CY flag is unaffected.

DCX R16a

(decrement register pair)

1 byte

no flags affected

The 16-bit contents of the specified register pair are decremented (1 is subtracted)

DI

(disable interrupts)

1 byte

no flags affected

This instruction sets one of the bits (bit 3) in the interrupt register (I) to 0. When this bit is set to 0, all interrupts, except the TRAP interrupt (Section 9) are ignored by the 8085 microprocessor.

EI

(enable interrupts)

1 byte

no flags affected

This instruction is the inverse of the DI instruction. Following EI, bit 3 of the I register is set to 1, and the microprocessor responds to interrupts (Section 9).

HLT

(halt execution)

1 byte

no flags affected

This instruction prevents the microprocessor from continuing with the execution of the next instruction. The only way to restart it is via an external interrupt (Section 9) or a system reset (Section 2).

IN A8

(input from I/O device)

2 bytes

no flags affected

This instruction loads an 8-bit data item from the I/O device whose I/O address is A8 into the A register.

INR R8

(increment register)

INR M

1 byte

Z, S, P, AC flags only

The contents of the specified operand are incremented (1 is added). Note that the CY flag is unaffected.

INX R16a

(increment register pair)

1 byte

no flags affected

The 16-bit contents of the specified register pair are incremented (1 is added)

JC A16

(jump if carry)

3 bytes

no flags affected

Identical to the JMP instruction, except that the action is performed if and only if the CY flag is set to 1.

JM A16

(jump if minus)

3 bytes

no flags affected

Identical to the JMP instruction, except that the action is performed if and only if the S flag is 1 (indicating negative or 'minus').

JMP A16

(jump)

3 bytes

no flags affected

This instruction loads the PC register with the value A16. This means that the next instruction to be executed will be that stored at address A16 in memory.

JNC A16

(jump if no carry)

3 bytes

no flags affected

Identical to the JMP instruction, except that the action is performed if and only if CY is set to 0.

JNZ A16

(jump if non-zero)

3 bytes

no flags affected

Identical to the JMP instruction, except that the action is performed if and only if the Z flag is 0 (indicating a non-zero result in a previous instruction).

JP A16

(jump if positive)

3 bytes

no flags affected

Identical to the JMP instruction, except that the action is performed if and only if the S flag is 0 (indicating positive).

JPE A16

(jump if parity even)

3 bytes

no flags affected

Identical to the JMP instruction, except that the action is performed if and only if the P flag is 1 (even parity).

JPO A16

(jump if parity odd)

3 bytes

no flags affected

Identical to the JMP instruction, except that the action is performed if and only if the P flag is 0 (odd parity).

JZ A16

(jump if zero)

3 bytes

no flags affected

Identical to the JMP instruction, except that the action is performed if and only if the Z flag is 1 (indicating a zero result in a previous instruction).

LDA A16

(load A register direct)

3 bytes

no flags affected

The contents of the memory location whose address is A16 are loaded into the A register.

LDAX R16c

(load A register indirect)

1 byte

no flags affected

The contents of the memory location whose address is contained in the register pair specified by the R16c operand (see Table 6.2) are loaded into the A register.

LHLD A16

(load H and L direct)

3 bytes

no flags affected

The contents of the memory location whose address is A16 are loaded into the L register. The contents of the memory location whose address is one more than A16 are loaded into the H register. Note that when 16-bit data items are stored in memory, the convention is that the least significant 8 bits are stored at the next lower address than the most significant 8 bits. If this convention is adhered to, after an LHLD instruction, the H register will correctly hold the most significant 8 bits of the 16-bit value loaded.

LXI R16a, D16

(load register pair immediate)

3 bytes

no flags affected

The 16-bit data item D16 is loaded into the specified register pair.

Example: LXI H,0123H

After this instruction, H will contain 01 (hex) and L will contain 23 (hex).

Example: LXI SP,2800H

This instruction defines the current top of stack as being at address 2800 (hex).

MOV R8,R8

(move)

MOV R8,M

MOV M,R8

1 byte

no flags affected

The data in the second operand register (or at the implied address) is loaded into the first operand register (or into memory at the implied address).

Examples: MOV A,B; load B into A.
MOV M,C; store C in memory at the address in H,L.
MOV L,M; load L from memory using address in H,L.

MVI R8,D8

(move immediate)

MVI M,D8

2 bytes

no flags affected

The data D8 is loaded into the specified register (or into memory at the address specified by the H, L register pair).

Examples: MVI A,32H; load A with 32 (hexadecimal).
MVI M,27; store 27 (decimal) at the address specified by contents of H,L.

NOP

(no operation)

1 byte

no flags affected

This instruction does nothing except use up a little time.

ORA R8

(inclusive OR with A register)

ORA M

1 byte

Z, S, P, CY, AC flags

Each bit in the A register that was previously 0 is set to 1 if the corresponding bit in the specified operand is 1. The result sets the Z, S, P flags appropriately and the CY and AC flags to 0.

Example: ORA H

Assuming A contained 10101010 (binary) and H contained 00001111 (binary), A would be changed to 10101111 (binary) with Z, CY, AC set to 0 and S, P to 1.

ORI D8

(inclusive OR immediate)

2 bytes

Z, S, P, CY, AC flags

Identical to the ORA instruction, except that the operand data is specified as D8, rather than as the contents of a register.

OUT A8

(output to I/O device)

2 bytes

no flags affected

This instruction transmits the contents of the A register to the I/O device whose I/O address is A8.

PCHL

(load program counter from H, L)

1 byte

no flags affected

The contents of the H,L register pair are loaded into the PC register, which causes program execution to continue at the address specified by the H, L register pair.

POP R16b

POP PSW

(pop a 16-bit data item from the stack)

1 byte

flags possibly affected (see discussion)

The content of the memory location, whose address is specified by the content of register SP, is moved to the low-order register of register pair R16b. The content of the memory location, whose address is one more than the content of register SP, is moved to the high-order register of register pair R16b. The content of register SP is incremented by 2 (Subsection 6.4.2). The flags are not affected except for the case when the *flag register* is loaded, together with the A register, by the POP PSW instruction.

PUSH R16b

PUSH PSW

(push a 16-bit data item on the stack)

1 byte

no flags affected

The content of the high-order register of register pair R16b is moved to the memory location whose address is one less than the content of register SP. The content of the low-order register of register pair R16b is moved to the memory location whose address is two less than the content of register SP. The content of register SP is decremented by 2 (Subsection 6.4.2).

RAL

(rotate A register left)

1 byte

CY flag only

The contents of each of the bits in the A register are moved one bit position to the left. The previous CY value is loaded into the rightmost (least significant) bit of the A register, and the new CY value is the previous value of the leftmost bit of the A register.

Example: If the previous A register contents were 00111100 (binary), with the CY flag set to 1, then 01111001 (binary) is the new value with the CY flag set to 0.

RAR

(rotate A register right)

1 byte

CY flag only

Similar to the RAL instruction, except that the data movement is to the right. The old CY value is loaded into the leftmost A register bit and the old value of the rightmost A register bit is loaded into the CY flag.

RC

(return if carry)

1 byte

no flags affected

Identical to the RET instruction, except that the action is performed if and only if the CY flag is set to 1.

RET

(return from subroutine)

1 byte

no flags affected

The 16-bit item currently on top of the stack is popped off (Section 6.4.2) and loaded into the PC register. Assuming that this data item was originally pushed onto the stack by the matching CALL instruction, the RET instruction therefore has the effect of resuming execution of the calling routine at the next instruction following the CALL instruction.

Example:

```
        ...             ;calling routine:
        CALL SUB1       ;execute subroutine SUB1
        ...             ;then do this
        .
        .
        .
        .
        .
        .
SUB1:   ...             ;subroutine SUB1
        .
        .
        .
        .
        .
        .
        RET             ;return to calling routine
```

Note that the subroutine labelled SUB1 can itself call other subroutines; the stack mechanism allows return addresses to be pushed and popped in the correct order.

RIM

(read interrupt register)

1 byte

no flags affected

The contents of the interrupt register (I), are loaded into the A register. This data contains information concerning the state of three of the 8085 interrupts, the state of the interrupt enable flag, and the state of the serial input line (see Section 9).

RLC

(rotate A register left with carry)

1 byte

CY flag only

The contents of each of the bits in the A register are moved one bit position to the left. The leftmost bit of the A register is loaded into its rightmost bit and into the CY flag.

Example: If the previous A register contents were 00111100 (binary), with the CY flag set to 1, then 01111000 (binary) is the new value with the CY flag set to 0.

RM

(return if minus)

1 byte

no flags affected

Identical to the RET instruction, except that the action occurs if and only if the S flag is 1 (indicating negative or 'minus').

RNC

(return if no carry)

1 byte

no flags affected

Identical to the RET instruction, except that the action occurs if and only if the CY flag is 0.

RNZ

(return if not zero)

1 byte

no flags affected

Identical to the RET instruction, except that the action occurs if and only if the Z flag is 0 (indicating a non-zero result in a previous instruction).

RP

(return if positive)

1 byte

no flags affected

Identical to the RET instruction, except that the action occurs if and only if the S flag is 0 (indicating positive).

RPE

(return if parity even)

1 byte

no flags affected

Identical to the RET instruction, except that the action occurs if and only if the P flag is 1 (even parity).

RPO

(return if parity odd)

1 byte

no flags affected

Identical to the RET instruction, except that the action occurs if and only if the P flag is 0 (odd parity).

RRC

(rotate right with carry)

1 byte

CY flag only

Similar to the RLC instruction, except that the data movement is to the right. The rightmost bit of the A register is loaded into its leftmost bit and into the CY flag.

RST A3

(restart)

1 byte

no flags affected

This instruction is effectively a one-byte CALL instruction, where the address of the called subroutine is given by A3 multiplied by 8. The only assembly language form allowed for A3 is a single decimal digit in the range 0–7.

Example: RST 7

Calls the subroutine at address $7 \times 8 = 56$ (decimal) = 38 (hex).

RZ

(return if zero)

1 byte

no flags affected

Identical to the RET instruction, except that the action occurs if and only if the Z flag is 1 (the result of a previous instruction was zero).

SBB R8

(subtract with borrow)

SBB M

1 byte

Z, S, P, CY, AC flags

The contents of the specified operand are subtracted, together with the CY flag value (acting as a 'borrow'), from the contents of the A register, using 2's complement arithmetic. The result is left in the A register and the flags are set according to this result.

Example: SBB B

Assuming the A register originally contained 4, the B register 2, and there is a borrow (the CY flag is set to 1), then the result will be 1. The Z, S, P and CY flags will be set to 0, and the AC flag to 1.

SBI D8

(subtract immediate with borrow)

2 bytes

Z, S, P, CY, AC flags

Identical to the SBB instruction, except that the operand value D8 is subtracted, rather than the contents of a register.

SHLD A16

(store H and L direct)

3 bytes

no flags affected

This instruction is the inverse of the LHLD instruction. The contents of the L register are stored at address A16, and the contents of the H register at the next higher address.

SIM

(set interrupt register)

1 byte

no flags affected

Like the RIM instruction, this is a multipurpose instruction, which uses the various bits of the A register to set or

clear individual bits of the interrupt register (I), clear one of the interrupt lines, and output a value on the serial output line from the microprocessor (see Section 9).

SPHL

(load SP register with H, L)

1 byte

no flags affected

The stack pointer (SP) is loaded with the 16-bit contents of the H, L register pair.

STA A16

(store A register direct)

3 bytes

no flags affected

This instruction is the inverse of the LDA instruction. The A register's contents are stored in memory at the operand address A16.

STAX R16c

(store A register indirect)

1 byte

no flags affected

This instruction is the inverse of the LDAX instruction. The A register's contents are stored in memory at the address specified by the operand register pair.

STC

(set carry)

1 byte

CY flag only

The carry flag (CY) is set to 1.

SUB R8

(subtract)

SUB M

1 byte

Z, S, P, CY, AC flags

This instruction is identical to SBB, except that the CY flag's value is not used in the subtraction.

SUI D8

(subtract immediate)

2 bytes

Z, S, P, CY, AC flags

This is identical to the SUB instruction, except that the value subtracted is the operand value D8, rather than a register's contents.

XCHG

(exchange H, L with D, E)

1 byte

no flags affected

The 16-bit contents of the H, L register pair are interchanged with the 16-bit contents of the D, E register pair.

XRA R8

(exclusive OR with A register)

XRA M

1 byte

Z, S, P, CY, AC flags

Each bit on the A register that was previously different from the corresponding bit in the specified operand is set to 1. If the bit was the same as the corresponding bit, it is set to 0. The result sets the Z, S, P flags appropriately, and the CY and AC flags to 0.

Example: XRA C

Assuming A contained 10101010 (binary) and C contained 00001111 (binary), A would be changed to 10100101 (binary), with the Z, CY, and AC flags set to 0 and the S and P flags set to 1.

Example: XRA A

This is an easy way to ~~set~~ CLEAR each bit in the A register and the CY flag to 0.

XRI D8

(exclusive OR immediate)

2 bytes

Z, S, P, CY, AC flags

Identical to the XRA instruction, except that the operand data is D8, rather than a register's contents.

XTHL

(exchange H, L with top-of-stack)

1 byte

no flags affected

The 16-bit data item, stored in memory starting at the location whose address is in the SP register, is exchanged with the 16-bit contents of the H, L register pair. The stack pointer value is unaffected.

6.5 Use of the assembler

An assembly-language program is prepared as a set of source lines, using the editor program (Section 5). Assuming the program is in the editor's text buffer, using the editor's A command will cause execution of the ROM-based assembler. The assembler will translate the source lines in the editor's text buffer and produce four types of information:

- user symbol table;
- machine-code version of the program;
- program listing;
- error messages, if any.

These are each discussed below.

6.5.1 User symbol table

Some of the symbols in the source program are *permanent symbols*; that is, their interpretation is fixed. Permanent symbols include the mnemonic opcodes (for example, MOV, PUSH) and the symbols denoting registers or register pairs (A, SP). The remaining symbols are created by the programmer, and a task of the assembler is to make a list of these symbols, and assign values to them. Then when the machine-code instructions are generated, the 8-bit or 16-bit values of these symbols can be used.

This overall task is split into three phases, each involving a scan of the whole source program from

beginning to end by the assembler. During the *first pass*, the assembler simply makes a list of the user-defined symbols, and stores the symbols as strings of characters in the *user symbol table* in read/write memory. During this pass, the assembler may discover errors such as incorrect forms of labels, and will display error messages (Subsection 6.5.4) on the screen to that effect.

During the *second pass*, the assembler assigns values to the symbols. The directly specified symbols, using the EQU directive, are clearly straightforward, but the label symbols can only be evaluated when the *origin* of the whole program is known and when the *number of locations* which are required for each machine-code instruction is known. The origin is either user-defined (Subsection 6.2.3) or is assigned by the assembler as being the next unused location in memory after the editor text buffer and the user symbol table. The number of locations required for each instruction is deduced from the opcode mnemonic (the operand values do not affect this number). During Pass 2 other errors may be revealed and reported, such as the use of an undefined opcode mnemonic. After Pass 2, all symbols will now have been listed and associated with a specific value or code.

6.5.2 Machine-code program

During the *third* and final pass, the machine code is produced, instruction by instruction, and stored in the *machine-code buffer* (the free area of memory following the text buffer and the user symbol table). If the origin has been assigned by the assembler, the machine-code program can be executed without moving it from the buffer. If the origin is user-defined, the code will have to be moved, after assembly, so that the first location that it occupies is the defined origin. During the third pass the operand information is translated, as well as the opcode mnemonic, thus fully defining each machine-code instruction. Any errors in operand specification will be revealed at this stage.

6.5.3 Program listing

An additional activity during Pass 3 is the production of the *program listing* (it is optional; see Subsection 6.5.5). This is a line-by-line display of the source text and the machine-code equivalent as generated by the assembler. The original source lines are formatted into columns, one column for each of the four fields. This presentation is not only more readable than the original source text, but also helps to identify some errors. (For example, if a 'label' appears in the opcode column, it is because the assembler has interpreted this symbol as an opcode mnemonic.)

FIG 6.5 shows a fragment of a listing produced by the HEKTOR assembler.

6.5.4 Error messages

The HEKTOR assembler has a repertoire of nine error messages. These are somewhat cryptic, so they are amplified below. They should also not be taken too literally, since they indicate the *most likely* source of the error, rather than its actual cause. All that is certain is that the assembler has had trouble with translating the line in question, either because of some aspect of the line itself or due to some interaction with other assembly-language lines in the same program.

'Bad label' means that a user-defined label symbol appears to violate the allowed form. That is, labels should consist of up to six alphanumeric symbols, of which the first should be alphabetic. The label should be followed immediately by a colon delimiter.

'Bad inst.' means that the assembler cannot recognize the opcode (or pseudo-operation) mnemonic. This may be due to the symbol in fact not being an allowed opcode mnemonic, or it may be that the assembler has not delimited the opcode field in the way the programmer intended. (See Subsection 6.4.4 for allowed opcode mnemonics, Subsection 6.2.3 for pseudo-operations, and Subsection 6.2 for allowed forms of field delimiting.)

```
memory    machine      editor     label        opcode    operand         comment
address   code         line no.   field        field     field           field


                       0060    ;
                       0061    ; MOTOR  DRIVE:
                       0062    ; USES   C  AS  TABLE  INDEX
                       0063    ;
18D6   0600            0064    DRIVE:   MVI    B,0
18D8   211640          0065             LXI    H,MOTAB     ; TABLE
18DB   09              0066             DAD    B           ; ADD  INDEX
18DC   7E              0067             MOV    A,M         ; GET  VALUE
18DD   320840          0068             STA    MDVO        ; SAVE
18E0   C9              0069             RET
```

Figure 6.5

'Bad register' means that the assembler is expecting, from its interpretation of the opcode mnemonic, one or more register operands of a particular type, and the symbols it has encountered do not belong to the type in question. Again, this may be due to improper delimiting as well as to use of the wrong symbol. (See Subsection 6.3.4 for register operand types, and Subsection 6.2 for field delimiting.)

'Missing opnd' means that one or more of the expected operands (for the opcode in question) appear to have been omitted. Apart from obvious causes, this may again be a delimiting problem.

'Undef. symbol' is a common error in early versions of large programs. It means that a user-defined symbol in the operand field has not been defined as a label or by an 'equate' directive. The symbol may have been mistyped or omitted, or the assembler may be attempting to translate an incorrectly specified number as a symbol. There are several other causes.

'Rep. label' means that the same symbol has been used to define labels in more than one place in the program. This is disallowed as each symbol must have a unique value.

'Forwd. ref' means that the assembler is trying to translate a symbol in the operand field which it has recognized as a valid user-defined symbol, but for which a value has not yet been defined. This only occurs with assembler directive lines, because they, unlike instruction lines, have to be fully translated during Pass 2.

'Bad number' means that a number in the operand field cannot be translated into a valid 8-bit or 16-bit code. Common causes of this error are omitting the 'H' delimiter from the hexadecimal number, not starting it with a numeric digit, or mistyping a digit character. Note that the assembler does not check that the value of the evaluated operand lies within the expected bounds. For example, MVI A,4003H will not produce an error message – the assembler will simply use the least significant 8 bits of the operand, 03H in this case, in the machine-code instruction.

Finally, the error message *'store full'* means that there is not room in the read/write memory for the source text, the symbol table, and the machine-code buffer. For typical programs, this becomes a hazard only when approaching some 300 lines of source program.

6.5.5 Assembler options

The editor command to execute the assembler (the A command) allows for optional variations in the assembler's behaviour during assembly. The A command can include *switches* which specify these options. Each switch is a single character (one of L, M, S, T, W) preceded by a back slash character '\'. The switches can be in any order in the command line, so the following are all examples of valid A commands: A (by itself); A\S; A\L\W\T; A\M\T\S. The effect of each switch is described below.

'\L' causes the program *listing* (Subsection 6.5.3) to be displayed during the assembler's Pass 3. (If this switch is omitted, there is no listing produced.) As an aid to readability during listing, the assembler will wait if the RETURN key is held down for a second or two and then released. A second use of the RETURN key allows the listing display to continue.

'\M' directs the assembler to *load* the machine code from the buffer into the area of memory where it is expected to execute (Subsection 6.5.2), and then to return to the *monitor*. Having returned to the monitor, the program can immediately be executed using the G command. Note that using G by itself, without an address being specified, will cause execution to begin at the program's *entry point* (Subsection 6.2.3).

'\S' causes the assembler to display the user *symbol* table that it has generated, after it has completed the assembly. Also included in this display are the program's *origin* and *entry point*.

'\T' causes the assembler to save the assembled machine-code on cassette *tape*. The program is saved on tape in such a way that when subsequently loaded again (using the monitor's L command), it is loaded into the area of memory specified by its true origin (that is ready for execution) rather than back into the assembler's machine-code buffer.

'\W' causes the assembler to *wait* during assembly if it encounters (and displays) an error condition. The user can absorb the displayed message, and then cause the assembler to continue by using the RETURN key.

7 USING HEKTOR's BASIC LANGUAGE

7.1 Introduction to BASIC

This section contains a complete description of the version of BASIC available in HEKTOR. It is not meant as an introductory teaching text, and assumes some familiarity with concepts of programming and BASIC. However, the assumed familiarity is kept to a minimum.

The BASIC facilities of HEKTOR are described in two parts. Subsection 7.2, 'The BASIC command mode', describes how a program is to be handled, and answers such questions as: How do you write and correct program lines? How do you run programs, or store them on a cassette? Subsection 7.3, 'Writing BASIC programs', contains the details of the language itself: i.e. What is the structure of a BASIC program? What is its syntax? What variables, functions, instructions, etc. can be used?

The various facilities in BASIC are grouped according to their function to enable you to find what you are looking for even if you don't know its name. However, to help you find a precise description of a given term quickly, use the short index below.

7.1.1 Short index of BASIC keywords

	page		page		page
ABS	62	LEN	62	ROW	60
ASC	62	LET	60	RUN	56
BAUD	60	LIST	56	SAVE	58
CHR$	62	LOAD	58	SCROLL	61
CLOSE	64	MID$	62	SIZE	60
COL	60	NEW	56	SOUND	64
DELETE	56	NEXT	63	STAT	60
DIM	59	OPEN	64	STOP	59
FOR	63	PRINT	63	STR$	62
GOSUB	62	PW	61	SW	60
GOTO	62	RANDOM	61	TAB	64
IF	63	REM	59	THEN	63
INPUT	64	RETURN	62	TIME	61
KSTAT	60	REWIND	58	VAL	62
LED	60	RIGHT$	62	VERIFY	58
LEFT$	62	RND	62		

HEKTOR will also accept abbreviations of these keywords, consisting of enough characters to specify the keyword unambiguously followed by '.'.

Example: REW. can be used for REWIND.

7.2 The BASIC command mode

7.2.1 Introduction: What is the BASIC command mode?

When you switch on HEKTOR, the monitor program comes into operation. It will accept a variety of commands which enable you to make use of many of HEKTOR's facilities. Similarly, to write, correct, run or store programs in BASIC you also need to be able to give HEKTOR appropriate commands. These commands are available in a program called the *BASIC interpreter*. To activate the BASIC interpreter from the monitor, key the command H RETURN (for 'high-level language'). You will then be in the BASIC *'command mode'*. The remainder of this section describes the commands available to you in this mode.

To leave the command mode, use the command MON, which returns you to the HEKTOR monitor. Pressing RESET also returns control to the MONITOR. In either case, the MONITOR's W command (see p.29) allows return to the BASIC command mode with lines of BASIC intact. The H command also returns to the BASIC interpreter but deletes any lines of BASIC previously stored.

When you are in BASIC command mode, the symbol '*' is displayed as a prompt to indicate that the BASIC interpreter is waiting for a command from you. This *BASIC command mode prompt* is analogous to the monitor prompt, '>'. You can tell whether HEKTOR is under the control of the monitor program or the BASIC interpreter by noting which prompt is displayed.

7.2.2 An introductory example

The following example illustrates the way in which a program is written and executed. It introduces concepts which will be described more fully later.

The lines

```
10 LET X = 2 + 2 RETURN
20 PRINT X RETURN
```

form a simple BASIC program. If you type them in, and then type (on a new line) RUN RETURN, HEKTOR will respond with '4'. HEKTOR does not execute the program until the command 'RUN' is given. Note that both lines of the program start with a *line number.*

Note also that in the BASIC command mode HEKTOR does not respond to any line or command until the RETURN key is pressed. From now on RETURN will be taken as understood, and omitted except where really necessary.

On the other hand, if you type in the two statements above *without* their line numbers, HEKTOR will respond with '4' immediately, without the command RUN. Statements without line numbers are called *'immediate execution'* statements. They are useful for quick calculations, for checking out the features of BASIC and, especially, for finding errors in programs (see Subsection 7.2.5). However, immediate execution statements are not stored and so cannot be changed or run again without re-typing them in completely.

If you write a BASIC program with line numbers, and wish to review what you have written, type the command 'LIST'. HEKTOR will respond with a listing of all the lines written so far, in order of their line numbers. Line numbers determine the order in which statements are listed and executed (unless otherwise directed), and line numbers remain fixed, as typed.

7.2.3 Storing and editing programs

Before a BASIC program can be executed it must be stored in HEKTOR's main memory. This may be done either by typing it in from the keyboard, or by reading it in from cassette tape. (For the latter, see Subsection 7.2.7.) You can create space for a new program in memory by typing the command NEW.

NEW

The command NEW deletes all previous lines of program in preparation for storing a new one.

Typing in a program

Lines of a program may be typed in at any time when the BASIC prompt '*' is displayed. A program line always starts with a line number and terminates when you press the [RETURN] key. A line may extend over several rows of the display screen, provided the total number of characters in that program line does not exceed 256. New lines may be added to the current program at any time and in any order of line numbers. BASIC keywords should always be typed using upper-case letters.

Changing and correcting single lines of program

There are several ways of making corrections to a program line. If you notice a mistake as you are typing it in, the back-space key [←] will delete the error. You can then type the correct character. If you wish to remove a line entirely, type its line number, followed by [RETURN]. If you wish to change a line completely, type its line number and then the new version of the line. The previous version will be automatically deleted when you press [RETURN].

To make corrections to a line you have previously typed in (and which is thus stored in HEKTOR's memory), start by typing the line number. Then press [→] repeatedly. At each press, a letter from the stored version of the line will be displayed. You can add characters to this new displayed version from the keyboard where appropriate, and delete incorrect characters using the [←] key. If you type [RETURN] (or type [→] after all the characters in the stored version have been displayed) the new displayed version will replace the old stored version in memory. (Thus any characters from the stored version which have not been displayed will be lost. Note that this is different from the behaviour of the assembly language editor.)

Example: Suppose you have typed

```
40 PRNT X, Y
```

and now wish to correct the typing error in the word PRINT. Type in the line number (40), and then press the [→] key twice. The result will be

```
40 PR
```

You can then type I, and then press [→] repeatedly until the remainder of the line appears

```
40 PRINT X, Y
```

This new version will replace the old version when [RETURN] is pressed.

DELETE n, m

This command deletes all lines with numbers in the range 'n' to 'm' inclusive. There need not necessarily be lines of program with the numbers 'n' or 'm'.

Examples:

DELETE 10, 1000	deletes all lines with numbers from 10 to 1000.
DELETE 40, 40	deletes line 40 only
DELETE 60 DELETE 35, 25	are invalid

LIST n

This command lists the program lines starting with line number n. If n is omitted, the first line of the program is used. The number n needn't actually be a line number in the program. Listing can be temporarily interrupted by holding down the space bar. It can be stopped completely by pressing the [BREAK] key.

Examples: LIST, LIST 100, LIST 25 are all valid
LIST 10, 20 is invalid

If you terminate a LIST statement with [CTRL]-[P] (see Appendix C) instead of [RETURN], the listing will be sent to the serial line instead of to the screen. This is useful, for example, if a printer is connected to the serial line.

Warning: Do not use LIST with [CTRL]-[P] if no device is connected to the serial lines, as this will cause HEKTOR to wait indefinitely for control signals. To return to BASIC command mode from this state you will first have to press [RESET], which will delete your program.

7.2.4 Running programs

RUN

This command will start the execution of a program, beginning at the lowest line number. It also initializes all variables (see Subsection 7.3.3). HEKTOR will then execute the program until:

(a) it encounters a STOP statement within the program,

or

(b) it runs out of statements to execute,

or

(c) it encounters an error in the program.

(HEKTOR will also temporarily stop executing a program when it executes an INPUT statement. Then it will wait for you to type in some information. See Subsection 7.3.7 for details.)

The end of execution will be signalled by a prompt, '*', or an ERROR message.

Some programs have no natural end, as in the following example:

Example:

```
10 PRINT 30 + 20
20 GOTO 10
```

This program will print '50' on line after line indefinitely. It can be stopped by pressing the [BREAK] key. You can tell that a program is still running by the absence of the prompt '*'. When a program stops for any of the reasons (a) – (c) above, the values of the variables at the time it was stopped are still stored and thus are available to you.

GOTO n

This command starts execution of a program at line number n. It does *not* alter the values of the variables when doing so. Thus it can sometimes be used to re-start a program where it left off after being stopped. (Using RUN would initialize all variables first.) It can also be used to start a program in the middle. (Note that there should be no space between GO and TO.)

7.2.5 Debugging: Finding program errors

Typing a program into HEKTOR and running it is no guarantee that the program will produce the results you expect of it. The BASIC interpreter provides various facilities to help find the mistake.

Programs which don't stop

It is possible for a program to get into a loop with no exit point, or to get stuck in other ways. Generally, pressing [BREAK] will cause execution to stop and a prompt '*' to be displayed. In extreme cases where this does not work you may have to resort to [RESET], which will also delete your program.

Error messages

Any time the BASIC interpreter tries to execute a command or BASIC statement which does not agree with its rules it will produce an error message, including a code. The error messages are of two main forms, *syntax errors* and *execution errors*. A syntax error is an error in the way a statement is put together, incorrect use of symbols, etc. A list of error messages is given in Appendix E, in code order.

Example:

```
10 PRINNT X
RUN
SYNTAX ERROR 1978
10 P?RINNT X
```

HEKTOR will print an error message and will print out the offending line, usually with '?' to indicate the point at which it was unable to make sense of the statement. In this case, since it did not recognize PRINNT as the keyword PRINT, it assumed that P was a variable and did not know why it was followed by other letters!

In most cases it will be obvious why the statement was incorrect. However, if it is not, you can refer to Appendix E for the meaning. In this case, error 1978 means 'equal sign expected'.

Sometimes all of the statements will be valid in themselves, but when they are executed HEKTOR may be asked to do something it cannot. This gives rise to an execution error.

Example:

```
10 PRINT 2 + 2
20 GOTO 15
RUN
    4
EXECUTION ERROR 614
20 GOTO 15
```

From Appendix E, error 614 is 'GOTO non-existent line'.

A third, and comparatively rare form of error message is the *implementation error*. It occurs when your instructions cannot be carried out due to a limitation of the BASIC interpreter (for example, too many nested parentheses in an expression).

Immediate execution statements

Sometimes a program will run without error messages, but it will be clear from the output that the results are not what they should be. To find the errors, it is necessary to find out exactly what the computer is doing, so that you can spot where it has diverged from what you expected it to do.

One way to do this is to insert STOP statements at various key points in the program. When the program stops you can use immediate-execution statements (PRINT X, Y and GOTO n) to check the values of important variables and then restart the program. You can also use immediate-execution assignment statements (LET X = 20) to set the variables to their correct values to see if the remainder of the program works correctly. Alternatively, you can insert PRINT statements in the program itself, with or without STOPs, to see what happens at key points.

Control characters

The ASCII characters whose codes are 00-1F (hex) are called *control characters* (see Appendix C). Only some of these are used in HEKTOR, but all are available for use if desired. Some have special keys on the keyboard ([RETURN], [HOME], [→], [←], [↑], [↓]). All can be obtained by holding down [CTRL] and, while it is held, pressing some other key. The key to use is the one for which the ASCII code is 40 (hex) greater than that of the desired key. For example, [CTRL] with [M] is the same as [RETURN], and would normally be written [CTRL]-[M].

7.2.6 Format and character control

Upper- and lower-case characters

When you first enter the BASIC command mode the keyboard will only produce upper-case letters. The shift key will have an effect only on numbers and most of the symbol keys. Pressing [CTRL]-[S] enables the keyboard to produce both upper- and lower-case letters. It will revert to all upper-case whenever [RETURN] is pressed or, within a line, if you press [CTRL]-[S] a second time.

Graphics characters

In addition to the usual characters in the ASCII table, HEKTOR can produce a set of graphic characters, as shown in Appendix D. In BASIC these can be obtained in two ways.

Example

```
PRINT CHR$ (241)
```

will display the ◆ symbol. CHR$ is suitable for displaying single characters, or for use in programs to produce patterns. Alternatively, strings of graphics characters may be obtained by keying [CTRL]-[G] (for lower-case) or [CTRL]-[G] and [CTRL]-[S] (for upper-case), as explained in Appendix D.

Examples

PRINT "[CTRL]-[G] F [CTRL]-[G]"

will display the → symbol, corresponding to capital F.

PRINT "[CTRL]-[G] [CTRL]-[S] FQ [CTRL]-[S] [CTRL]-[G]"

will display **6**◆, corresponding to lower-case f and q.

Note that when using [CTRL]-[G] in this way, the keys [↑], [↓], [→], [←], [HOME] and [RETURN] all produce graphics characters. To return to the usual character set, key [CTRL]-[G].

7.2.7 Saving and loading programs on cassette tape

This section explains how to save a complete program on tape and how to read back into memory (load) a program you have previously stored on tape. These operations are done using the SAVE and LOAD commands respectively. To check that a program you have just saved has been stored correctly, use the VERIFY command. To rewind the tape use the REWIND command. Note that there is another use for tape, which is not considered in this section, namely to transfer data to or from a tape while a program is running. This is explained in Subsection 7.3.3 'Input/output instructions'.

SAVE *'name'*

This command saves a complete program on tape and labels it there with *'name'*. This is so that HEKTOR can distinguish it from other programs stored on the same tape with other names. Before using it, insert a tape into your recorder either fully rewound or following previously recorded programs. (If the recorder is connected to HEKTOR you will need to use the REWIND command to enable you to operate it, as explained below.) Then type the command

SAVE *'name'*

The name used can be any string of characters up to 200 long.

Examples: SAVE 'TEST PROGRAM'
SAVE '16JULY'

HEKTOR will respond with 'SET RECORD'. You should then set the controls on your tape recorder to record, and press RETURN. HEKTOR will then record all lines of BASIC program that are currently stored in its memory. The recording of the program itself will be preceded by a high-pitched tone (which will be audible through your TV loudspeaker if the volume is turned up), and followed by a lower-pitched hum. The program itself sounds like a long scratchy buzz.

When the recording is finished HEKTOR will stop the recorder and display a prompt, '*'. Be sure to write down the name you have used to label the program, preferably on the cassette itself. If you forget it, you will not be able to load the program!

REWIND

The REWIND command allows you to operate the tape recorder's controls manually. This is necessary, for example, to rewind a tape before and after a program has been saved on it, or to prepare for loading a program from it. When you have finished rewinding, or otherwise using the recorder's controls, press the RETURN key a second time to inform HEKTOR that you are ready. It will respond with the prompt, '*'.

VERIFY *'name'*

The VERIFY command tells HEKTOR to check a file that you have just recorded on tape. If verification is successful then it is likely that the program has been saved correctly, and it is highly likely that it can be loaded successfully later on.

HEKTOR will respond with 'SET TO PLAY'. You should then set the controls on your recorder to playback, and press RETURN.

HEKTOR will search the tape for files labelled with *'name'*, so the rules for program names in the VERIFY command are the same as in the SAVE command (e.g. up to 200 characters long). It will continue searching until it finds a correctly labelled file. A successful verification is indicated by the display of the BASIC prompt, and by the absence of any error message. If HEKTOR has obviously gone past the point where your program was recorded, you will have to stop it using the BREAK key. If this happens, check that the name you used was correct. Otherwise, it is likely that the program was not saved successfully.

Examples: VERIFY 'SAMPLE'
VERIFY 'TM222 PROJECT'

LOAD *'name'*

This command tells HEKTOR to load the BASIC program file labelled *'name'* from tape into its memory. If you simply wish to continue work on a program previously stored on tape you must make sure there is no other program already stored in internal memory. To do this, use the NEW command before using LOAD. If there is another program stored in HEKTOR's memory, the new program will be appended to the end of the first program. This can be useful in some circumstances, but it is important that line numbers in the new program are higher than any in the old program.

As with VERIFY, the rules for the label *'name'* are the same as for the SAVE command. Again, HEKTOR will search the tape until it finds the correctly labelled file. An unsuccessful search can be stopped by pressing BREAK.

Examples: LOAD 'PLOT PROGRAM',
LOAD 'VERSION 14B'

7.3 Writing BASIC programs

7.3.1 The structure of a BASIC program

A BASIC program consists of a sequence of lines containing statements in BASIC, as shown in the following example:

Example:

```
10 REM THIS PROGRAM CALCULATES 2 + 2
20 X = 2 + 2
30 PRINT X
40 STOP
```

This example illustrates the use of line numbers, comments or remarks, and the STOP statement, which we will now look at in more detail.

Line numbers

If a statement in BASIC is to form part of a program, it must start with a line number. If it has no line number, it will be executed immediately after the RETURN key is typed, instead of when the program is run. Line numbers must be positive integers between 1 and 32767. It is common practice to number lines in multiples of 10, to make it easy to insert extra lines, should this become necessary.

REM *remarks or comments*

Any line beginning with REM will be ignored when the program is executed. Such lines are used to help make the program more understandable. An alternative to REM is '!'. For example, line 10 above could be written:

```
10 ! THIS PROGRAM CALCULATES 2 + 2
```

Order of execution

HEKTOR will execute the statements in a BASIC program in numerical order of line number unless some statement tells it to do otherwise. Statements which do so are described in Subsection 7.3.6 'Flow-of-control statements'. Lines do *not* need to be typed in numerical order.

STOP

This instruction stops the execution of the program and returns control to the BASIC command mode. It can be inserted at the end of a program, or at any point within a program, if appropriate. STOP is optional at the end of a program. The program will stop anyway if it has no more instructions to execute.

7.3.2 A line of BASIC

Lines, statements and multiple statements

A 'line' of BASIC, starting with a line number, may be up to 253 characters long. (It will then extend over several screen lines.) It may include several BASIC statements, each separated by a colon, ':'.

Example:

```
20 X = 2 + 2: PRINT X
30 C = A + B: REM FORM THE SUM
```

In general, as many statements as will fit may be combined in one line, including REM statements. However statements containing the instructions GOTO, STOP, RETURN and IF (to be described later) must be last on a line.

Spaces within lines

Spaces may be inserted freely to improve the readability of a statement. They are ignored by HEKTOR. (The obvious exception is within a BASIC keyword: PR INT will be misinterpreted, and GO TO will not be accepted as GOTO.) Thus line 20 above could be written as:

```
20X=2+2:PRINTX
```

It is considered good programming practice to use spaces to keep a program easy to read, unless there are severe space limitations.

Syntax of BASIC

Example:

```
10 REM SQUARES OF NUMBERS
20 X = 2
30 Y = X*X
40 PRINT X, Y
50 X = X + 1
60 GOTO 30
```

This program will print a list of numbers and their squares until it is stopped by pressing the BREAK key. It illustrates many of the features of the syntax of the BASIC language, which will be described in the next few sections. Statements contain *variables*, such as X and Y, which can take on many different values. There are also *constants*, such as 1, 2, 30. Variables and constants are combined in *expressions*, such as X*X and X + 1. They are combined using *operators*, such as * (multiply) and +.

Statements also contain *keywords*, such as PRINT and GOTO. (The statement PRINT X, Y causes the current values of X and Y to be printed. The statement GOTO 30 causes line 30 to be the next line executed.)

7.3.3 BASIC variables

There are four types of variable in HEKTOR BASIC, *numeric* variables, *arrays*, *string* variables and *system variables*.

Numeric variables

A numeric variable takes on integer values in the range −32767 to +32767. It is denoted by one of the 26 letters A to Z.

Examples:

A, C, X are valid numeric variables

AB, C2, N.I are invalid

Arrays

An array is a list of numeric variables referred to by a single name. Each of the separate elements of an array is identified by using an index. For example, A(4) refers to the fourth element in the array A(*index*). There are 26 allowable arrays, whose elements are denoted by symbols A(*index*) to Z(*index*). The index can be a numeric variable, an integer constant, or any numeric expression.

The index must always be in the range 1 to 127.

Note that the numeric variable A and the array A (*index*) can both be used without confusion in the same program.

Examples:

C(X), F(6), and G(I + 3) are valid uses of an array variable.

C(200) is not valid because the index is too large.

DIM *dimension statements*

Before using an array variable, HEKTOR must be told how much storage to set aside for it (i.e., what is the maximum size you will want for the index?). This is done using a DIM (for dimension) statement. DIM statements must *not* be executed twice.

Example:

```
DIM A(8)
```

declares that the array A(*index*) contains 8 elements. Thus its index can range from 1 to 8. The number used in the DIM statement can be any integer between 1 and 127, or any numeric expression with a value in that range.

Examples:

```
DIM C(120);  }
DIM B(25*4)  } are valid
```

DIM T(X + 10) is valid if X + 10 is between 1 and 127.

String variables

A string variable takes on values which are letters, numbers, or other characters, stored as their character codes. It may be up to 254 characters in length. Thus abcd, TM221, $4.83 are all valid values for string variables. A string variable is denoted by one of the 26 symbols A$ to Z$. A string constant is denoted by enclosing it within either single quotes ' ', or double quotes " ". (Mixed quotes, " ', or ' " are not valid.)

Examples:

A$, C$, X$ are valid string variables

BC$, DE$ are invalid

"HOW ARE YOU", "Is it 2:30" are valid string constants

"ERROR' is invalid (mixed quotes)

Assignments: LET ... =

The value of a variable is set or changed using an assignment statement of the form:

LET *variable* = *expression*

Example:

```
LET X = 5
```

or simply

```
X = 5
```

The keyword 'LET' is optional. Its use helps to clarify the meaning of the statement, which can be interpreted as 'let the value 5 be assigned to the variable X'. The use of the '=' should be distinguished from its meaning in algebra, where it would be 'is equivalent to'. Thus the statement:

```
X = X + 1
```

in BASIC means 'let X take on a value one greater than its previous value'. In algebra, such a statement would be self-contradictory.

Examples:

```
integer variables:  G = X
                    Y = 3 + Z
array variables:    H(3) = 43 + X
                    LET K(2*Y) = L(3*Y)
string variables:   T$ = 'THIS IS THE END'
                    LET V$ = "Did you say /@£%&?"
```

Initialization

When you start the execution of a program using the RUN command, the value of all numeric variables is set to zero and the value of all string variables is set to the 'null string', i.e. no characters. RUN also deletes any arrays set up in previously run programs, but does *not* initialize arrays in the current program. That is because the BASIC interpreter does not know about them until it reaches the statement in which they are dimensioned. Unless you explicitly give array elements a value using assignment statements, they will have values which are unpredictable.

System variables

The three sets of variables defined above are general-purpose variables which you can use within programs as you wish. In addition to them, there are a number of special-purpose variables defined by the system, which you can use.

Some of the system variables are 'read-only' variables. In effect they are 'windows' which allow you to see the value of some parameter in HEKTOR. You can use these variables in expressions or PRINT statements, but you cannot change their value using assignment statements.

The remaining system variables are 'read/write' variables. You can change their values using assignment statements, although in some cases there is only a limited set of allowable values. All the read/write system variables have 'default' values which are set by the system when you enter the BASIC interpreter. All but TIME and BAUD are reset to those default values by the RUN command.

Read-only system variables

COL — Its value is the current column of the display.

KSTAT — Keyboard status. Its value is 0 if no key is being pressed, and the ASCII value of the key if one is being pressed.

ROW — Its value is the current row of the display.

SIZE — Its value is the number of bytes of available memory.

STAT — Its value represents the state of the peripheral board pushbuttons, as follows:
Neither button pressed, STAT = 0;
PD button pressed, STAT = 1;
PL button pressed, STAT = 2;
Both buttons pressed, STAT = 3.
Reading STAT also automatically resets the latch of the PL button.

SW — Its value is the numeric equivalent of the peripheral board switches. If the switches are taken to represent the eight bits of a binary number, the value of SW is the denary representation of that number.
Example:
If the left-most two switches only are set to 1, the value of SW will be 128 + 64 = 192.

Read/write system variables

BAUD — The value of this variable determines both the bit rate and the parity of data sent to the serial line. (This is needed if, for example, a printer is connected to HEKTOR.) The default value of BAUD is 0, which gives a bit rate of 1200 baud and parity = space. For all other values, use Table 7.1.

Table 7.1
Values of the system variable BAUD

Bit rate	*Parity*			
	Space	*Mark*	*Even*	*Odd*
1200	0	128	192	224
300	256	384	448	480
110	512	640	704	736

Values other than those in this table will produce unpredictable results! The value of this variable is not reset by the RUN command.

Example:

```
BAUD = 256
```

This sets the bit rate to 300 and the parity to 'space'.

LED — Its value is the numerical equivalent (denary) of the peripheral board LEDs, with each LED taken to be one bit of an eight-bit binary number.

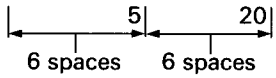
Example:

The statement

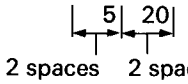
LED = 192

will light up the left-most two LEDs (192 = 128 + 64)

PW — Print width. This variable sets the number of columns allocated to each number printed by a PRINT statement. The default value is 6. With the default value, the statement PRINT 5,20 would produce the output:



With PW = 2, the same statement would produce the output:



PW only affects the column width allocated to numbers, whether output directly as in the example, or as a numeric variable or array. For example, it determines the number of columns allocated to X and G(I) in PRINT X, G(I). It has no effect on the format of string variables or string constants. PW should be in the range 0 to 63.

RANDOM — The value of this variable is used as a starting point by the function RND which generates pseudo-random numbers. For details of its use see the description of RND in Subsection 7.3.5.

SCROLL — This variable determines the screen format. For SCROLL = 0 the screen 'scrolls', i.e. when the screen is full, a new line is added at the bottom and all other lines move up one. With SCROLL = 1 (or any other non-zero number), when the screen is full the cursor moves back to the top, and all lines remain where they were until they are overwritten. The default value is 0.

TIME — There is a 'clock' in HEKTOR which is incremented every 100th of a second. The value of TIME is the current value of the clock. The clock is reset to 0 when you enter the BASIC interpreter and runs continuously thereafter, *except while a cassette tape is being read.* When it reaches its maximum value of 32767, it jumps to its minimum of −32767 and continues counting. It can be set to any desired value within its range using an assignment statement (e.g., TIME = 100) and will continue counting from that value. It is not reset by RUN commands.

7.3.4 BASIC operators and expressions

BASIC variables and constants can be combined to form expressions using operators.

Numeric operators and expressions

There are four numeric operators in HEKTOR BASIC

+	add	−	subtract
*	multiply	/	divide

These operators all produce results which are 16-bit integers (−32767 to +32767). The results of division are rounded down to the nearest integer.

Examples:

100/8 = 12, 3/2 = 1, 4/8 = 0

Within an expression, * and / are evaluated before + and −, unless modified by parentheses. Otherwise, expressions are evaluated from left to right.

Examples:

2+3−4
5 −4
1

Both operators have the same priority, so the expression is evaluated from left to right.

2+3*2
2+ 6
8

The multiplication is evaluated before the addition

(2+3)*2
5 *2
10

The contents of the parentheses are evaluated first, changing the order of priority of the operators.

Examples:

X + 3; 4*Y−H; N(6)*N(H)+X−2; (A−4)/(B*6)

Relational operators

Relational operators are used to compare the magnitudes of two arithmetic variables, constants or expressions.

>	greater than	<	less than
>=	greater than or equal to	<=	less than or equal to
=	equal to	<>	not equal to

Examples:

X>3

Y<=(X+4)/A(3)

Relational operators can only be used in IF...THEN instructions (Subsection 7.3.6). An expression such as X>Y can take only two values, 'true' and 'false'.

String operators and expressions

String expressions can be formed from string variables or constants using the '&' symbol for concatenation. (The symbol & is called the ampersand.)

Example:

A$ = "HELLO" & "THERE"

will produce the same value for A$ as

A$ = "HELLOTHERE"

7.3.5 BASIC functions

A BASIC function is an expression of the form KEYWORD (*argument(s)*), where the KEYWORD is one of the set described below. Each argument is either a numeric type (X,Y,Z) or a string type (X$). It can be either a variable, a constant or an expression. A function is a built-in routine which acts on its argument(s) and returns either a number or a string. If it returns a string it is called a *string function*, and if it returns a number it is called a *numeric function*. There are, however, some numeric functions whose arguments are strings and vice-versa.

Numeric functions

ABS(X) Returns the absolute value (or modulus) of X.
Example: ABS(−5) is 5, ABS(10) is 10

RND(X) Returns a pseudo-random integer between 1 and X inclusive. Thus RND(20) will give a random integer between 1 and 20. X must be greater than or equal to 1. When used repeatedly within the same program, RND will return different numbers. The function RND uses the system variable RANDOM as a 'seed' or starting point. When RND is used it changes the value of RANDOM, so that when it is next used, RND will produce a different result. However, the command RUN resets RANDOM to its default value of zero so, if the program is run a second time, the same sequence of numbers will be generated. To produce different sequences of numbers from RND when a program is re-run, you must change the starting value of RANDOM in the program.

ASC(X$) Returns the ASCII character code for the first character of X$.

Examples: Y = ASC("A") the value of Y is 65, the ASCII code for A.
Y = ASC("HELLO") the value of Y is 72, the ASCII code for H.

VAL(X$) This function is used to transform a string expression containing a series of digits into a number.

Example: LET X$ = "−32"
LET Y = VAL(X$) then Y will have the integer value −32.

LEN(X$) Returns the number of characters in the string X$.

Example: PRINT LEN("THIS IS A TEST") will print 14.

String functions

CHR$(X) Returns the character whose ASCII code is X. This function is the inverse of ASC. X should be between 1 and 255.

Example: PRINT CHR$(80) will print P, the letter whose ASCII code is 80.

LEFT$(X$,Y) Returns the left-most Y characters of X$

Examples: X$ = 'HELLO'
LEFT$(X$,3) is HEL
LEFT$(X$,1) is H
LEFT$(X$,4) is HELL

RIGHT$(X$,Y) Returns the right-most Y characters of X$.

Examples: X$ = "HELLO"
RIGHT$(X$,4) is ELLO
RIGHT$(X$,1) is O
RIGHT$(X$,2) is LO

MID$(X$,Y,Z) Returns Z characters from X$, starting with the Yth character.

Examples: X$ is "HELLO"
MID$(X$,3,2) is LL
MID$(X$,1,5) is HELLO
MID$(X$,4,1) is L

STR$(X) This function is used to transform a number stored as an integer to one stored as a string.

Example:
STR$(100) has the value "100". If A = 300 and X$ = "THERE ARE" & STR$(A) & "PEOPLE", then the value of X$ = "THERE ARE 300 PEOPLE".

7.3.6 Flow-of-control statements

Lines in a BASIC program are executed in sequential order of line number unless an instruction appears altering this order. There are four such instructions in HEKTOR BASIC. They correspond to the four cases:

(a) a simple transfer from one part of a program to another (GOTO)

(b) a transfer to another part of a program followed by a return to the point of transfer (GOSUB...RETURN)

(c) repeated execution of a sequence of statements (FOR...NEXT)

(d) a statement to be executed, if some condition is true (IF...THEN).

GOTO n

This statement causes execution to jump to line n. A GOTO n statement must be the last one on a line. n may be a positive number or a numeric expression with values in the range 1 to 32767, and must be the number of a line in the program.

Examples: GOTO 20
GOTO A*6+B

GOSUB n

This statement is used in conjunction with RETURN to transfer to and from subroutines. GOSUB is similar to GOTO except that the current line number is stored. GOSUBs may be 'nested', i.e. further GOSUBs may be used before the RETURN which terminates a previous GOSUB.

Examples: GOSUB 200
GOSUB A*6+B

RETURN

A RETURN statement causes execution to jump to a statement following the corresponding GOSUB statement. For nested subroutines, RETURN transfers control to the statement after the most recently executed GOSUB to which control has not yet been returned.

Example:

```
10 REM PRINT X TO THE N, FOR N = 2, 3 OR 4
20 INPUT X, N
30 GOSUB N*100
40 PRINT Y
50 STOP
200 Y = X*X : RETURN
300 Y = X*X*X : RETURN
400 Y = X*X*X*X : RETURN
```

Note the use of the STOP statement to prevent the program from continuing into the subroutines.

FOR...NEXT

The FOR and NEXT instructions are used to set up a *'loop'* – a section of program which is executed repeatedly.

The general form is
FOR *assignment* TO *num.expr.* STEP *num.expr.*

:

NEXT *variable*

Example:

```
10 REM PRINT NUMBERS FROM 1 TO 10 AND
                              THEIR SQUARES
20 FOR X = 1 TO 10
30 PRINT X, X*X
40 NEXT X
```

In this example, statement 30 is executed repeatedly: once with X = 1, once with X = 2, etc. ending with X = 10. The general form of a FOR...NEXT loop is:

```
FOR X = A TO B STEP C
        :
Program lines to be
executed repeatedly
        :
NEXT X
```

The loop is executed first with X = A. When the 'NEXT X' statement is reached, X is incremented by C. It is then tested to see whether or not it is still within the range A to B (inclusive). If so, the loop is executed again. If not, the statement following NEXT X is executed.

Any numeric variable can be used in place of X. Any numeric variable, constant or expression can be used in place of A, B and C. The values of B and C are stored when the FOR statement is executed, so that they can be tested later. Thus changing their values during execution of the loop will not affect the loop operation. C can be positive or negative. (C = 0 will produce a never-ending loop.) 'STEP C' may be omitted if the desired step is +1.

The statements within a FOR...NEXT loop may contain another FOR...NEXT loop, which may, in turn, contain another FOR...NEXT loop, etc. Such 'nested' loops should use different index variables as otherwise the NEXT statement might terminate the wrong loop. When a NEXT X statement is executed, the name of the variable is checked with that of the most recent FOR instruction. If they do not agree, that FOR is terminated and the next most recent FOR is tested. If they do agree, the variable X is incremented and tested as described above.

IF *condition* THEN *statement*

The statement following THEN is executed only after evaluating the condition and finding it to be TRUE.

Example:

```
IF X>Y THEN PRINT X
```

X will be printed only if it is greater than Y. If not, the program will simply go on to the next line. Only one statement is allowed following THEN, and moreover, the IF statement must be the last in a line. If you wish to execute more than one statement using an IF statement, use a GOSUB or use a GOTO, transferring control to the desired statements. The condition tested by an IF statement is normally a comparison of two numeric variables, constants or expressions using the relational operators (see Subsection 7.3.4).

Examples:

```
IF X + Y> = 0 THEN GOSUB 20
IF X = 2+3+Z THEN Y = 20
```

However, it is also possible to compare strings using relational operators. Their ASCII code values will be compared, character-by-character.

Example:

```
IF X$ = "END" THEN GOTO 100
```

7.3.7 Input/output instructions

Within a HEKTOR BASIC program, the 'standard' devices for input and output are the keyboard and the television screen, respectively. The principal instructions for handling them are PRINT and INPUT. There are also other, more general-purpose, instructions, which allow a variety of devices to be used for input and output.

PRINT

The general form of a PRINT statement is:
PRINT *expression 1, expression 2, etc.*

Examples:

```
PRINT X
PRINT X, Y, C$, "THE MOON IS BLUE"
PRINT H$ & CHR$ (Y) & T$, X + (Y – 3) /6
```

Each expression will be evaluated and output to the screen. Numeric results will be printed at the right of a space whose width is the current value of the system variable PW (print width). The default value of PW is 6. If PW is smaller than required for a particular number then extra space will be allowed as required for that number only. Subsequent numbers will use the width set by PW.

Successive PRINT statements in a program can be made to produce output either on the same line on the screen or on a new line. If a PRINT statement ends in a comma the next PRINT statement executed will continue on the same line, as though both were part of a single statement. A PRINT statement which ends without a comma will move the cursor to the beginning of a new line after printing its last expression.

Examples:

```
10 FOR X = 1 TO 4
20 PRINT X, X*X
30 NEXT X
```

This program will print the numbers from 1 to 4 and their squares, with each pair printed on a separate line.

```
10 FOR X = 1 TO 4
20 PRINT X, X*X,
30 NEXT X
```

This time the pairs of numbers will be printed on the same line.

The statement

PRINT

will produce a blank line (unless a previous PRINT statement has left the cursor in the middle of a line, in which case that line will be terminated).

TAB (I)

An expression of the form TAB(I) can be inserted in a PRINT statement at any point. It will move the screen cursor to the Ith column, where I = 1 to 64. The argument I can be an integer constant, variable or expression. TAB cannot be used to make the cursor go backwards. If the value of I is less than the current column, the cursor will move to the Ith column on the next line.

Example:

```
10 FOR I = 1 TO 25
20 PRINT TAB(I), "HELLO"
30 NEXT I
```

This program prints a diagonal line of "HELLO"s across the screen.

INPUT

The general form of an INPUT statement is:

INPUT *variable 1, variable 2, etc.*

When HEKTOR is expecting an input, the value of the first variable must be typed in followed by RETURN, followed by the value of the second variable and RETURN, etc. When all values have been typed in HEKTOR will continue executing the program. Since there is no indication that HEKTOR is expecting input, it is helpful to precede INPUT statements with PRINT statements requesting the required input.

The variables may be numeric or string variables, or array elements. The maximum size of string variables which can be input from the screen is 77 characters.

Example:

```
10 PRINT "TYPE VALUES FOR X, Y, Z$"
20 INPUT X, Y, Z$
RUN
TYPE VALUES FOR X, Y, Z$
```

(you then type)

```
30 RETURN
40 RETURN
HELLO HEKTOR! RETURN
```

The stored values of X, Y and Z$ will be, respectively 30, 40 and "HELLO HEKTOR!".

SOUND (X, Y)

This instruction, when executed, produces a sound from HEKTOR's sound generator which is played through the TV loudspeaker. The length of the sound is set by Y, in multiples of 100th of a second. The pitch of the sound is set by X. The period of the pitch is X times 250 microseconds.

Example:

SOUND (4, 100) produces a sound of period 4 × 250 microseconds, or 1/1000th of a second (i.e. a frequency of 1000 Hz). The sound lasts for 100 × 1/100 = 1 second.

X and Y can be numeric variables, constants or expressions, but must be in the range 1 to 255.

General purpose input and output instructions

The PRINT and INPUT statements are used for output to and input from the TV screen and the keyboard respectively. However, within a program, data can also be sent to and from a cassette recorder, or a printer or other device connected to the serial line. To do so there are more general input/output instructions, PRINT # n, and INPUT # n. Whereas HEKTOR is always ready to send data to the TV screen or receive it from the keyboard, this is not so for the other devices. Before they are used either for input or output, a channel must be opened with the OPEN instruction. If a channel is subsequently to be used for another purpose it must first be closed using the CLOSE instruction and then reopened.

OPEN # n;

This command opens a channel. You must (a) choose a channel number, n, between 1 and 9, (b) specify the device the channel leads to, (c) specify whether you want input or output, (d) for cassette tape, specify a file name to be used as a label.

Example:

The statement OPEN # 3; CR, I, 'TEST DATA'

opens channel number 3 on to the cassette recorder (CR), for input (I), in a file labelled 'TEST DATA'. The general form of the OPEN # n statement is:

OPEN # n; *device, access, name*

where n can be any number between 1 and 9 (or a numerical expression with a value between 1 and 9), *device* must be either CR for cassette recorder or SL for serial line, *access* must be either I for input or O for output, and *name* is a string of characters up to 200 long (or a string variable or expression). Note that # n is followed by a semi-colon.

Within a program, for a given channel n, OPEN # n should be used before using either the INPUT # n or PRINT # n statements. If the channel specified is a tape recorder, when OPEN # n is executed HEKTOR will display either the message SET RECORD (for output) or SET TO PLAY (for input) on the screen. You should then operate the controls of your cassette recorder accordingly, and press RETURN to signal that you have done so. For input, HEKTOR will search the tape until it has found the file labelled with the correct name and then stop the tape until it reaches the first INPUT # n statement.

CLOSE # n;

This command closes a previously opened channel to make it available again for another purpose. n must be a number, variable or expression with a value between 1 and 9.

PRINT # n;

The general form of a PRINT # n statement is:

PRINT # n; *expression 1, expression 2, etc.*

This statement is used in exactly the same way as the PRINT statement. When a PRINT statement is executed, each expression is evaluated and the resulting data is sent to channel n, which must previously have been opened for output using the OPEN command. The total data from a PRINT # n statement forms a single 'record'. n can be a number, variable or expression with a value between 0 and 9 inclusive. Channel 0 is the TV screen and is always open. Thus PRINT # 0 (and also PRINT # n where channel n has not been opened) is equivalent to PRINT.

If the specified channel is the cassette recorder, each expression will be stored as a string with its length as a prefix.

For tape, the total maximum length per print statement should be 253 characters. This is not checked, so if the maximum is exceeded it will not be possible to retrieve some of the characters with an INPUT statement.

Example:

```
10 REM CALCULATES THE SQUARES OF
20 REM 100 NUMBERS AND STORES ON TAPE
30 OPEN # 5; CR, O, 'SQUARES'
40 FOR I = 1 to 100
50 PRINT # 5; I*I
60 NEXT I
```

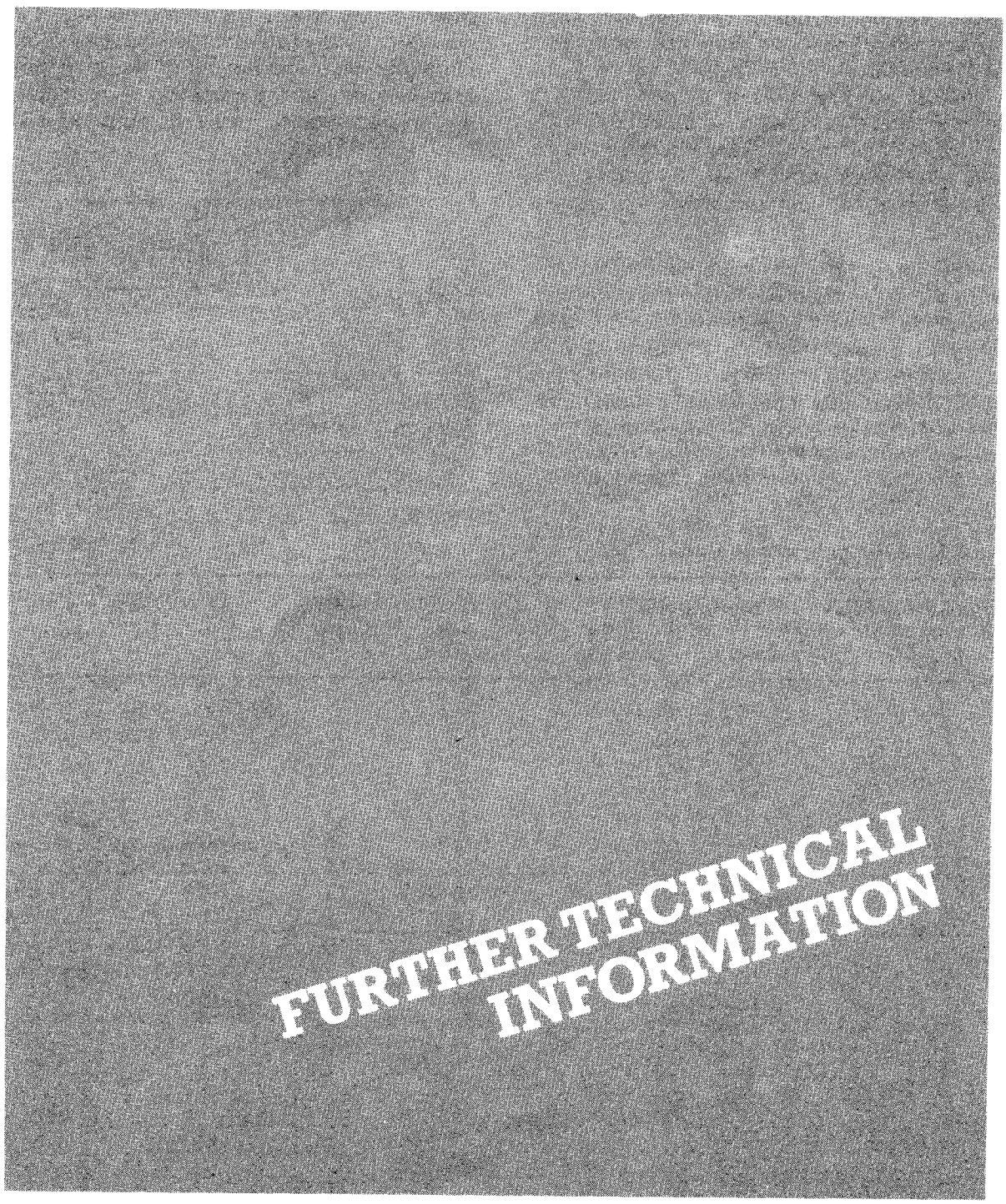
INPUT # n;

The general form of an INPUT # n statement is:

INPUT # n; *variable 1, variable 2, etc.*

This statement is used in the same way as the INPUT statement, except that input is expected from channel n (which must previously have been opened with an OPEN command). Channel numbers can be any number between 0 and 9. Channel 0 is the keyboard and is always open. Thus INPUT # 0 (or INPUT # n where channel n has not been opened) is equivalent to INPUT. When input is taken from the cassette recorder, the whole of the next record is read (the entire output of a PRINT statement).

P A R T III



8 HEKTOR SYSTEM HARDWARE

This section contains a technical description of the HEKTOR system hardware. It includes a description of the circuits and operation of each of the microcomputer subsystems, and information to enable expansion of HEKTOR.

8.1 The microcomputer bus

Table 8.1 lists the full set of microcomputer bus signals available on the HEKTOR board edge connector. The corresponding pin on the microprocessor is also listed, where applicable. Note that the address bus is available in two forms. (AD0–7, A8–15) is the conventional 8085 address bus, with (AD0–7) being the multiplexed address/data bus lines. (A0–7, A8–15) is the full demultiplexed 16-bit address bus, where (A0–A7) are the address bus lines supplied by the memory subsystem's address latch.

The edge connector signals are unbuffered, but there are pull-up and pull-down resistors connected to some of the bus lines, as shown in Table 8.1. Note that the microprocessor signals (X1, X2, SOD, SID, TRAP, RST6.5, RST5.5, and $\overline{\text{RESET IN}}$) are used by the HEKTOR hardware, but are not available externally. The status signals (S0, S1) are unused.

8.2 The microprocessor

The microprocessor is an 8085A-type device, connected to the microcomputer bus as shown in Table 8.1. There is a 6.048 MHz crystal connected to the X1 and X2 signal pins on the microprocessor. This defines the system clock frequency as 3.024 MHz, and a signal of this frequency is output from the microprocessor on the CLK bus line.

The $\overline{\text{RESET IN}}$ pin of the microprocessor is connected to the RESET key and power-on reset circuit shown in Figure 8.1. The combination of R and C provides a delay of at least 100 milliseconds following switch-on before the $\overline{\text{RESET IN}}$ signal rises sufficiently to enable the execution of instructions by the microprocessor (starting at address 0000 (hex)).

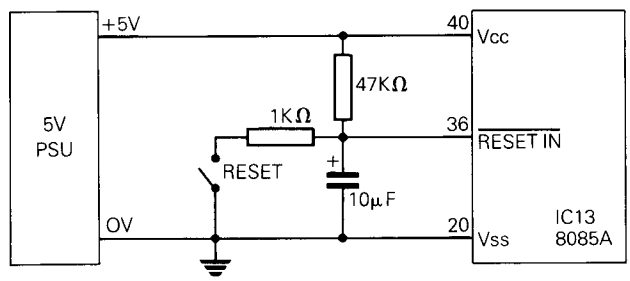


Figure 8.1 Reset circuit

Table 8.1 Edge connector signals

Component side pin	*8085 pin*	*Signal name*	*Wiring side pin*	*8085 pin*	*Signal name*
1		10V unreg. DC power	2		10V unreg. DC power
3		power ground	4	20	Signal ground
5	32	$\overline{\text{RD}}$ (47kΩ pull-up)	6	19	AD7
7	31	$\overline{\text{WR}}$ (47kΩ pull-up)	8	18	AD6
9	30	ALE	10	17	AD5
11	11	$\overline{\text{INTR}}$	12	16	AD4
13	3	RESET OUT	14	15	AD3
15	37	CLK	16	14	AD2
17	28	A15	18	13	AD1
19	39	HOLD (3.9kΩ pull-down)	20	10	INTR (3.9kΩ pull-down)
21	38	HLDA	22	7	RST7.5 (47kΩ pull-down)
23	35	READY (3.9kΩ pull-up)	24	12	AD0
25	34	IO/$\overline{\text{M}}$	26	2	A0
27	27	A14	28	5	A1
29	26	A13	30	6	A2
31	25	A12	32	9	A3
33	24	A11	34	12	A4
35	23	A10	36	15	A5
37	22	A9	38	16	A6
39	21	A8	40	19	A7

(pins on IC 12 (74LS373))

8.3 The memory subsystem

The on-board memory of HEKTOR has been designed to offer considerable flexibility in the amounts of ROM and RAM that can be fitted. The flexibility is achieved by the use of an FPROM for address decoding, and by the use of 24-pin sockets which will accept a variety of ROM and RAM devices.

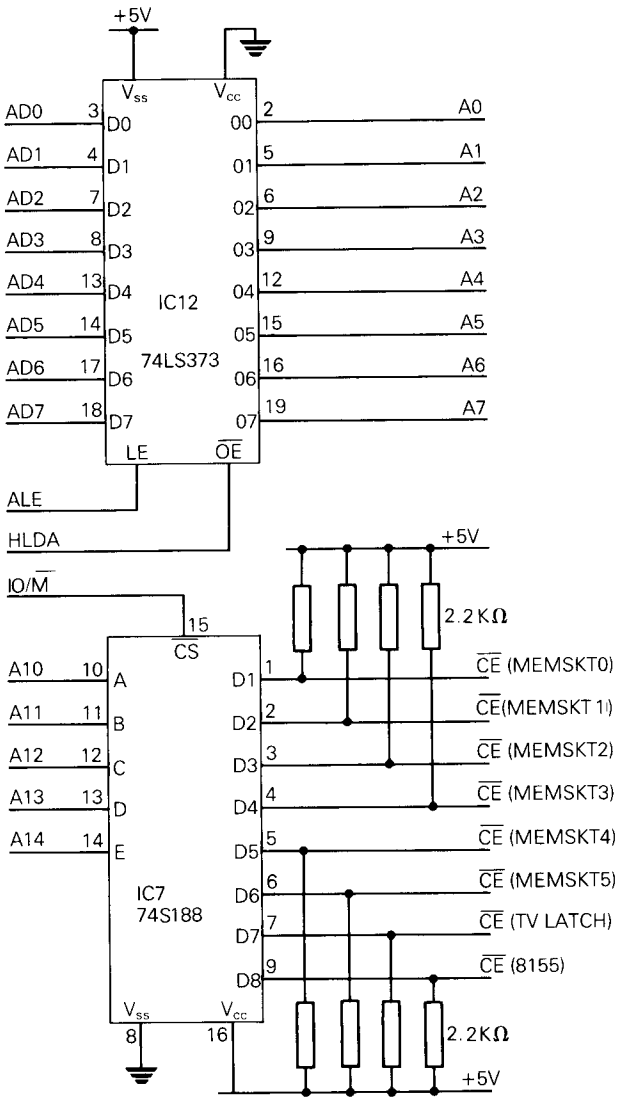


Figure 8.2 Address decoding circuitry

The address-decoding circuitry is shown in Figure 8.2. IC12 latches the contents of the AD bus lines when the microprocessor supplies an ALE pulse. At this time, the AD lines are carrying the lower 8 bits of an address, and so the A0–7 outputs of the latch retain this address information throughout the memory access cycle. By connecting HLDA to the output enable of the latch, these address outputs can be disabled if, for example, an external device wishes to gain control of the bus in order to directly access memory. IC7 is a 32 × 8 bit FPROM. It is enabled ($\overline{CS}$) at the beginning of every memory access cycle and decodes the higher-order address (A10–14) to provide eight chip enable signals within fifty nanoseconds. The chip enables that it generates select one of the six memory devices (mounted in MEMSKT 0–5), the TV latch circuitry or the 8155 RAM/IO chip. The FPROM decoding map used in HEKTOR is shown in Table 8.2.

Each of the six memory devices is mounted in a 24-pin MEMSKT. For all but two pins, the connections required are the same for several types of ROM/RAM device. The address lines (A0–A7) are the outputs of the address demultiplexing latch (Figure 8.2) and together with the address bus lines A8 and A9, define 1K addresses within the device. The eight data lines are connected directly to the AD bus lines so that addressed data is output to the bus when $\overline{RD}$ is asserted by the microprocessor. The $\overline{CE}$ signal for the device is one of the chip enables output by the memory decoder (Figure 8.2).

The two remaining signal pins, numbers 19 and 21, have a function which depends on the device actually mounted in the socket, and there are link holes on the HEKTOR pcb so that any of (+5 V, $\overline{WR}$, or

Table 8.2 Address decoding

E D C B A	*Device enabled*	*Effective address for device (hex)*
0 0 0 X X	MEMSKT 0	0000–0FFF, 8000–8FFF
0 0 1 X X	MEMSKT 1	1000–1FFF, 9000–9FFF
0 1 0 0 0	TV LATCH	2000–23FF, A000–A3FF
0 1 0 0 1	8155	2400–27FF, A400–A7FF
0 1 0 1 X	MEMSKT 5	2800–2FFF, A800–AFFF
0 1 1 0 X	MEMSKT 4	3000–37FF, B000–B7FF
0 1 1 1 X	MEMSKT 3	3800–3FFF, B800–BFFF
1 0 X X X	—	4000–5FFF, C000–DFFF
1 1 0 X X	—	6000–6FFF, E000–EFFF
1 1 1 X X	MEMSKT 2	7000–7FFF, F000–FFFF

A11) can be connected to pin 21, and either of (0V or A10) can be connected to pin 19. The table in Figure 8.3 shows which connections are required for four types of device, two being ROM/EPROM and two being RAM devices. The links shown are those for MEMSKT 0–2, where 4K masked ROMs or 2732-type EPROMS are fitted to HEKTOR as standard. For the 2K RAMs in MEMSKT 3–5, fitted as standard, the links are pin 21–$\overline{WR}$, pin 19–A10.

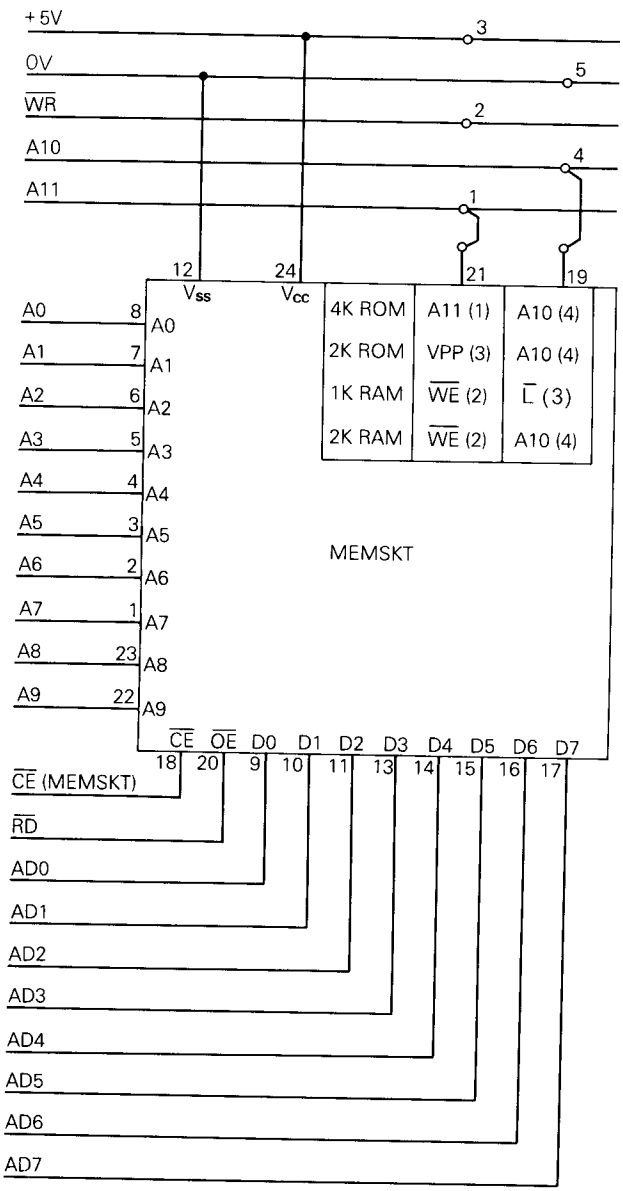


Figure 8.3 MEMSKT decoding

The third component of the memory subsystem is the 8155 RAM/IO device, connected as shown in Figure 8.4. This device is purpose-designed for use with 8085-based microcomputers. It has on-chip address demultiplexing, and so most address/data/control pins connect directly to their corresponding bus lines (i.e. AD0–7, RESET, ALE, $\overline{RD}$, $\overline{WR}$). With the chip enable supplied by address decoder (IC7), the 8155 appears as a 256-byte RAM device, responding to addresses in the range 2400 to 27FF (hex). As this range covers 1024 addresses, each

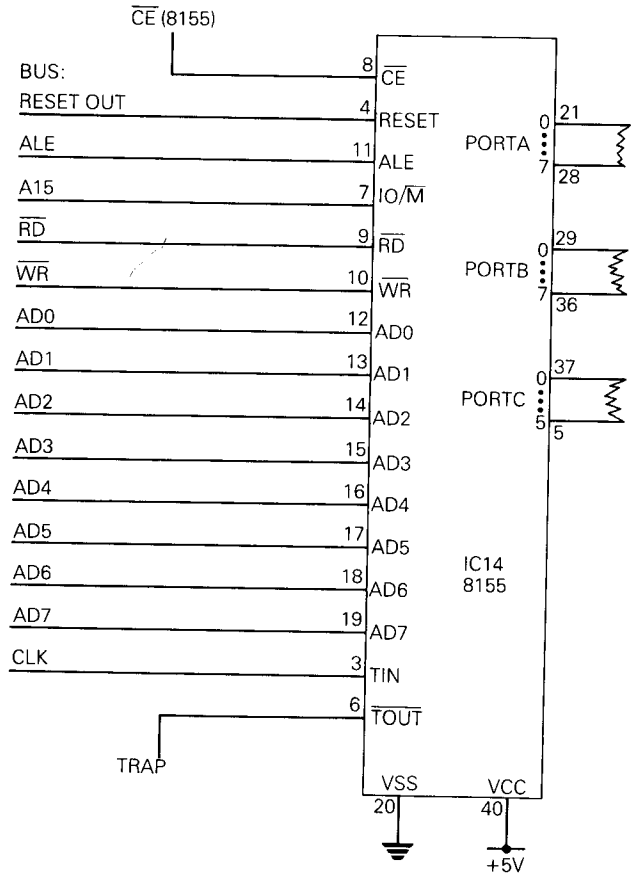


Figure 8.4 8155 device connections

RAM location XY responds to any of four addresses (24XY, 25XY, 26XY or 27XY). Only addresses of the form 27XY (hex) are used by the HEKTOR software.

The I/O addresses for this device are determined by the connection of the IO/$\overline{M}$ pin on the 8155 device *not* to the IO/$\overline{M}$ bus line but to the A15 bus line. This enables the I/O registers in the device to be addressed as memory, rather than as I/O devices (for which only the IN and OUT machine instructions are operable). With the connection used, the addresses XY00–XY05 (hex) access the five I/O registers where XY is any of A4–A7 (hex). HEKTOR system software uses the addresses A400–A405 (hex), for the control register, port A, port B, port C, and the two timer registers, respectively. Note that the connection of the timer pins (TIN and $\overline{TOUT}$), to the bus CLK and microprocessor TRAP interrupt pin, means that the microcomputer has a built-in programmable real-time interrupt. This is used by the HEKTOR system software to provide the 'single-step' facility (see Sections 4 and 9).

8.4 The keyboard interface

The keyboard contains sixty keys, of which two (RESET and BREAK) are connected directly to individual microprocessor pins ($\overline{RESET\ IN}$ and RST6.5, respectively). The remaining keys are arranged as part of an 8 × 8 matrix as shown in Figure 8.5.

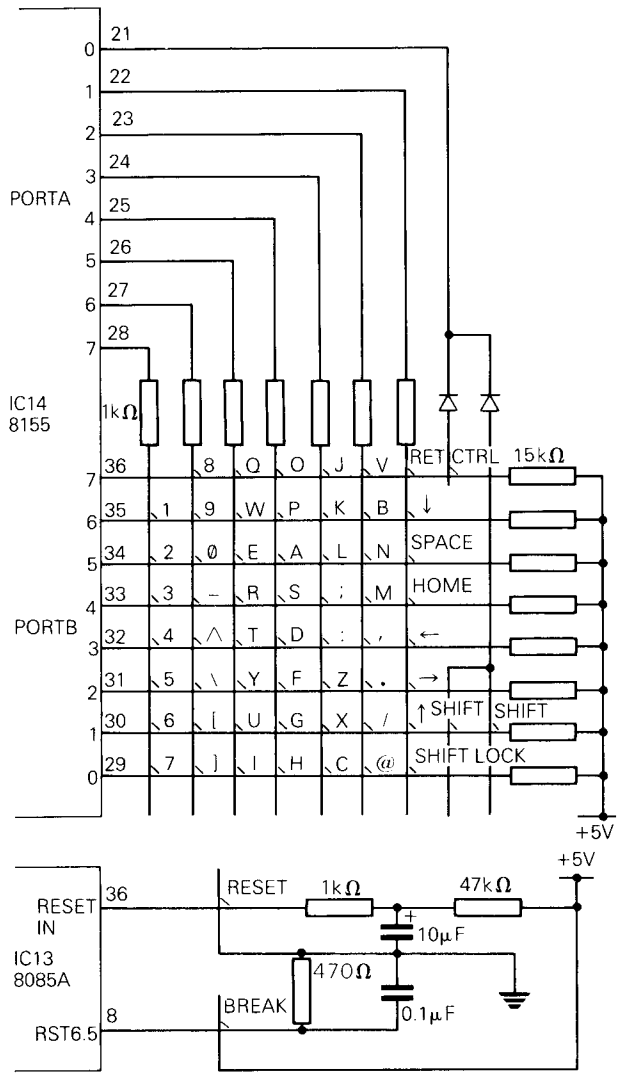


Figure 8.5 Keyboard interface

Reading the keyboard operates as follows. Port A of the 8155 device is programmed permanently as an 8-bit output port and Port B as an input port. (Port C is also an output port but is used to control the TV and cassette interfaces.) By outputting a bit pattern on Port A for which just one bit is 0, and then by reading the data at port B, any key depressed on the selected row of the matrix can be identified by the corresponding bit at port B being 0. The keyboard is therefore scanned row by row to detect and identify any key depression. Note that special precautions have to be taken for the CTRL and SHIFT keys. If these were connected on a row of the matrix using the resistive coupling method, then when CTRL, SHIFT and Q are pressed, this combination would be indistinguishable from the CTRL, SHIFT and U combination, for example. This is avoided by decoupling the CTRL and SHIFT keys with diodes, as shown.

8.5 The TV interface

The TV interface centres around the use of the SFF96364 controller device. A block diagram of the interface is shown in Figure 8.6. The screen display consists of sixteen lines of sixty-four characters, which are stored as ASCII codes in the 1K RAM. Each character is displayed as a 7 × 5 dot matrix with the CHAR ROM containing the patterns for each ASCII code.

The display is refreshed as follows. The OSCILLATOR supplies one pulse (F/8) for each character position on each line of the display. The controller addresses the RAM, and the ASCII code supplied is latched into the DISPLAY LATCH. The display pattern for that code is looked up in the CHAR ROM, using the ASCII code combined with a 3-bit address from the controller to access one (of the seven) rows of dot patterns for that character. The row pattern is loaded into the SHIFT REG., where it is serially output on the VIDEO line, using a clock frequency (F) which is eight times the character frequency F/8. The dot pattern stream is mixed with the line and frame SYNC signal to provide a composite video signal. This is available directly, for use with a video monitor, or modulated for use with a standard UHF TV receiver.

The SYNC signal is generated from the nominal 1 MHz Q1 input to the controller. In HEKTOR, the bus CLK signal of 3.024 MHz is divided by 3 to give a 1.008 MHz frequency. This leads to SYNC tuning which is well within the horizontal and vertical synchronization capabilities of most TV receivers. The $\overline{INI}$ output of the controller stops the oscillator and blanks the MIXER during line flyback. The cursor is displayed as a result of the PT output of the controller disabling the CHAR ROM, so as to produce a line of dots in the otherwise blank line underneath the character.

Writing a new character to the refresh RAM is a two-stage process involving the WRITE LATCH. The microcomputer stores the ASCII code for a character in the latch by writing to address 2000 (hex). (Figure 8.6 shows the relevant signals: the $\overline{CE}$ from the address decoder, and the $\overline{WR}$, AD0–7 bus signals). Having written the character, it sends a control code to the controller by setting bits 1–3 of port C to 111 (binary) and pulsing bit 0. (Bit 4 of port C is normally 1; it acts as an enable for the latch output). The character is then transferred to the RAM during the next line flyback period, when the W output of the controller enables the WRITE LATCH output and simultaneously write-enables the RAM (assuming bit 4 of port C is 1).

The special display functions performed by the controller are executed by the microcomputer writing a 'space' character to the WRITE LATCH, but with a different control code on bits 1–3 of port C. The code 001 (binary) causes a 'carriage return', 010 causes a 'line feed', 100 causes 'cursor left', 110 causes 'cursor up', and 000 causes the screen to be cleared and the cursor positioned in the top left-hand corner; the 'home' function.

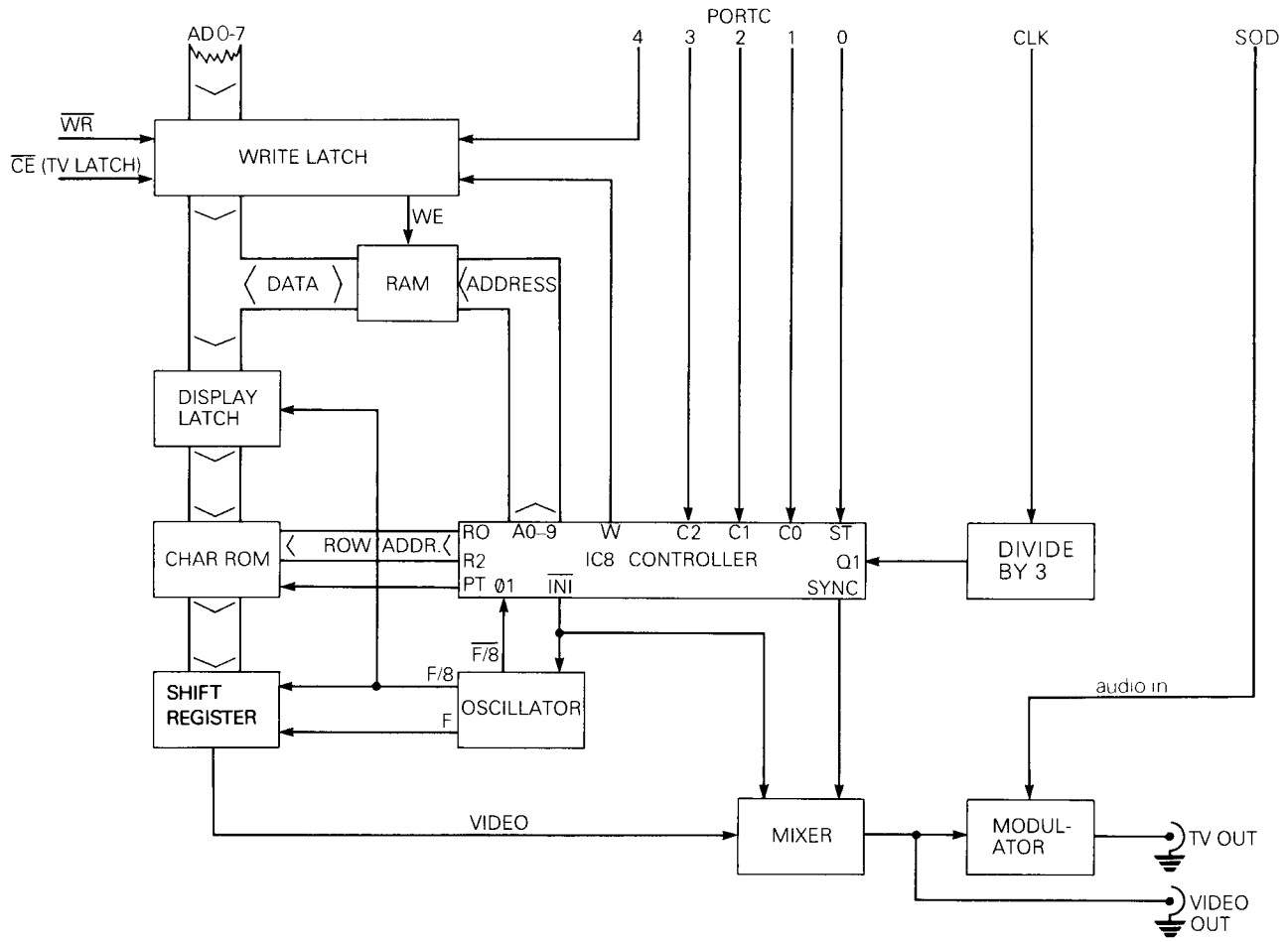


Figure 8.6 TV interface

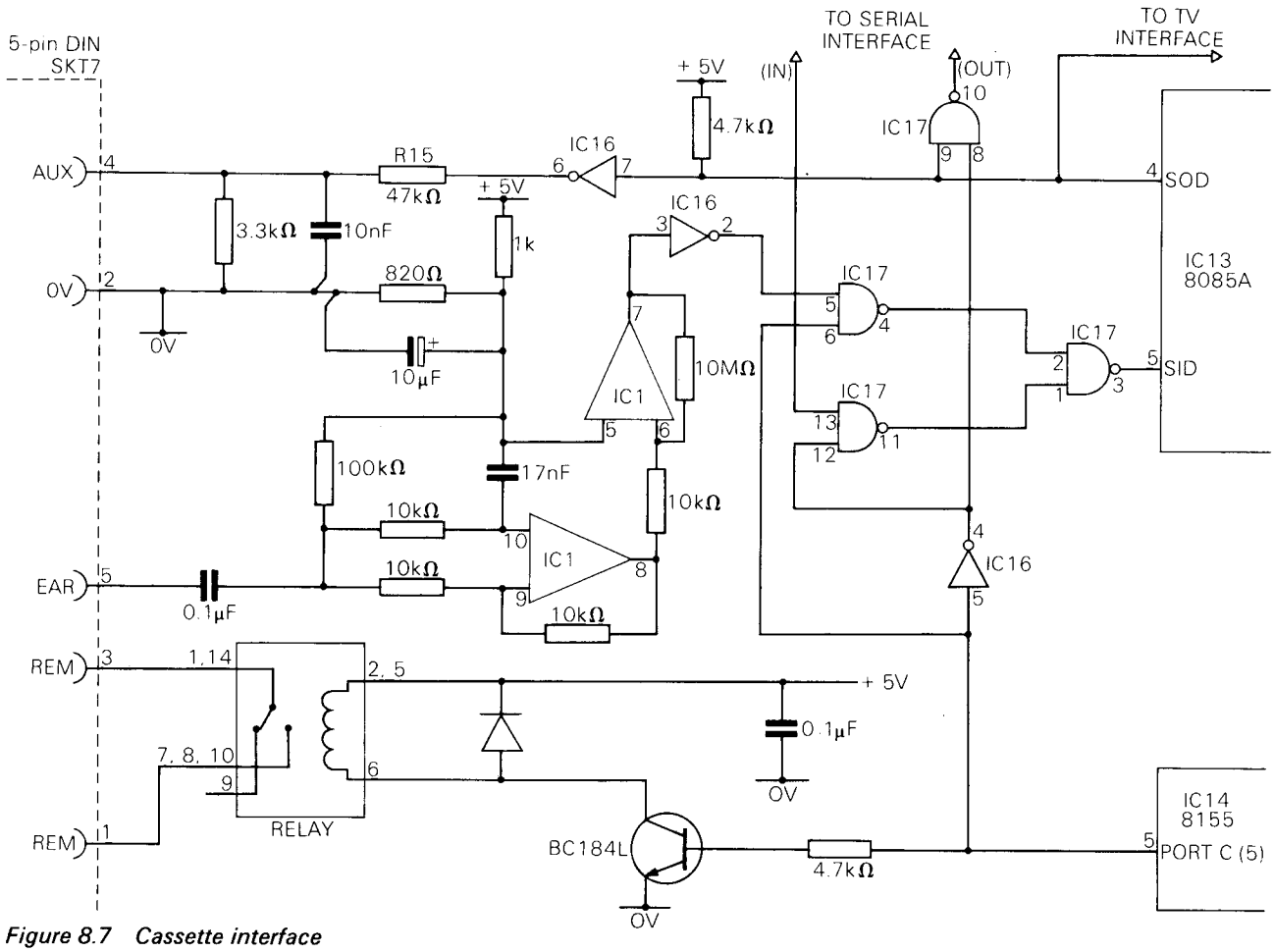


Figure 8.7 Cassette interface

The microcomputer is required to wait before attempting to write another character or control function. The hardware allows one character per line (64 microseconds), but the 'carriage return' takes longer (up to 4.2 milliseconds), and the 'home' up to 132 milliseconds, as the RAM memory has to be cleared. In fact, the HEKTOR software inserts much longer delays than are necessary, for readability reasons.

8.6 The cassette/serial line interface

The microprocessor's SID and SOD direct I/O lines are used for input and output in the cassette/serial line interface. A multiplexer is used to select either the cassette interface or the serial line interface at any instant.

The cassette interface circuit and multiplexer is shown in Figure 8.7. The multiplexer is controlled by the logic level on an output bit of the 8155 device (Port C, bit 5). When this is 1, the cassette interface is selected. That is, the cassette recorder motor is enabled (via the relay and REM connector), data reaching the SID pin on the 8085 is supplied by the cassette input interface circuit (rather than the serial interface), and data output from the SOD pin on the 8085 is inhibited from reaching the serial line interface.

The cassette output circuit (Figure 8.7) consists simply of an attenuator and filter. The signal on the SOD line is an oscillating logic level in which each full cycle at 2kHz represents a '1' data bit, and each half-cycle at 1kHz represents a '0' data bit. The conditioned signal is intended to supply the AUX connection on a cassette recorder.

The cassette input circuit accepts a signal from the EAR connection on a cassette recorder. This signal is first processed by an all-pass filter and the result is available on pin 8 of the IC1 (the output of one of its four operational amplifiers). The all-pass filter has the effect of reducing the phase distortion introduced by the record/playback process, thus restoring the position of the zero-crossings in the received signal to approximately their original position. By amplifying and clipping this signal (in a second operational amplifier and logic gate), the zero-crossing positions are converted to logic-level transitions and this logic signal is presented to the SID input of the 8085. The data bit values are then recovered by (software-based) measurement of the intervals between transitions of the SID signal.

The serial line interface (see Figure 8.8) strictly requires an external ±12 V power supply to meet the conditions of the RS232C serial line specification. For most types of serial equipment, however, the

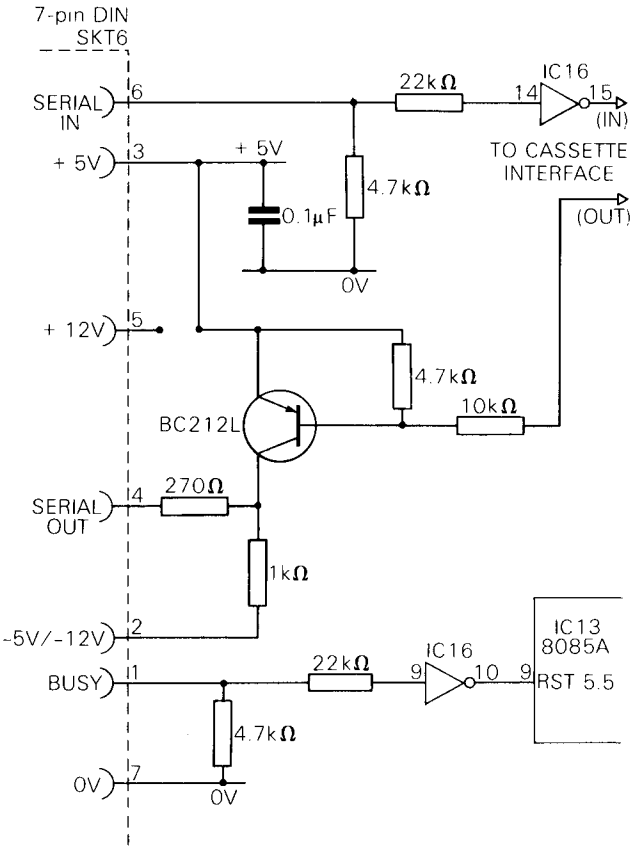


Figure 8.8 Serial line interface

addition of a single external supply of about −5 V on pin 2 is adequate (see Subsection 8.9).

The BUSY input signal, which will switch between positive or negative levels of several volts, is conditioned to supply a switched 0/5 V signal at the RST5.5 input pin of the microprocessor. The HEKTOR system software treats this input as a flag signal, rather than an interrupt. BUSY being low (or disconnected) inhibits the serial output software from sending the next output character. The SERIAL IN interface is identical, except that the SID microprocessor input is driven when the multiplexing circuit (Figure 8.7) is switched to the serial line mode.

The SERIAL OUT interface is driven by the microprocessor SOD line (again assuming the multiplexer is in serial line mode). Signal conditioning converts the switched 0/5 V SOD signal to a $\pm V$ signal, where V depends on the power-supply configuration. The HEKTOR board is linked so that the SERIAL OUT levels are +5 V and $-V$, where V is the external voltage applied to pin 2 of the SKT6 connector. (These links can be re-configured so as to enable +12 V power supplies to be attached, giving ±12 V signals on SERIAL OUT.)

8.7 The power supply

The HEKTOR power supply is shown in Figure 8.9. The mains connects to a protected transformer, which supplies 9 V r.m.s. to the HEKTOR microcomputer board via the 6-pin SKT8. The remaining components are mounted on the HEKTOR board, including the isolating switch, SW1. A full-wave bridge rectifier provides a 'raw' d.c. voltage of 10 volts (nominal). This is smoothed by a reservoir capacitor, and made available not only to the on-board regulator but also via pins 1 and 2 of any edge connector connected to the microcomputer board. HEKTOR itself takes less than 700mA from the power supply and an additional 100mA can be taken from the edge connector.

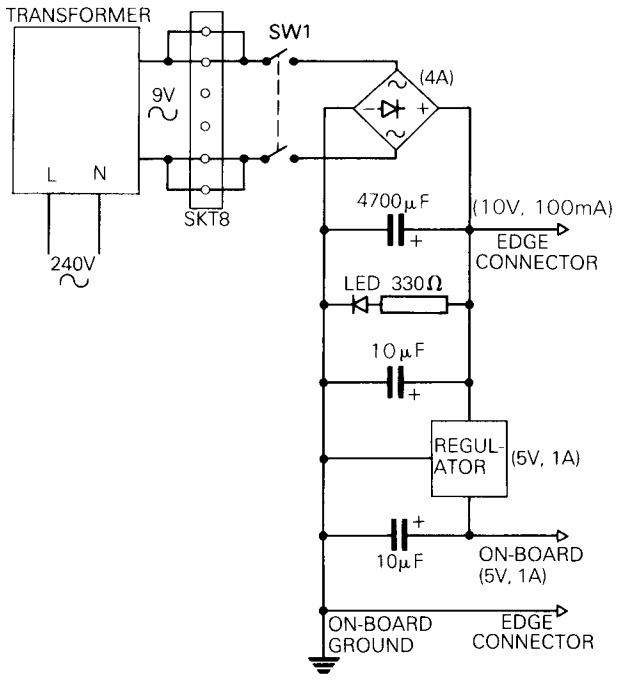


Figure 8.9 Power supply

The on-board regulator is mounted on a heat sink, whose fins prevent hot components being accidentally touched.

As well as the power supply decoupling capacitors around the regulator, there are several 100 nF ceramic capacitors at various points around the board which also perform this function where the power supply tracks connect to integrated-circuit devices.

8.8 Connection to the bus

It is possible to expand both HEKTOR's memory and I/O capability if off-board equipment is connected to the microcomputer bus, using the edge connector. It is not possible to list all the possibilities but some guidelines can be laid down.

Firstly, there are some electrical considerations. The bus is unbuffered, so the connections to a bus line should not present a load of more than one or two LSTTL devices. Bus capacitance will limit the cable length to a few inches for reliable operation, particularly for the ALE and CLK bus signals. The 'raw' d.c. supply, at 10 V nominal, should not be required to supply more than 100mA.

The address bus is available in both its multiplexed and expanded form, and the address ranges 4000–6FFF, and C000–EFFF (hex) are unused on the HEKTOR microcomputer board. However, the external equipment will have to arrange for its own address decoding.

The data transfer control signals ($\overline{RD}$, $\overline{WR}$, $\overline{ALE}$, HOLD, HLDA, READY, and IO/$\overline{M}$) are all available, and enable slow devices or DMA operations to be used, as well as conventional synchronous data transfers. Concerning interrupts and reset facilities, these also exist, although only one of the four special RST-type interrupts is available on the bus.

8.9 Connection to the serial line

The HEKTOR hardware is capable of sending and receiving RS232C-standard signals. There is a utility software package which will output 8-bit data according to the serial transmission standard, with even or odd parity, and at a variety of standard baud rates. It uses the BUSY signal as a 'clear-to-send' input.

Unfortunately, an external power supply is required before a serial line output device, such as a printer, can be used. Very little power is needed, so a battery is adequate for intermittent use, and a simple connection arrangement is shown in Figure 8.10. Note that if no 'printer busy' signal is available, then connecting the BUSY line to +5 V will enable continuous output of printer data. (This is adequate for electromechanical teletypes, for example.)

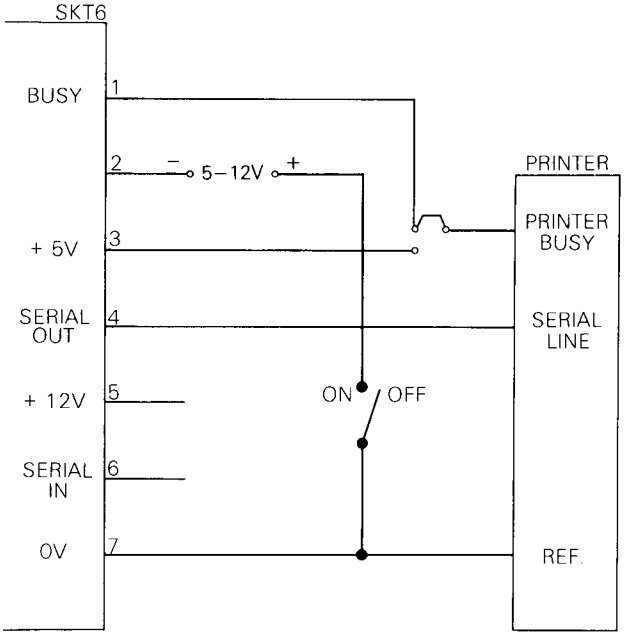


Figure 8.10 Printer connections

9 HEKTOR SYSTEM SOFTWARE

This section describes the HEKTOR system software with the aim of enabling user programs to interact with it successfully. Accordingly, the system data structure and the utility routines are described functionally. By using RAM-based *vectors* for interrupt and I/O handlers, and key I/O *parameters*, the design of the HEKTOR software has retained considerable freedom for advanced users to modify or extend the system software without having to replace the system ROMs. There is a summary of the more useful routines and data locations in Appendix A.

9.1 System data structure

Table 9.1 gives an overall description of the components of the system data structure. The system RAM, in the 8155 device, is conceived of as containing nine groups of variable data, each of which are vectors, parameters or buffers. *Vectors* are used both for interrupt service routine entry points and for the calls to I/O handler routines. These vectors are initialized on switch-on or following a RESET, to point to the ROM-based handlers for these functions, but subsequent operations (either the use of monitor commands or within a user program) can alter these vectors so that user-designed software can respond appropriately.

Table 9.1 RAM usage

Address	Usage	
2700 2711	Interrupt vectors	Initialized by RESET
2712 2723	I/O vectors	
2724 2733	Saved status buffer	
2734 2740	Interrupt parameters	
2741 2744	I/O parameters	
2745 2789	Keyboard buffers	Unused by MONITOR
278A 2791	(Editor parameters)	
2792 27BC	(Assembler parameters)	
27FF	Stack	
2800 ABUFE	(Text buffer)	
ABUFE+1 OBPTR	(Symbol table and code buffer)	
OBPTR+1 3FFF	(Unused)	

The saved status buffer is described in Section 4, and is used to keep a copy of the microprocessor registers' contents following interrupts (BREAK key, break-point, or single-step).

The interrupt parameters specify information associated with the break-point and real-time functions, whereas the I/O parameters define such things as the serial line output baud rate, the keyboard SHIFT LOCK state, and the TV output mode (graphic or normal).

There are three levels of keyboard buffer. The lowest holds the ASCII code for the last character keyed. The next buffer holds the previous line keyed, ready for processing by the monitor or editor command decoders. The highest-level buffer contains the decoded command line: command letter, three command argument values, and the line terminator code.

Next there are two workspace areas, which are used by the editor and assembler respectively. Finally, the rest of the system RAM is available as a stack. User programs can use up to 170 (denary) bytes for a stack, without interfering with the operation of the monitor or the system utilities. (The monitor will, however, report stack overflow if it is re-entered with a stack which encroaches on the assembler's data structure; that is with SP less than 27BC (hex).)

The user RAM area, addresses from 2800 (hex), is used for the editor's text buffer and for the assembler's symbol table and machine code buffer, but is otherwise unused by the system software.

9.2 Interrupt structure

This is summarized in Table 9.2. There are five hardware interrupt sources, and eight 'software' interrupts. (One, the RESET interrupt, is initiated either by hardware or software.) The hardware interrupt sources are connected to particular pins on the microprocessor, and signals on these pins cause the appropriate response by the microprocessor. The software interrupts are the one-byte RST instructions. When these instructions are executed, the subsequent behaviour of the microprocessor is similar to that of the hardware interrupts; hence their name.

When an interrupt is received by the microprocessor, its response is to call the routine whose address is given in Table 9.2 as the ROM address. For all but the RESET interrupt, the instruction at that address is a jump to a location in the system RAM. That location, in turn, contains another jump instruction. The second and third bytes of this instruction, therefore, define the address of the relevant interrupt handler, and form the *interrupt vector*.

Being in RAM, the contents of the interrupt vector (whose addresses are given in Table 9.2) can be

Table 9.2 Interrupt sources, vectors, and handlers

Source of interrupt	*Name*	*ROM address*	*Vector address*	*Default handler address*	*Handler function*
RESET key	RST0	0000	—	0000	Initialize system
(program)	RST1	0008	2701/2702	0000	,,
(program)	RST2	0010	2701/2702	0000	,,
(program)	RST3	0018	2701/2702	0000	,,
(program)	RST4	0020	2701/2702	0000	,,
(program)	RST5	0028	2701/2702	0000	,,
(program)	RST6	0030	2701/2702	0000	,,
break-point	RST7	0038	270A/270B	0223	Saves status, removes break-point exit to monitor
timer	TRAP	0024	2710/2711	0242	Saves status, exit to monitor
BUSY	RST5.5	002C	2704/2705	0000	Initialize system
BREAK key	RST6.5	0034	2707/2708	020C	Saves status, exit to monitor
(ext)	RST7.5	003C	270D/270E	0000	Initialize system

altered. The *handler* addresses shown in Table 9.2 are those which are set during system initialization, following a RESET or power-on. For example, when the BREAK key is pressed, the microprocessor effectively executes a CALL 0034 instruction. The three locations in RAM whose first address is 2706 contain a JMP 020C instruction. Finally, the handler proper, at 020C, performs the required actions.

This chain of actions can be intercepted by the user. If the BREAK interrupt vector is changed, namely the contents of the locations with addresses 2707 and 2708, the BREAK interrupts will be vectored to the new handler. Users may wish to develop their own handlers for any of the TRAP, RST5.5, RST6.5 or RST7.5 interrupts, and these can be linked into the system software by setting their starting address in the relevant interrupt vector.

The software interrupts can also be vectored to user-defined handlers. Each of the RST1–RST6 instructions is vectored via the address stored in 2701/2702, and RST7 is vectored via 270A/270B.

There are differences within the hardware interrupt group, as regards their detailed effects on the microprocessor, and any alternative handlers will have to take these into account. The differences centre around the *interrupt register* and the effect of the EI, DI, RIM and SIM instructions. The interrupt register can be considered as reflecting the status of the interrupt system as shown in Figure 9.1. When the RIM instruction is executed, this status information is transferred to the A register for subsequent processing. The SID, I 5.5, I 6.5 bits simply reflect the signal actually present on the indicated microprocessor pin. The I 7.5 bit is a true flag. That is, the flag is set to 1 by a 0–1 transition on the RST7.5 interrupt line, but is reset to 0 by a SIM instruction (see below). The enable flag is set or

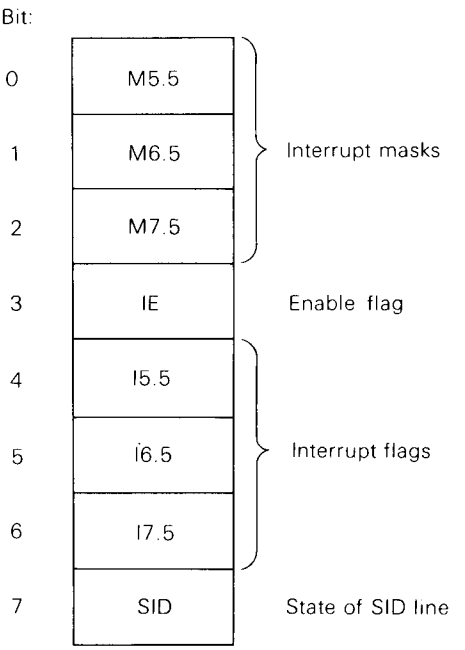


Figure 9.1 Interrupt status register

reset by the EI and DI instructions respectively. The three interrupt mask bits (M5.5, M6.5, M7.5) are set or reset by SIM instructions.

The SIM instruction performs operations which depend on the data in the A register when it is executed. There are three independent operations, as shown in Table 9.3 and all combinations of these operations are possible, depending on the 'enable' bits 3, 4 and 6. Two of these three operations are of interest here. Firstly, if the SIM instruction is executed with bit 4 of the A register set to 1, then the flag associated with the RST7.5 interrupt is reset to 0. (This operation is also performed automatically when the microprocessor responds to an RST7.5 interrupt.)

Table 9.3 Effect of SIM instruction

A register contents:								*Function*
7	6	5	4	3	2	1	0	
X	X	X	1	X	X	X	X	reset I 7.5 flag to 0
d	1	X	X	X	X	X	X	set SOD line to d
X	X	X	X	1	d	d	d	set mask bits to ddd

Table 9.4 Interrupt handlers

Procedure	*TRAP*	*RST7.5*	*RST6.5, RST5.5*
		DI	DI
Initialization:	Set vector (Init. timer)	Set vector (Init. ext hardware) SIM (A=1B) EI	Set vector (Init. hardware) SIM (A=0D, or 0E) EI
Handler	Save registers	Save registers	Save registers Wait for I 6.5 or I 5.5 to return to 0
	Perform function Restore registers RIM RET	Perform function Restore registers EI RET	Perform function Restore registers EI RET

Secondly, the three interrupts (RST5.5, RST6.5, RST7.5) can be individually *masked*. That is, the microprocessor will respond to an interrupt from one of these sources if and only if:

- the signal has been received (flag bit = 1),

and

- interrupts are enabled overall (enable flag = 1),

and

- interrupt mask bit = 0.

By selectively setting or resetting the mask bits, while leaving the interrupt system enabled as a whole, particular interrupts can be masked off. In fact, the HEKTOR system software executes with the M5.5 and M7.5 mask bits set to 1, but the M6.5 bit reset to 0, allowing BREAK key interrupts, but not BUSY or external interrupts. Note that the TRAP interrupt cannot be masked or disabled, so interrupts from the onboard timer are always recognized. (The timer is used primarily for single-step monitor operations.)

The RST5.5 and RST6.5 interrupts are *level-sensitive*; that is, they supply an interrupt *request* whenever the corresponding input pin is at the 1 level. (Whether this request is *acknowledged* by the microprocessor depends on the mask and enable bits in the interrupt register, and interrupts are automatically disabled when an interrupt is acknowledged.) RST7.5 is *edge-sensitive*; that is the interrupt request follows a 0–1 transition on the input pin, and persists until the interrupt is acknowledged, or the RST7.5 flag is reset by a SIM instruction. A TRAP interrupt request requires *both* conditions to be satisfied; a 0–1 transition must have occurred and the input level must remain at 1. These differences are reflected in the suggested procedure for handling interrupts given in Table 9.4. The SIM instructions clear the relevant mask bits, and in the case of RST7.5 also clear the interrupt flag. On receipt of an RST5.5 or RST5.6 interrupt, the handler should wait until the interrupt signal disappears, to prevent immediate re-interrupting. Finally, before returning from interrupt, the interrupt system should be re-enabled. In the case of the TRAP interrupt, the RIM instruction restores the interrupt enable flag to its state prior to the TRAP interrupt; interrupts are disabled whenever *any* interrupt is serviced.

Table 9.1 not only shows that there are the interrupt vectors in the system RAM, but also that there are interrupt *parameters*. Most of these parameters are connected with timer interrupts (TRAPS), but two parameters are associated with the *break-point* handler.

The monitor's B command inserts a break-point in a user program by replacing the code at the specified address by the RST7 instruction (FF (hex)). Both the address of the location and its original contents are saved as parameters of the break-point function; the address is stored in the locations with addresses 2734/2735 and the contents in the location with

address 2736 (hex). When the break-point is encountered, the handler (which is entered via the RST7 vector) can then restore the original code to the break-point location. When no break-point is specified, dummy values are used for the parameters.

A final point on interrupts concerns the saved status buffer (see Table 9.1). All of the interrupt handlers in the HEKTOR system software (RESET, BREAK key, single-step, and break-point) save the processor status in the saved status buffer (see Table 9.5). Note that the first two bytes are used for instructions. This is to enable subsequent G or 1 commands to continue user program execution by a jump to 2724 (hex), with *all* registers correctly restored.

Table 9.5 Saved status buffer

Address	*Contents*
2724	EI/DI
	JMP
2726	Saved PC
2728	Saved SP
272A	Saved AF
272C	Saved BC
272E	Saved DE
2730	Saved HL
2732	Saved I

9.3 I/O structure

Table 9.1 shows that an area in system RAM is used for I/O vectors. These vectors are used in exactly the same way as the interrupt vectors. That is, when the system software calls an I/O handler, it does so with reference to a RAM-based address where a jump instruction vectors the call to the ROM-based handler itself. For example to output a character to the TV display, a CALL 2715 instruction is used. The locations 2715/2716/2717 contain the instruction JMP 0931, and the TV output routine is located at address 0931. In this way, a user could redirect all system software output to an alternative handler simply by changing the I/O vector, namely the contents of the locations with addresses 2716/2717. Table 9.6 shows the I/O vectors used, for the keyboard and TV, and for the cassette handlers. Note that the H command of the monitor is also vectored. This provides a mechanism by which the user can 'escape' from the monitor to an arbitrary program instead of the default BASIC interpreter.

9.4 TV handlers

The basic TV output handler (TV) is accessed by the system software via the vector shown in Table 9.6. This handler simply outputs a byte to the TV interface hardware, and is described below. However, there are a number of non-vectored utilities which themselves call TV via the vector, and which perform a variety of pre-processing operations which are generally useful. For example, an 8-bit code can be converted for display into two hex digits. These routines are also described below, and are summarized in Table 9.7.

9.4.1 TV output

The TV routine (address 0931) is used for all system output to the TV interface. It uses the code in the A register as a parameter, as summarized in Table 9.7. Some of the 'control' codes (00–1F (hex)) perform cursor movements, but if the code is in the range 20–7F (hex), a character is displayed (see Appendix C) and the cursor moved right. If the code is in the range 80–FF (hex), a graphic character is displayed. These characters enable simple pictures to be drawn on the screen, but they do not conform to any of the many 'standard' sets of graphic characters used in some microcomputer systems.

A second TV output routine, TVT, is similar to TV, but instead of converting codes 00–1F into cursor movements it simply outputs them as 'characters' from the graphics set.

Both routines wait, usually for ten milliseconds, to allow the TV hardware to accept the output data, and then return with the contents of all registers intact.

Table 9.6 I/O Vectors

Name	*Vector address*	*Default handler address*	*Handler function*
CINV	2713/2714	077D	KR: Awaits key depression; returns with ASCII in A register
COUTV	2716/2717	0931	TV: Outputs ASCII in A register to TV display
TINV	2719/271A	056F	CRL: Loads data from cassette
TOUTV	271C/271D	0488	CRS: Saves data on cassette
TVERV	271F/2720	056F	CRL: Verifies data on cassette
HCMDV	2722/2723	00F6	Exit to BASIC interpreter

Table 9.7 Summary of display routines

Name	*Address*	*Behaviour*
TV	0931	Sends (A) to TV hardware. 0D=Carriage return, 0A=line feed, 0C=clear/home, 08=cursor left, 09=cursor right, 0B=cursor up, 20–7F=ASCII char, 80–FF=graphics.
TVT	092B	As TV, but no conversion of codes 00–1F
PRERR	0324	Displays 'ERROR'. Exit to monitor at 0057
PRNL	032A	Displays carriage return, line feed
PRSP	0347	Displays space
PRCOL	0342	Displays colon, space
PRMES	0368	Displays codes in table pointed to by (HL), until zero encountered
PRB	03BB	Displays (A) as 2 hex digits
PRWD	03AF	Displays (HL) as space, 4 hex digits
PRBI	03B7	Displays byte pointed to by (HL) as space, 2 hex digits
PRWI	03A5	Displays 16-bit code pointed to by (HL) as space, 4 hex digits

9.4.2 Special messages

This group of routines makes use of calls via the COUTV vector at 2716/2717 and so normally outputs using the TV routine. PRERR (address 0324) displays 'ERROR' and then enters the monitor at 0057. That is, the system RAM initialization is bypassed, and the monitor displays its prompt character.

PRNL, PRSP and PRCOL perform the actions summarized in Table 9.7. The contents of the microprocessor registers are left intact, except that PRCOL changes the contents of the AF registers. PRMES displays the 'message' string of bytes pointed to by HL, interpreting them as ASCII codes, until the zero code is encountered. PRMES then returns, with all register contents undisturbed, except AF, HL.

9.4.3 Output with hex conversion

The last four routines in Table 9.7 perform conversions on 8-bit or 16-bit codes, so as to display them as strings of hexadecimal characters, rather than interpreting them as ASCII codes. Two of these routines output 8-bit data, and two of them output 16-bit data. For example, if the A register contains 42 (hex), CALL TV will display 'B', whereas CALL PRB will display '42'.

These routines output via the COUTV vector in 2716/2717 and leave the registers' contents intact, except for the AF registers' contents.

9.5 Keyboard handlers

There are three levels of keyboard handling. At the lowest level, individual characters are accepted from the keyboard, to give an ASCII code for the key, possibly modified by the SHIFT or CTRL keys. At the next level, there is a routine which accepts and stores a whole line of characters. The highest-level routines are used by the system software for accepting and decoding a command line into its constituent parts.

Those routines likely to be useful to users are summarized in Table 9.8 and are described in more detail below.

Table 9.8 Keyboard routines

Name	*Address*	*Behaviour*
KSTAT	078A	Returns NZ if key pressed (not SHIFT, CTRL)
KR	077D	Awaits key. Stores code (modified by SHIFT/CTRL) and returns with code in A. Sets flags if special key
KLUC	0850	Converts (A) to upper case ASCII if lower case
KGRAF	0859	Converts (A) to graphic code if flag set
KBUF	0663	Enter with HL=buffer end address. Accepts line into buffer until terminator character. Echoes input

9.5.1 Accepting characters

The KSTAT routine (address 078A) returns immediately with the Z flag set if no character key on the keyboard is activated. If there is a key detected, it is 'debounced' using a delay parameter in KDEL (at 2743). Assuming a valid key depression is in progress a return from KSTAT occurs after the key is encoded. If SHIFT LOCK is detected, KSTAT returns with Z set, but the system parameter KFLAG (at 2744) is altered so as to record the new SHIFT LOCK status. For other keys, the ASCII code is computed, taking into account the current state of the CTRL and SHIFT keys and the SHIFT LOCK status. KSTAT returns with this ASCII code in the accumulator (and stored in CHAR, at 2746) and with the Z flag cleared. A side effect of KSTAT is to set/clear bit 7 of KFLAG whenever CTRL-G/CTRL-A is detected. In this way the CTRL-G and CTRL-A key actions can be used to turn on or off the 'graphics' mode.

Note that the cursor keys return the following codes:

- ↑ 0B (hex), same as CTRL-K;
- ↓ 0A (hex), same as CTRL-J;
- RETURN 0D (hex), same as CTRL-M;
- → 09 (hex), same as CTRL-I;

- ← 08 (hex), same as CTRL-H;
- HOME 0C (hex), same as CTRL-L.

The KR routine awaits the 'no key' condition, if necessary, before accepting and encoding the next key depression. Any SHIFT LOCK is internally processed, within KR, and a return from KR only occurs when some key other than SHIFT LOCK, SHIFT or CTRL is depressed. KR operates entirely by KSTAT calls.

The KLUC routine converts the ASCII code in the A register to an upper-case alphabetic ASCII code, if it was a lower-case code. This routine is used by the system software, for all keyboard input. The KGRAF routine converts the ASCII code in the A register to the corresponding graphic code (bit 7 = 1), if the graphic mode flag (that is, bit 7 of KFLAG) indicates 'graphics on'.

9.5.2 Accepting a line

The only routine used by the system software for accepting a line of characters from the keyboard is KBUF (see Table 9.8). This routine calls KR via the CINV vector repeatedly, and 'echoes' the typed-in characters using the TV routine (again vectored via COUTV). The input ASCII codes are stored in the line buffer which starts at address 274D. The routine is parameterized, with the address of the end of the line buffer in HL. For example, if KBUF is called with HL containing 276D (hex) the line buffer is defined as comprising 20 (hex) locations. KBUF is a complex routine, since it not only accepts characters, but also handles the display and the line buffer.

KBUF accepts characters, using the KR routine, and converts lower-case to upper-case using KLUC. If the character is a non-control character (that is, an ASCII code in the range 20–7F (hex)), then it is echoed on the screen and inserted in the line buffer. (If the end of the line buffer has been reached, the character is neither inserted nor echoed.) The next character is then awaited.

The left and right cursor keys cause the appropriate cursor movement on the display, but this is accompanied by the display, and storage in the line buffer, of a space character. Thus, keying right cursor is identical to keying a space, and the left cursor is effectively a 'rub-out' key. Again, these cursor movements are ignored if their implementation would cause operations outside the limits of the line buffer.

KBUF accepts no more characters when one of four terminators is keyed. A number is then stored in the TERM variable (address 2745) which specifies the terminator thus:

- ↑ –store FF in TERM
- RETURN –store 00 in TERM
- ↓ –store 01 in TERM
- CTRL-P –store 02 in TERM

The terminator is neither echoed nor stored in the line buffer. Instead, any unused locations in the line buffer are filled with zeros, and KBUF returns.

Apart from the cursor movement and terminator keys, KBUF ignores all keys with ASCII codes in the range 00–1F (hex).

9.6 Cassette handlers

Data is recorded on cassettes using the format shown in Figure 9.2. Individual bits occupy a fixed length of tape, defined by a reversal of magnetization at each end. As shown in Figure 9.2(a), there may be an additional magnetization reversal in the middle of the bit position. If there is this additional reversal, the bit value is 1; if not, the bit value is 0. Bits are recorded at a rate of about 1300 per second (determined by the software driver).

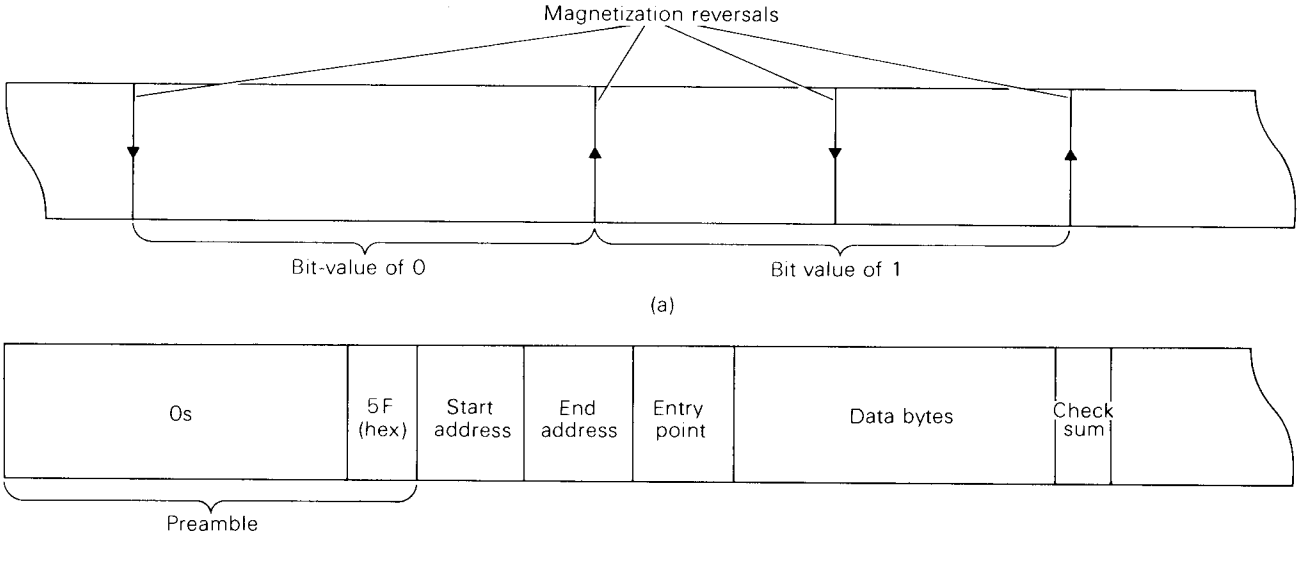


Figure 9.2 Cassette reading format

Looking at a recorded block of data as a whole, it consists of the sections shown in Figure 9.2(b). The first section comprises a ten-second 'preamble', which enables the cassette-reading routine to synchronize to the recorded signals (after the few seconds of blank leader on most cassettes) in preparation for acceptance of the recorded data. The preamble comprises a long sequence of 0s followed by a standard identification character (5F hex).

The subsequent data is recorded as continuous 8-bit quantities. The first six bytes define the addressing information associated with the subsequent data bytes. There are therefore three 16-bit addresses defining respectively:

- The starting address in memory of the subsequent data;
- The last address of the subsequent data;
- The entry address for the data (for the case where it is an assembler-generated program of machine instructions.)

There then follows the data itself. The number of bytes will be end address − start address + 1. Finally, there is a *checksum* byte. This is an error-detection code, whose value is such that if all the bytes on the tape are added together (including the six bytes of addressing information), the least significant 8 bits of the sum must be the same as the checksum. When reading the tape, the system software does this arithmetic and comparison, and thereby is able to verify that the reading was accurate. (Peculiar combinations of errors in recording or reading the tape could give an apparently valid checksum, but the likelihood is very low.)

The main cassette-handling routines are CRS and CRL, and are described in more detail below.

9.6.1 Saving data

The system software uses the CRS routine (called via the TOUTV vector; see Table 9.6) to record a block of data according to the format of Figure 9.2. This routine makes use of several parameters, some of which are in the system RAM, and some passed to the routine in the registers, as shown in Table 9.9.

Table 9.9 Parameters for CRS

Name	*Address*	*Comment*
ARG1	2747/2748	Start address recorded on tape
ARG2	2749/274A	End address recorded on tape
UPC	2726/2727	Entry point recorded on tape
	HL reg	Start address for recorded data block
	DE reg	End address for recorded data block

The three addresses recorded on the tape at the start of the record are the values of the ARG1, ARG2 and UPC system variables. The data bytes recorded are taken from the memory locations whose address range is specified by the HL and DE registers. (Although the addresses in HL and ARG1 are usually the same, as are those in DE and ARG2, this system enables data to be recorded from one area of memory for subsequent loading back into a different area.)

CRS starts by displaying the 'SET RECORD' message, and awaiting a key depression to indicate that the recorder controls have been set for recording. It then calls the CRON routine (at address 0529) to turn on the recorder's motor. It then waits for about ten seconds while recording the preamble of 0s. The address, data, and checksum bytes are then recorded, the motor is turned off (calling CROFF at 0532) and CRS returns. Note that CRS uses the timer in the 8155 device for bit-timing, via TRAP interrupts.

9.6.2 Loading data

The system software uses the CRL routine (called via the TINV vector, see Table 9.6) to load a block of data recorded according to the format of Figure 9.2. This routine is called with the carry flag as a parameter. If the carry is 1, the *loading* of the data is performed, as described below. If the carry is 0, *verification* of the tape is performed. Verification is identical to loading, except that the data read from tape is not actually loaded into memory. A second parameter is the contents of the BC register pair. This is used as a loading offset. That is, the start and end addresses read from tape are each augmented by the contents of BC to delimit the area of memory into which data is actually loaded from tape. Normally, this offset will be zero, but it does offer some flexibility in loading, via buffers for example.

CRL starts similarly to CRS. That is, a dialogue involving the message 'SET TO PLAY' ends with the recorder's motor being turned on. The incoming signal from the cassette interface is then sampled at a rate of once every 125 microseconds (using the 8155 timer attached to the TRAP interrupt). For the initial 0s in the preamble section, each magnetization-reversal detected corresponds to the start of a bit, and its value is deduced as 0 because the next reversal occurs after a 'long' time. CRL requires that a continuous record of at least twelve 0s followed by the identification character (5F) be detected after which it assumes that the preamble has passed.

The subsequent bits are assembled into bytes, the first six bytes being used as address information, and subsequent bytes as data. The checksum, as computed from the bytes read from tape, is compared with the checksum read from tape, and CRL returns.

If the preamble-seeking operation fails, CRL will

continue to seek it, and the operation will have to be manually aborted (using the BREAK or RESET keys, for example). Otherwise CRL will return with various parameters. The DE register pair will contain the actual memory address into which the last data byte was loaded (or would have been loaded, if CRL is being used for verification). The Z flag indicates whether there was a checksum error (Z = 1 means 'correct') and the UPC variable (address 2726/2727) contains the entry address read from tape. Note that CRL does not turn the recorder motor off.

9.7 Serial line handlers

The standard serial line format is shown in Figure 9.3 as it appears on the SERIAL OUT line of the interface. Each character is specified by its 7-bit ASCII code, and embedded in an 11-bit format which includes a start bit, a parity bit, and two stop bits. When the line is idle, it is maintained at the lower voltage, and the start bit (always the high voltage) synchronizes the receiving equipment for the reception of the subsequent seven data bits. These data bits are signalled by the line being at a high or low voltage for the same period T. The data bits are transmitted starting with the least significant bit, and with a bit value of 0 corresponding to the *higher* voltage.

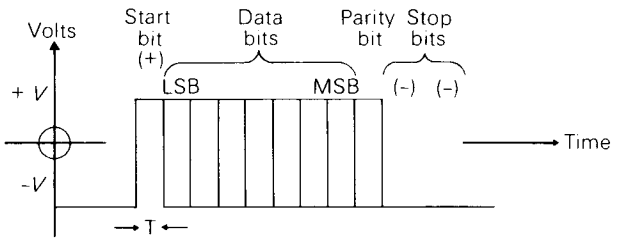


Figure 9.3 Serial line format

Following the seven data bits, the parity bit is transmitted. The value of this bit reflects the number of data bits which are 1, and acts as an error-detection facility. Following the parity bit, two stop bits (low voltage) are transmitted, leaving the line in the idle state ready for the next character.

Serial line data transmission and reception involve four system routines:

- SSET (at address 03D3)
- SOUT (at address 0883)
- SIN (at address 08EA)
- SSTOP (at address 0258)

Of these, SSET and SSTOP start and stop (respectively) the timer-interrupt system in a manner suitable for timing the serial line bits. Therefore, SSET should be called before performing any input or output via the serial line. Calling SSTOP afterwards is not essential, unless cassette I/O is required, but the timer interrupts do consume about half the available processor time.

SOUT is the routine for sending a 7-bit ASCII code via the serial line. It assumes that SSET has been called previously, and outputs the character in the format shown in Figure 9.3. The parity bit is computed according to the RAM-based I/O parameter SOPAR (at address 2741). Meaningful values for SOPAR are given in Table 9.10, and the value matching the requirements of the receiving equipment attached to SERIAL OUT should be set up before calling SOUT. (The default value is E0 for odd parity.)

Table 9.10 Baud rate/parity parameters

Baud rate	SOBAUD (*hex*)	*Parity*	SOPAR (*hex*)
1200 (default)	07	odd (default)	E0
600	0D	even	C0
300	1B	mark	80
110	49	space	00

The second parameter defines the effective baud rate for the serial line; that is, the rate of transmission of bits, or 1/T bits per second in terms of Figure 9.3. Table 9.10 shows the 8-bit value required in SOBAUD (address 2742) for various standard baud rates.

SOUT is called with the 7-bit ASCII code in the A register (all register contents are undisturbed on return). The parity bit value is computed, using the SOPAR parameter. The routine then waits until the BUSY line is at high voltage level by continually examining the I 5.5 flag (see Subsection 9.2). It then outputs the 11-bit pattern described by Figure 9.3, and returns.

SIN is the routine for accepting an ASCII code from the SERIAL IN line, assuming SSET has been called previously. SIN awaits a negative-going transition on the SERIAL IN line, and assumes that this represents the beginning of a START bit (see Figure 9.3). Using the current SOBAUD value, it then samples the line at appropriate intervals to construct an 8-bit code (7 data bits plus parity, in the MSB position), and returns with this code in the A register, and with other registers undisturbed. Since it does not wait to detect the stop bits, there is time to process this character and call SIN again for an immediately-following character, if any.

9.8 Processing utilities

There are three groups of system utilities, other than the I/O routines, that may prove useful. One group offers a range of software-timed delays, another

facilitates the use of BCD arithmetic. The third group performs binary multiplication and division. These are summarized in Table 9.11. The delay routines are carefully constructed loops. The timing of DELAY1 is accurate to two microseconds, and includes the execution time for the CALL DELAY1 instruction.

Table 9.11 Processing utilities

Name	*Entry address*	*Behaviour*
DELAY1	09A3	Enter with value in HL (>1). Returns after HL/128 millisecond with A, HL set to 0
DELAY2	09AB	Enter with value in A. Returns after A seconds with A set to 0
BCDADD	0FA3	(HL)=(HL)+(DE) as 4-digit BCD. (HL)=9999 and carry if overflow
BCDSUB	0F90	(HL)=(HL)−(DE) as 4-digit BCD. (HL)=0000 and no carry if overflow
DMUL	1A97	(HL)=(A)×(B) 2's complement multiplication
DSUB	1AB2	(A)=(HL)/(B) 2's complement division

The BCD routines assume that valid four-digit unsigned BCD integers are stored in the HL and DE registers. The value returned in HL is the correct BCD result unless overflow has occurred, in which case the limiting values are set in HL. (The carry flag indicates whether overflow has occurred.)

The routines DMUL and DSUB perform multiplication and division on binary-coded unsigned integers. If the A and B registers contain 8-bit integers (0–255), then DMUL returns their 16-bit product (0–65535). Similarly, if HL contains a 16-bit unsigned integer, and B contains an 8-bit integer (0–127), then DDIV returns the 8-bit quotient in the A register. If the correct quotient exceeds the available range (0–255), then its least significant 8 bits are returned. The special case of dividing by zero returns 255 (decimal). Note that non-integer quotients are rounded down to the nearest integer. For example, 70/71 returns zero.

9.9 Tune machine

There is a demonstration program which will play 'tunes' via the TV loudspeaker, by interpreting codes representing musical notes and sequencing information.

There are two entry points for the 'tune machine' program. Execution from 1B38 (hex) plays a standard tune and returns to the monitor. Execution from 1B43 (hex) plays a user-defined tune, whose list of 'musical instructions' is stored in the user RAM, starting at address 2810 (hex). The 'instruction set' is summarized in Tables 9.12 and 9.13.

Table 9.12 Note instructions for tune machine

Note codes		*Note*
Short	*Long*	
75	F5	F (lowest)
6E	EE	G
68	E8	G#
62	E2	A
5C	DC	A#
56	D6	B
52	D2	middle C
4D	CD	C#
48	C8	D
44	C4	D#
40	C0	E
3C	BC	F
39	B9	F#
35	B5	G
32	B2	G#
2F	AF	A
2C	AC	A#
29	A9	B
26	A6	C
24	A4	C#
22	A2	D
20	A0	D#
1E	9E	E
1C	9C	F (highest)
00	80	pause

Table 9.13 Control instructions for tune machine

Control codes	*Effect*	*Initial value*	*Examples:*
FB XX	Set duration of short notes to XX	28	10 is fast, 40 is slow
FC XX	Length factor for long notes	02	04 means long notes four times as long as short
FE XX	'Instrument' type	FF	FF is 'piano', 00 is 'clarinet'
FF XX YY	Jump to tune at address YYXX (hex)	—	
FA	Stop and exit to monitor	—	
F9 XX YY	Call 'subroutine' at YYXX	—	
F8	Return from 'subroutine'	—	

The 'note' codes shown act as parameters of the timing loop which determines the frequency of the note played. The codes given are approximations to the diatonic musical scale over two octaves, but intermediate values can be used. Bit 7 of the note code indicates whether the note lasts for a 'short' or a 'long' time. The short notes last for a period of time specified by a parameter that can be changed by using the FB XX control code (that is, FB (hex) is interpreted as meaning 'change duration', and the next byte in the music program is an operand defining the new duration). The initial tone duration for short notes (parameter value = 28 (hex)) is about 200 milliseconds.

The duration for the 'long-note' codes is given by another parameter, which is a multiplying factor with respect to the prevailing 'short-note' duration.

The parameter is initally 2, but can be changed by a two-byte control instruction, whose coding is of the form FC XX. The third musical parameter is the 'instrument' type, changed by the FE FF or FE 00 control instructions. The 'clarinet' mode produces a continuous tone for the duration of the note, but the 'piano' mode produces a harmonically rich percussive sound.

The remaining control instructions are for sequencing, in an analogous manner to the sequence control instructions in a conventional program. The three-byte 'jump' instruction (FF XX YY) causes the tune machine program to start interpreting the music program beginning at the memory address YYXX (hex). The 'call' and 'return' instructions enable the 'music programmer' to include standard or repeated musical phrases (or 'subroutines').

9.10 SORT routine

This is an application subroutine which sorts the contents of a specified set of memory locations into ascending order. The entry point for this routine is 1FD5 (hex).

The data to be sorted should be stored in consecutive locations in RAM. The sorted data will be left in the same set of memory locations. On entry to the subroutine, the number of data items to be sorted should be held in the accumulator and the memory address of the first data item should be held in the HL register pair.

The data is treated as a set of (binary) unsigned integers in the range 00–FF. If the data used is interpreted as ASCII codes for alphabetical characters of a *single case only* (either upper-case or lower-case) the result will be that the characters are sorted into alphabetic order.

The routine works by successively comparing adjacent data bytes. If the data bytes in a given pair are not in ascending order it swaps them. The routine checks every adjacent pair in this way until no swaps are needed on a complete pass through the data.

APPENDICES

APPENDIX A: HEKTOR MEMORY MAPS

The 8085 microprocessor in HEKTOR uses 16-bit addresses. That is, up to $2^{16} = 65536$ different memory locations can be individually addressed, when storing a byte of data in memory or retrieving a byte from memory. However, not all of these 65536 potential memory locations actually exist as physical storage locations in HEKTOR, and some of the addresses refer to I/O devices rather than memory locations. As an additional complication, some storage locations and I/O devices will respond to any of several addresses, although only one address is used by the system software. However, the converse is not true; for each address, there is at most one physical device that will respond.

Table A.1 is a *memory map* for HEKTOR, showing what physical device, if any, corresponds to each 16-bit address. The figure also shows the main usage of that device, and, for multiple addresses, the address actually used by the HEKTOR system software. Note that addresses are specified as hexadecimally coded numbers (see Appendix B).

Tables A.2 and A.3 are memory maps summarizing the key addresses used by the system software. Table A.2 gives entry points for ROM-based system routines, and Table A.3 the usage of the on-board RAM memory. More details are given in Section 9.

Table A.1 HEKTOR memory map

Hexadecimal address	*Corresponding physical device*	*Principal usage by system software*	*Address range used by software*
0000–1FFF	2 × 4K ROM devices	Storage of system programs/ routines	0000–1FFF
2000–23FF	TV output data port	TV interface	2000 only
2400–27FF	$\frac{1}{4}$K RAM in 8155 device	Workspace/stack for system programs	2700–27FF
2800–3FFF	3 × 2K RAM devices	User read/write memory	2800–3FFF
4000–6FFF	None	None	None
7000–7FFF	Same devices as for addresses F000–FFFF		F000–FFFF
8000–9FFF	Same devices as for addresses 0000–1FFF		0000–1FFF
A000–A3FF	Same devices as for addresses 2000 to 23FF		2000 only
A400–A7FF	I/O ports of 8155 device	Keyboard, TV, cassette control	A400–A405
A800–BFFF	Same devices as for addresses 2800 to 3FFF		2800–3FFF
C000–EFFF	None	None	None
F000–FFFF	1 × 4K ROM device	Storage of BASIC interpreter	F000–FFFF

Table A.2 ROM entry points

Address	*Name*	*Comment*
0000	MONC	Monitor 'cold' entry
0057	MONW	Monitor 'warm' entry
00F6	HCMD	BASIC entry
0258	SSTOP	Serial I/O disable
0324	PRERR	Displays error message
032A	PRNL	New line on display
0342	PRCOL	Displays colon
0347	PRSP	Displays space
0368	PRMES	Displays string of characters
03A5	PRWI	Displays 4 hex digits, HL=pointer
03AF	PRWD	Displays 4 hex digits, contents of HL
03B7	PRBI	Displays 2 hex digits, HL=pointer
03BB	PRB	Displays 2 hex digits, contents of A
03D3	SSET	Serial I/O enable
0488	CRS	Saves data on cassette
0529	CRON	Starts cassette motor
0532	CROFF	Stops cassette motor
056F	CRL	Loads/verifies data on cassette
0663	KBUF	Accepts line from keyboard
077D	KR	Accepts key from keyboard
078A	KSTAT	Tests if key pressed
0850	KLUC	Converts lower to upper case
0859	KGRAF	Converts to graphic code
0883	SOUT	Serial line output
08EA	SIN	Serial line input
092B	TVT	TV output, no conversion
0931	TV	TV output, control code conversion
09A3	DELAY1	Short delay
09AB	DELAY2	Long delay
0B5E	EDITC	Editor entry (cold)
0B65	EDITW	Editor entry (warm)
0F90	BCDSUB	BCD subtraction
0FA3	BCDADD	BCD addition
101C	ASSEM	Assembler entry
1A97	DMUL	Binary multiplication
1AB2	DDIV	Binary division
1B38	TMS	Tune machine (standard tune)
1B43	TMU	Tune machine (user tune)
1FD5	SORT	Alphabetical sort, HL=pointer, A=no. of characters
1FFF	—	End of system ROM

Table A.3 System RAM usage

Address	*Name*	*Comment*
2701	RST1V	Vector for RST1–6 software interrupts
2704	RST55V	Vector for BUSY interrupts
2707	RST65V	Vector for BREAK interrupts
270A	RST7V	Vector for break-point interrupts
270D	RST75V	Vector for RST7.5 interrupts
2710	TRAPV	Vector for timer interrupts
2713	CINV	Vector for command input (keyboard)
2716	COUTV	Vector for command output (TV)
2719	TINV	Vector for load input (cassette)
271C	TOUTV	Vector for save output (cassette)
271F	TVERV	Vector for verify input (cassette)
2722	HCMDV	Vector for H command exit
2724	—	(workspace for saved status buffer)
2726	UPC	Saved PC
2728	USP	Saved SP
272A	UAF	Saved AF
272C	UBC	Saved BC
272E	UDE	Saved DE
2730	UHL	Saved HL
2732	UI	Saved I
2734	—	(interrupt parameters)
2741	SOPAR	Serial line output parity
2742	SOBAUD	Serial line output baud rate
2743	KDEL	Key debounce parameter
2749	KFLAG	Lock/graphic flags
2745	TERM	Type of terminator keyed
2746	CHAR	Last character keyed
2747	ARG1	Start address for cassette
2749	ARG2	End address for cassette
274B	—	(Monitor workspace)
274D	BUFFS	Line buffer for KBUF
278A	—	(Editor workspace)
2792	—	(Assembler workspace)
27FF	—	End of stack
2800	—	(Editor text buffer) (Assembler symbol table) (Assembler code buffer)
3FFF	—	End of on-board RAM

APPENDIX B: HEXADECIMAL CONVERSION TABLES

In the hexadecimal system, two-digit codes, ranging from 00 to FF, are equivalent to 8-bit binary codes, and four-digit codes (0000 to FFFF) are equivalent to 16-bit binary codes. These codes are clearly compact, and since each hexadecimal digit exactly corresponds to a group of four bits in the equivalent binary code, translation to and from binary codes is straightforward.

Conversion between denary (decimal) and hexadecimal codes is less straightforward, so conversion tables are useful. Table B.1 gives the denary equivalent for all two-digit codes of the form YZ, where Y and Z are each hexadecimal digits. Thus, the denary equivalent of A2 (hex) is found in the A^{th} row and the 2^{nd} column of Table B.1, and is 162.

Table B.2 gives the denary equivalent for four-digit hexadecimal codes of the form WX00, where the least significant pair of digits are both zero. For example, BE00 (hex) has the denary equivalent 48640. The tables can be combined, to convert between denary and any four-digit hexadecimal codes. For example, ABCD (hex) = AB00 + CD (hex) = 43776 + 205 = 43981 (denary).

Table B.1 Denary values for YZ (hex)

Y	Z = 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Table B.2 Denary values for WX00 (hex)

W	X = 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	256	512	768	1024	1280	1536	1792	2048	2304	2560	2816	3072	3328	3584	3840
1	4096	4352	4608	4864	5120	5376	5632	5888	6144	6400	6656	6912	7168	7424	7680	7936
2	8192	8448	8704	8960	9216	9472	9728	9984	10240	10496	10752	11008	11264	11520	11776	12032
3	12288	12544	12800	13056	13312	13568	13824	14080	14336	14592	14848	15104	15360	15616	15872	16128
4	16384	16640	16896	17152	17408	17664	17920	18176	18432	18688	18944	19200	19456	19712	19968	20224
5	20480	20736	20992	21248	21504	21760	22016	22272	22528	22784	23040	23296	23552	23808	24064	24320
6	24576	24832	25088	25344	25600	25856	26112	26368	26624	26880	27136	27392	27648	27904	28160	28416
7	28672	28928	29184	29440	29696	29952	30208	30464	30720	30978	31232	31488	31744	32000	32256	32512
8	32768	33024	33280	33536	33792	34048	34304	34560	34816	35072	35328	35584	35840	36096	36352	36608
9	36864	37120	37376	37632	37888	38144	38400	38656	38912	39168	39424	39680	39936	40192	40448	40704
A	40960	41216	41472	41728	41984	42240	42496	42752	43008	43264	43520	43776	44032	44288	44544	44800
B	45056	45312	45568	45824	46080	46336	46592	46848	47104	47360	47616	47872	48128	48384	48640	48896
C	49152	49408	49664	49920	50176	50432	50688	50944	51200	51456	51712	51968	52224	52480	52736	52992
D	53248	53504	53760	54016	54272	54528	54784	55040	55296	55552	55808	56064	56320	56576	56832	57088
E	57344	57600	57856	58112	58368	58624	58880	59136	59392	59648	59904	60160	60416	60672	60928	61184
F	61440	61696	61952	62208	62464	62720	62976	63232	63488	63744	64000	64256	64512	64768	65024	65280

APPENDIX C: 8-BIT ASCII CODES

Table C.1 shows the standard coding scheme used for the representation of character information in most 8-bit microcomputers, including the system software of HEKTOR. The standard ASCII code (American Standard Code for Information Interchange) is used, but as this defines character coding only in terms of seven bits (to allow 128 different characters and data-communication codes), the eighth bit, the most significant, is usually set to zero, for storage and communication using the 8-bit-orientated 8-bit microcomputers.

Table C.1 8-bit ASCII codes for character data (hex)

*Character**	*ASCII*		*Character*	*ASCII*		*Character*	*ASCII*		*Character*	*ASCII*	
	hex	*decimal*		*hex*	*decimal*		*hex*	*decimal*		*hex*	*decimal*
NUL	00	00	SPACE	20	32	@	40	64	`	60	96
SOH	01	01	!	21	33	A	41	65	a	61	97
STX	02	02	"	22	34	B	42	66	b	62	98
ETX	03	03	#	23	35	C	43	67	c	63	99
EOT	04	04	$	24	36	D	44	68	d	64	100
ENQ	05	05	%	25	37	E	45	69	e	65	101
ACK	06	06	&	26	38	F	46	70	f	66	102
BEL	07	07	'	27	39	G	47	71	g	67	103
BS(←)	08	08	(	28	40	H	48	72	h	68	104
HT(→)	09	09	)	29	41	I	49	73	i	69	105
LF(↓)	0A	10	*	2A	42	J	4A	74	j	6A	106
VT(↑)	0B	11	+	2B	43	K	4B	75	k	6B	107
FF (home)	0C	12	,	2C	44	L	4C	76	l	6C	108
CR (return)	0D	13	-	2D	45	M	4D	77	m	6D	109
SO	0E	14	.	2E	46	N	4E	78	n	6E	110
SI	0F	15	/	2F	47	O	4F	79	o	6F	111
DLE	10	16	0	30	48	P	50	80	p	70	112
X-ON	11	17	1	31	49	Q	51	81	q	71	113
~~TAPE~~	12	18	2	32	50	R	52	82	r	72	114
X-OFF	13	19	3	33	51	S	53	83	s	73	115
TAPE	14	20	4	34	52	T	54	84	t	74	116
NAK	15	21	5	35	53	U	55	85	u	75	117
SYN	16	22	6	36	54	V	56	86	v	76	118
ETB	17	23	7	37	55	W	57	87	w	77	119
CAN	18	24	8	38	56	X	58	88	x	78	120
EM	19	25	9	39	57	Y	59	89	y	79	121
SUB	1A	26	:	3A	58	Z	5A	90	z	7A	122
ESC	1B	27	;	3B	59	[	5B	91	{	7B	123
FS	1C	28	<	3C	60	\	5C	92	¦	7C	124
GS	1D	29	=	3D	61	]	5D	93	}	7D	125
RS	1E	30	>	3E	62	^	5E	94	~	7E	126
US	1F	31	?	3F	63	_	5F	95	DEL	7F	127

* **Note on control characters:** The characters in this column are control characters. Most of them do not have designated keys on the keyboard. To obtain them you press and hold the CTRL key, and while it is held, press some other key. The key to use is the one listed in the third column of the table, at a position corresponding to the desired control character. Thus pressing CTRL and E gives the control character ENQ (ASCII code 05, hex).

Example:
CTRL-M (control M) is the same as RETURN.

APPENDIX D: HEKTOR's GRAPHICS CHARACTERS

Table D.1 shows the set of graphics characters available in HEKTOR.

The characters can be obtained in assembly language by means of their ASCII codes, given in the table in denary.

In the BASIC interpreter, the graphics characters can also be obtained by use of their ASCII codes. Single characters can be obtained most conveniently by use of the string function CHR$ (Subsection 7.3.5).

Alternatively, characters can be obtained by keying CTRL-G (see Appendix C for note on control characters) and then keying the corresponding keyboard key, as shown in Table D.1. This mode is most convenient for producing strings of graphics characters.

After keying CTRL-G, only characters corresponding to the upper-case keyboard symbols and letters will be produced. To obtain the characters corresponding to the lower-case keyboard symbols and letters, CTRL-S must be keyed before and after the keystroke (or series of keystrokes).

In the CTRL-G graphics mode, the cursor control keys, →, ←, ↑, ↓ and HOME, and the RETURN key, all produce graphics characters. To revert to the normal keyboard characters and functions, CTRL-G must be keyed again.

Table D.1 HEKTOR graphics characters

128		144		160		176 (Ø)		192 (@)		208 (P)		224		240 (p)	
129		145		161 (!)		177 (1)		193 (A)		209 (Q)		225 (a)	A	241 (q)	
130		146		162 (")		178 (2)		194 (B)		210 (R)		226 (b)	2	242 (r)	
131		147		163 (#)		179 (3)		195 (C)		211 (S)		227 (c)	3	243 (s)	
132		148		164 ($)		180 (4)		196 (D)		212 (T)		228 (d)	4	244	
133		149		165 (%)		181 (5)		197 (E)		213 (U)		229 (e)	5	245	
134		150		166 (&)		182 (6)		198 (F)		214 (V)		230 (f)	6	246	
135		151		167 (')		183 (7)		199 (G)		215 (W)		231 (g)	7	247	
136 (←)		152		168		184 (8)		200 (H)		216 (X)		232 (h)	8	248	
137 (→)		153		169		185 (9)		201 (I)		217 (Y)		233 (i)	9	249	
138 (↓)		154		170 (*)		186 (:)		202 (J)		218 (Z)		234 (j)	10	250	
139 (↑)		155		171 (+)		187 (;)		203 (K)		219 ([)		235 (k)	J	251	
140 (HOME)		156		172 (,)		188 (<)		204 (L)		220 (\)		236 (l)	Q	252	
141 (RET)		157		173 (-)		189 (=)		205 (M)		221 (])		237 (m)	K	253	
142		158		174 (.)		190 (>)		206 (N)		222 (^)		238 (n)		254	
143		159		175 (/)		191 (?)		207 (O)		223 (_)		239 (o)		255	

APPENDIX E: HEKTOR BASIC ERROR MESSAGES

Error code	*Error message*
0101	Line number too large
0174	Not enough space to insert program line
0206	Illegal characters in or after statement
0312	Array subscript too large
0320	Zero array subscript
0364	No such array
0383	Array subscript too large
0445	Out of RAM
0480	Number too large
0488	Number too large
0617	GOTO non-existent line
0624	Line number too large
0639	No comma in DELETE statement
0648	No delimiting number
0651	Line number too large
0695	Line number too large
0773	Parameter to TAB negative or exceeds 256
0779	Illegal zero parameter to TAB function
0784	Parameter to TAB exceeds 63
0789	Use of TAB function to other than TV screen
0949	GOSUB to non-existent line
0989	Invalid symbol in DIM statement
0994	Invalid symbol in DIM statement
1007	Argument to DIM greater than 127
1012	Zero argument to DIM statement
1015	Argument to DIM greater than 127
1041	No RAM left to declare another array
1055	Attempt to redimension array
1081	Return without matching GOSUB
1101	Variable name expected after FOR
1191	TO after FOR missing or misspelt
1194	String variable not allowed in NEXT statement
1197	Variable name expected in NEXT statement
1210	Improper nesting of FOR and GOSUB
1327	THEN expected
1350	Read from file OPEN for OUTPUT
1365	Unable to read this tape
1373	Read a file name instead of data
1387	Variable name expected in INPUT statement
1425	Number too large
1434	Number too large
1440	Characters beyond end of input number
1635	Overflow when adding terms
1678	Overflow when multiplying factors

Error code	*Error message*
1690	Overflow when multiplying factors
1719	Dividing by zero
1731	Overflow in multiply or divide
1746	String variable not allowed here
1757	Number too large
1770	Parenthesis expected
1828	Negative parameter to RND
1831	Zero parameter to RND
1915	Overflow
1937	Parenthesis expected
1974	String too long
2024	No RAM left for string operation requested
2027	Inappropriate use of string variable
2046	Equals sign expected
2113	Invalid characters on end of line
2290	Line number too large
2329	Line number too large or negative
2386	String too long for buffer
2410	String too long for buffer
2430	Equals sign expected
2573	String expression too complex
2675	FOR or GOSUB statements nested too deep
2710	Null string illegal as file name
2751	VERIFY found bad program
2758	No file name on LOAD
2777	Program read is faulty or too big
2792	File number negative or over 255
2800	File unit over 9
2816	Cassette syntax error in OPEN, CLOSE, INPUT or PRINT
2823	No file number in OPEN or CLOSE statement
2825	Trying to open or close unit 0 – always open
2832	Attempt to open a file already open
2885	Null string given for file name in OPEN
2987	Syntax error in OPEN statement
3093	Parameter to string function negative or over 255
3097	Zero numeric parameter to string function
3115	Beyond end of string
3139	Beyond end of string
3161	Comma expected
3194	Beyond end of string
3212	Comma or parenthesis expected
3275	SOUND parameters incorrect
3279	SOUND argument over 255
3282	Zero argument to SOUND function
3323	Print to file OPEN for INPUT

APPENDIX F: 8085 OPERATION CODES

The instruction set for the 8085 microprocessor used in HEKTOR is presented in three forms. Table F.1 gives the hexadecimal operation codes in numeric order, along with the assembly language symbolic equivalent. Table F.2 gives the symbolic forms in alphabetical order, along with the hexadecimal machine code equivalent. Table F.3 gives a summary of the main effects of each instruction type. More detailed description of the effects of individual instructions is given in Subsection 6.4.

Table F.1 Instructions in operation code sequence

OP CODE	MNEMONIC						
00	NOP	40	MOV B,B	80	ADD B	C0	RNZ
01	LXI B,D16	41	MOV B,C	81	ADD C	C1	POP B
02	STAX B	42	MOV B,D	82	ADD D	C2	JNZ A16
03	INX B	43	MOV B,E	83	ADD E	C3	JMP A16
04	INR B	44	MOV B,H	84	ADD H	C4	CNZ A16
05	DCR B	45	MOV B,L	85	ADD L	C5	PUSH B
06	MVI B,D8	46	MOV B,M	86	ADD M	C6	ADI D8
07	RLC	47	MOV B,A	87	ADD A	C7	RST 0
08	-	48	MOV C,B	88	ADC B	C8	RZ
09	DAD B	49	MOV C,C	89	ADC C	C9	RET
0A	LDAX B	4A	MOV C,D	8A	ADC D	CA	JZ A16
0B	DCX B	4B	MOV C,E	8B	ADC E	CB	-
0C	INR C	4C	MOV C,H	8C	ADC H	CC	CZ A16
0D	DCR C	4D	MOV C,L	8D	ADC L	CD	CALL A16
0E	MVI C,D8	4E	MOV C,M	8E	ADC M	CE	ACI D8
0F	RRC	4F	MOV C,A	8F	ADC A	CF	RST 1
10	-	50	MOV D,B	90	SUB B	D0	RNC
11	LXI D,D16	51	MOV D,C	91	SUB C	D1	POP D
12	STAX D	52	MOV D,D	92	SUB D	D2	JNC A16
13	INX D	53	MOV D,E	93	SUB E	D3	OUT A8
14	INR D	54	MOV D,H	94	SUB H	D4	CNC A16
15	DCR D	55	MOV D,L	95	SUB L	D5	PUSH D
16	MVI D,D8	56	MOV D,M	96	SUB M	D6	SUI D8
17	RAL	57	MOV D,A	97	SUB A	D7	RST 2
18	-	58	MOV E,B	98	SBB B	D8	RC
19	DAD D	59	MOV E,C	99	SBB C	D9	-
1A	LDAX D	5A	MOV E,D	9A	SBB D	DA	JC A16
1B	DCX D	5B	MOV E,E	9B	SBB E	DB	IN A8
1C	INR E	5C	MOV E,H	9C	SBB H	DC	CC A16
1D	DCR E	5D	MOV E,L	9D	SBB L	DD	-
1E	MVI E,D8	5E	MOV E,M	9E	SBB M	DE	SBI D8
1F	RAR	5F	MOV E,A	9F	SBB A	DF	RST 3
20	RIM	60	MOV H,B	A0	ANA B	E0	RPO
21	LXI H,D16	61	MOV H,C	A1	ANA C	E1	POP H
22	SHLD A16	62	MOV H,D	A2	ANA D	E2	JPO A16
23	INX H	63	MOV H,E	A3	ANA E	E3	XTHL
24	INR H	64	MOV H,H	A4	ANA H	E4	CPO A16
25	DCR H	65	MOV H,L	A5	ANA L	E5	PUSH H
26	MVI H,D8	66	MOV H,M	A6	ANA M	E6	ANI D8
27	DAA	67	MOV H,A	A7	ANA A	E7	RST 4
28	-	68	MOV L,B	A8	XRA B	E8	RPE
29	DAD H	69	MOV L,C	A9	XRA C	E9	PCHL
2A	LHLD A16	6A	MOV L,D	AA	XRA D	EA	JPE A16
2B	DCX H	6B	MOV L,E	AB	XRA E	EB	XCHG
2C	INR L	6C	MOV L,H	AC	XRA H	EC	CPE A16
2D	DCR L	6D	MOV L,L	AD	XRA L	ED	-
2E	MVI L,D8	6E	MOV L,M	AE	XRA M	EE	XRI D8
2F	CMA	6F	MOV L,A	AF	XRA A	EF	RST 5
30	SIM	70	MOV M,B	B0	ORA B	F0	RP
31	LXI SP,D16	71	MOV M,C	B1	ORA C	F1	POP PSW
32	STA A16	72	MOV M,D	B2	ORA D	F2	JP A16
33	INX SP	73	MOV M,E	B3	ORA E	F3	DI
34	INR M	74	MOV M,H	B4	ORA H	F4	CP A16
35	DCR M	75	MOV M,L	B5	ORA L	F5	PUSH PSW
36	MVI M,D8	76	HLT	B6	ORA M	F6	ORI D8
37	STC	77	MOV M,A	B7	ORA A	F7	RST 6
38	-	78	MOV A,B	B8	CMP B	F8	RM
39	DAD SP	79	MOV A,C	B9	CMP C	F9	SPHL
3A	LDA A16	7A	MOV A,D	BA	CMP D	FA	JM A16
3B	DCX SP	7B	MOV A,E	BB	CMP E	FB	EI
3C	INR A	7C	MOV A,H	BC	CMP H	FC	CM A16
3D	DCR A	7D	MOV A,L	BD	CMP L	FD	-
3E	MVI A,D8	7E	MOV A,M	BE	CMP M	FE	CPI D8
3F	CMC	7F	MOV A,A	BF	CMP A	FF	RST 7

Note: D8 = 8-bit data
A8 = 8-bit I/O address
D16 = 16-bit data
A16 = 16-bit memory address

Table F.2 Instructions in alphabetical order

MNEMONIC	OP CODE	TIME
ACI D8	CE	7
ADC A	8F	4
ADC B	88	4
ADC C	89	4
ADC D	8A	4
ADC E	8B	4
ADC H	8C	4
ADC L	8D	4
ADC M	8E	7
ADD A	87	4
ADD B	80	4
ADD C	81	4
ADD D	82	4
ADD E	83	4
ADD H	84	4
ADD L	85	4
ADD M	86	7
ADI D8	C6	7
ANA A	A7	4
ANA B	A0	4
ANA C	A1	4
ANA D	A2	4
ANA E	A3	4
ANA H	A4	4
ANA L	A5	4
ANA M	A6	7
ANI D8	E6	7
CALL A16	CD	18
CC A16	DC	9/18
CM A16	FC	9/18
CMA	2F	4
CMC	3F	4
CMP A	BF	4
CMP B	B8	4
CMP C	B9	4
CMP D	BA	4
CMP E	BB	4
CMP H	BC	4
CMP L	BD	4
CMP M	BE	7
CNC A16	D4	9/18
CNZ A16	C4	9/18
CP A16	F4	9/18
CPE A16	EC	9/18
CPI D8	FE	7
CPO A16	E4	9/18
CZ A16	CC	9/18
DAA	27	4
DAD B	09	10
DAD D	19	10
DAD H	29	10
DAD SP	39	10
DCR A	3D	4
DCR B	05	4
DCR C	0D	4
DCR D	15	4
DCR E	1D	4
DCR H	25	4
DCR L	2D	4
DCR M	35	10
DCX B	0B	6
DCX D	1B	6
DCX H	2B	6
DCX SP	3B	6
DI	F3	4
EI	FB	4
HLT	76	5
IN A8	DB	10
INR A	3C	4
INR B	04	4
INR C	0C	4
INR D	14	4
INR E	1C	4
INR H	24	4
INR L	2C	4
INR M	34	10
INX B	03	6
INX D	13	6
INX H	23	6
INX SP	33	6
JC A16	DA	7/10
JM A16	FA	7/10
JMP A16	C3	10
JNC A16	D2	7/10
JNZ A16	C2	7/10
JP A16	F2	7/10
JPE A16	EA	7/10
JPO A16	E2	7/10
JZ A16	CA	7/10
LDA A16	3A	13
LDAX B	0A	7
LDAX D	1A	7
LHLD A16	2A	16
LXI B,D16	01	10
LXI D,D16	11	10
LXI H,D16	21	10
LXI SP,D16	31	10
MOV A,A	7F	4
MOV A,B	78	4
MOV A,C	79	4
MOV A,D	7A	4
MOV A,E	7B	4
MOV A,H	7C	4
MOV A,L	7D	4
MOV A,M	7E	7
MOV B,A	47	4
MOV B,B	40	4
MOV B,C	41	4
MOV B,D	42	4
MOV B,E	43	4
MOV B,H	44	4
MOV B,L	45	4
MOV B,M	46	7
MOV C,A	4F	4
MOV C,B	48	4
MOV C,C	49	4
MOV C,D	4A	4
MOV C,E	4B	4
MOV C,H	4C	4
MOV C,L	4D	4
MOV C,M	4E	7
MOV D,A	57	4
MOV D,B	50	4
MOV D,C	51	4
MOV D,D	52	4
MOV D,E	53	4
MOV D,H	54	4
MOV D,L	55	4
MOV D,M	56	7
MOV E,A	5F	4
MOV E,B	58	4
MOV E,C	59	4
MOV E,D	5A	4
MOV E,E	5B	4
MOV E,H	5C	4
MOV E,L	5D	4
MOV E,M	5E	7
MOV H,A	67	4
MOV H,B	60	4
MOV H,C	61	4
MOV H,D	62	4
MOV H,E	63	4
MOV H,H	64	4
MOV H,L	65	4
MOV H,M	66	7
MOV L,A	6F	4
MOV L,B	68	4
MOV L,C	69	4
MOV L,D	6A	4
MOV L,E	6B	4
MOV L,H	6C	4
MOV L,L	6D	4
MOV L,M	6E	7
MOV M,A	77	7
MOV M,B	70	7
MOV M,C	71	7
MOV M,D	72	7
MOV M,E	73	7
MOV M,H	74	7
MOV M,L	75	7
MVI A,D8	3E	7
MVI B,D8	06	7
MVI C,D8	0E	7
MVI D,D8	16	7
MVI E,D8	1E	7
MVI H,D8	26	7
MVI L,D8	2E	7
MVI M,D8	36	10
NOP	00	4
ORA A	B7	4
ORA B	B0	4
ORA C	B1	4
ORA D	B2	4
ORA E	B3	4
ORA H	B4	4
ORA L	B5	4
ORA M	B6	7
ORI D8	F6	7
OUT A8	D3	10
PCHL	E9	6
POP B	C1	10
POP D	D1	10
POP H	E1	10
POP PSW	F1	10
PUSH B	C5	12
PUSH D	D5	12
PUSH H	E5	12
PUSH PSW	F5	12
RAL	17	4
RAR	1F	4
RC	D8	6/12
RET	C9	10
RIM	20	4
RLC	07	4
RM	F8	6/12
RNC	D0	6/12
RNZ	C0	6/12
RP	F0	6/12
RPE	E8	6/12
RPO	E0	6/12
RRC	0F	4
RST 0	C7	12
RST 1	CF	12
RST 2	D7	12
RST 3	DF	12
RST 4	E7	12
RST 5	EF	12
RST 6	F7	12
RST 7	FF	12
RZ	C8	6/12
SBB A	9F	4
SBB B	98	4
SBB C	99	4
SBB D	9A	4
SBB E	9B	4
SBB H	9C	4
SBB L	9D	4
SBB M	9E	7
SBI D8	DE	7
SHLD A16	22	16
SIM	30	4
SPHL	F9	6
STA A16	32	13
STAX B	02	7
STAX D	12	7
STC	37	4
SUB A	97	4
SUB B	90	4
SUB C	91	4
SUB D	92	4
SUB E	93	4
SUB H	94	4
SUB L	95	4
SUB M	96	7
SUI D8	D6	7
XCHG	EB	4
XRA A	AF	4
XRA B	A8	4
XRA C	A9	4
XRA D	AA	4
XRA E	AB	4
XRA H	AC	4
XRA L	AD	4
XRA M	AE	7
XRI D8	EE	7
XTHL	E3	16

Note: D8 = 8-bit data
A8 = 8-bit I/O address
D16 = 16-bit data
A16 = 16-bit memory address

TIME = execution time in machine cycles; where two times given, longer time is if action is performed

Table F.3 Summary of 8085 instruction types

Mnemonics and operand types used in Table F.3

D8	8-bit operand (data) value
D16	16-bit operand (data) value
M	Implied address (specified by HL register contents)
A16	16-bit memory address
A8	8-bit I/O address
A3	One of 8 restart addresses
R8	8-bit register (A, B, C, D, E, H or L)

Symbols for 16-bit register pairs

	AF	BC	DE	HL	SP
R16a		B	D	H	SP
R16b	PSW	B	D	H	
R16c		B	D		

Data Copy Group

MOV	R8,R8	Copy data between register/memory and register/memory
MOV	M,R8	
MOV	R8,M	
MVI	R8,D8	Copy operand data to register/memory
MVI	M,D8	
LDA	A16	Copy data from memory to A register
LDAX	R16c	
STA	A16	Copy data from A register to memory
STAX	R16c	
LHLD	A16	Copy data from memory to HL register pair
SHLD	A16	Copy data from HL register pair to memory
LXI	R16a,D16	Copy operand data to register pair
XCHG		Exchange data between HL and DE register pairs
XTHL		Exchange data between HL register pair and top of stack

Arithmetic Group

ADD	R8	Add register/memory contents to A register contents
ADD	M	
ADI	D8	Add operand data to A register contents
ADC	R8	Add register/memory and Carry contents to A register contents
ADC	M	
ACI	D8	Add operand data and Carry contents to A register contents
SUB	R8	Subtract register/memory contents from A register contents
SUB	M	
SUI	D8	Subtract operand data from A register contents
SBB	R8	Subtract register/memory and Carry contents from A register contents
SBB	M	
SBI	D8	Subtract operand data and Carry contents from A register contents
INR	R8	Increment register/memory contents by 1
INR	M	
DCR	R8	Decrement register/memory contents by 1
DCR	M	
INX	R16a	Increment register pair contents by 1
DCX	R16a	Decrement register pair contents by 1
DAD	R16a	Add register pair contents to HL register pair contents
DAA		Adjust A register contents for BCD result following addition

Logical Group

ANA	R8	Logical AND register/memory contents with A register contents
ANA	M	
ANI	D8	Logical AND operand data with A register contents
ORA	R8	Logical (inclusive) OR register/memory contents with A register contents
ORA	M	
ORI	D8	Logical (inclusive) OR operand data with A register contents
XRA	R8	Logical exclusive OR register/memory contents with A register contents
XRA	M	
XRI	D8	Logical exclusive OR operand data with A register contents
CMP	R8	Compare register/memory contents with A register contents
CMP	M	
CPI	D8	Compare operand data with A register contents
RLC		Rotate A register contents left, and into Carry
RRC		Rotate A register contents right, and into Carry
RAL		Rotate A register and Carry contents left
RAR		Rotate A register and Carry contents right
CMA		Complement A register contents
CMC		Complement Carry contents
STC		Set Carry contents to 1

Program Sequence Control Group

Jump if:	Call if:	Return if:	Condition is:
JC	CC	RC	Carry (Carry=1)
JNC	CNC	RNC	No Carry (Carry=0)
JZ	CZ	RZ	Zero (Zero=1)
JNZ	CNZ	RNZ	Not Zero (Zero=0)
JP	CP	RP	Plus (Sign=0)
JM	CM	RM	Minus (Sign=1)
JPE	CPE	RPE	Parity even (Parity=1)
JPO	CPO	RPO	Parity odd (Parity=0)
JMP	CALL	RET	Unconditionally
PCHL			Copy data from HL register pair to Program Counter
RST A3			Call routine at restart address

A16 (Jump if: JC … JMP; Call if: CC … CALL)

Stack Operation Group

PUSH	R16b	Push register pair contents onto the stack
POP	R16b	Pop top of stack data into register pair
SPHL		Copy data from HL register pair to Stack Pointer

Input/Output Group

IN	A8	Copy data from I/O device to A register
OUT	A8	Copy data from A register to I/O device

Machine Control Group

EI	Enable interrupt system
DI	Disable interrupt system
RIM	Copy Interrupt Status data to A register
SIM	Copy A register contents to Interrupt Control
HLT	Halt processor
NOP	No operation

APPENDIX G: HEKTOR COMMAND LISTS

All commands must have terminators. For more details, see the appropriate sections.

Monitor Commands

(Addresses are denoted by *addr, addr1*, etc)

B — removes a Break-point

B*addr* — sets a Break-point at *addr*

C*addr1,addr2,addr3* — Copies data between *addr1* and *addr2*, to locations starting at *addr3*

E — execute Editor

F*addr1,addr2,data* — Fill memory from *addr1* to *addr2* with *data*

G — (Go to) execute from saved PC address

G*addr* — (Go to) execute from *addr*

H — (High-level language) execute BASIC interpreter

L — Load from tape

M*addr* — (Memory) examine and modify at *addr*

P*addr1,addr2* — (Print) display memory from *addr1* to *addr2*

Q*addr1,addr2,data* — (Query) search memory from *addr1* to *addr2* for *data*

R — (Rewind) connect power to cassette recorder

S*addr1,addr2* — Save data from *addr1* to *addr2* on tape

T — Test subsystems

V — Verify recorded data

W — (Warm start) Re-enter BASIC interpreter

X — eXamine and modify saved status

1 — single-step execution from saved PC address

1*addr* — single-step execution from *addr*

Editor Commands

A *option1 option2 etc* — execute Assembler using *option(s)*

D*line* — Delete *line*

D*line1,line2* — Delete from *line1* to *line2*

E*line* — Edit characters in *line*

I*line* — Insert lines after *line*

K — (Kill) delete all lines in buffer

L — (Load) append lines from tape to buffer

M — return to Monitor

P*line* — (Print) display *line*

P*line1,line2* — (Print) display from *line1* to *line2*

Q*line1,line2 character(s)* — (Query) search from *line1* to *line2* for *character(s)*

R — (Rewind) connect power to cassette recorder

S*line1,line2* — Save from *line1* to *line2* on tape

V — Verify recorded lines

Assembler Options

(in editor's A command)

L — List program

M — return to Monitor after loading

S — display Symbol table

T — save on Tape

W — Wait if error found

BASIC keywords

Command Mode

RUN — initialize variables and execute program

GOTO n — execute program from line n

NEW — delete all program lines

LIST — list all program lines

LIST *line 1* — list program lines starting from *line 1*

DELETE *line 1, line 2* — delete from *line 1* to *line 2*

SAVE *"name"* — save program on tape in file labelled *name*

LOAD *"name"* — load program labelled *name* from tape

VERIFY *"name"* — verify program labelled *name* from tape

REWIND — connect power to cassette recorder

General Keywords

STOP — stops execution

REM (or !) — remarks or comments, not to be executed

Variables

A, B, ... , Z — numeric variables

A(*index*), B(*index*), ... , Z(*index*) — arrays

A$, B$, ... , Z$ — string variables

DIM — dimension array

LET *variable = expression* — assign the value of *expression* to *variable*

System Variables

(R = read only; R/W = read/write)

BAUD — bit rate and parity of serial line (R/W)

COL — current column of display (R)

KSTAT — ASCII value of key pressed (R)

LED — value of peripheral board LEDs (R/W)

PW — print width allocated for numbers (R/W)

RANDOM – variable used by RND function (R/W)

ROW — current row of display (R)

SIZE — remaining unused memory (R)

STAT — status of peripheral board buttons (R)

SW — value of peripheral board switches (R)

TIME — value of real-time clock (R/W)

Operators

+,−,*,/ — numeric operators

>,<,>=,<=,=,< > — relational operators

& — string operator

Functions

(All arguments, while shown as X, X$, etc., can be variables, constants or expressions.)

ABS(X) — absolute value of X

ASC(X$) — ASCII code of first character of X$

CHR$(X) — character whose character code is X

LEFT$(X$,Y) — left-most Y characters of X$

LEN(X$) — length of string X$

MID$(X$,Y,Z) — Z characters from X$ starting with the Yth

RIGHT$(X$,Y) — right-most Y characters of X$

RND(X) — pseudo-random number between 1 and X

STR$(X) — string representation of number X

VAL(X$) — numeric representation of string X$

Flow of Control

GOTO n — execute line n next

GOSUB n — execute line n next and save return address

FOR *variable* = (a) TO (b) STEP (c)
... *lines* ...
NEXT *variable* — loop in which *lines* are repeated while *variable* varies from (a) to (b) in steps of (c)

IF *condition* THEN *statement* — if *condition* is true, then execute *statement*

RETURN — execute statement following GOSUB

Input/Output

CLOSE # n; — close channel number n

INPUT *variable 1, variable 2,* ... — read values for *variable 1, variable 2,* etc. from keyboard

INPUT # n; *variable 1, variable 2,* ... — read values for *variable 1, variable 2,* etc. from channel n.

OPEN # n; (CR or SL), (I or O), *name* — open channel n to (CR or SL) for (I or O) labelled with *name*

PRINT *expression 1, expression 2,* ... — evaluate *expression 1, expression 2,* etc. and display them on screen

PRINT # n; *expression 1, expression 2,* ... — evaluate *expression 1, expression 2,* etc. and send them to channel n

TAB (n) — (in PRINT statement) move cursor to column n

SOUND (X,Y) — produce sound in loudspeaker of pitch determined by period = X × 250 μs, and length = Y × 0.01 s

INDEX

ABS 62
accumulator 11
ACI 46
ADC 46
ADD 46
address bus 13, 68
address/data bus 68
addressing modes 37
ADI 46
ALU 11
ANA 46
ANI 47
application programs 15
argument of command 24
argument list
  editor 33
  monitor 26
arithmetic/logic unit 11
arrays 59
ASC 62
ASCII code 79, 89
assembler program 15, 31, 37, 52
  entry command from editor (A) 34
  error messages 53–54
  options 33, 54
  passes 53
  program listing 53
  symbol table 42, 52
assembly language 15
  directives 38–39
  instructions 38
  programming 37
  statements 37–39
  symbols 37, 52

BASIC 55
  arrays 59
  assignments 60
  character control 57
  command mode 55
  concatenation 61
  control characters 57
  dimension statements 60
  entry command from monitor (H) 27
  error messages 91
  execution of program 56
  expressions 61
  flow-of-control statements 62
  format control 57
  functions 61
  graphics characters 57, 90
  input/output instructions 63
  interpreter 15
  keywords 55
  line numbers 58
  lines 59
  multiple statements 59
  numeric expressions 61
  numeric functions 62
  numeric operators 61
  numeric variables 59
  operators 61
  program structure 58
  prompt (*) 55
  read-only system variables 60
  read/write system variables 60
  relational operators 61
  running a program 56
  spaces in lines 59
  statements 59
  string expressions 61
  string operators 61
  string variables 59, 60
  syntax 59
  system variables 60
  variables 59
  writing programs 58
'Bad-label' error message 53
'Bad instr.' error message 53
'Bad number' error message 54
'Bad register' error message 54
BAUD 60
baud rate/parity parameters 82
BCD arithmetic 83
binary codes 13
binary voltages 12
bit 13
break-point
  handler 77
  command (B) 26
BREAK vector 76
bus 9, 10, 68
bus connections 74
byte 13

CALL 47
call instruction 13
cassette recorder 9, 10, 16
  assembler options 54
  BASIC instructions 64
  cables 16
  connection 19
  editor commands 35
  handler 80–81
  interface 72–73
  monitor commands 28–29
  problems 21
  tapes 16
  tests 20
CC 47
CHR$ 62
clock 12
clock frequency 68
CLOSE # 64
CM 47
CMA 47
CMC 47
CMP 47
CNC 47
CNZ 47
COL 60
command acceptance
  editor 31
  monitor 25
command characters
  editor 33
  monitor 26
command execution
  editor 31
  monitor 25, 26
command lists 96–97
comment field
  assembler directives 38
  assembler instructions 38
comment lines
  assembly language 37
  BASIC 59
component checklist 16
connecting cables 16
connecting HEKTOR components 17–19

control characters 57
control signals 12
control unit 11, 12
copy command (C) 26
COUTV vector 79
CP 47
CPE 47
CPI 47
CPO 48
CRL 81
CRS 81
cursor 79
CZ 48

DAA 48
DAD 48
data 9
damage to HEKTOR 21
data-byte.pseudo-operation 38
data-word pseudo-operation 38
DB pseudo-operation 38
DCR 48
DCX 48
debugging 57
DELETE 56
delete command (D) 34
DI 48, 76
DIM 59
dimension statements 60
display routines 79
DMUL 83
DSUB 83
DW pseudo-operation 38

edge connector 10, 22, 68
edit line command (E) 34
editor program 15, 26, 31
  behaviour 32–33
  command format 33–34
  entry command from monitor (E) 27
  line numbers 33
  prompt (#) 31
  structure 31–32
EI 48, 76
END 39
end-of-program pseudo-operation (END) 39
equate pseudo-operation (EQU) 38
EQU directive 38, 53
error messages 21, 53–54, 57, 91
even parity 46
examine register command (X) 29
execution command (G) 27
execution errors in BASIC 57

faults in HEKTOR 21, 22
ferric cassettes 16
fetch/execute sequence (cycle) 12, 42
fill command (F) 27
flags 11, 46
  auxiliary carry 46
  carry 46
  parity 46
  sign 46
  zero 46
flag register 11, 46
format control 57
FOR ... NEXT loop 63
'Forward ref' error message 54
fuse 16

graphics characters 57, 90
go command (G) 27
GOSUB 62
GOTO 57, 62

hardware malfunctions 21
handling precautions 18
heat dissipation 18
HEKTOR system hardware 68
HEKTOR system software 75
hexadecimal numbers 24
  conversion 88
high-level language command (H) 27
HLT 48

IF ... THEN statement 63
immediate execution statement 55, 57
implementation errors in BASIC 57
IN 48
INPUT 64
INPUT # 65
input/output
  handler routines 75
  map 15
  parameters 75
  structure 78
INR 48
insert command (I) 34
instruction fetch 11
instruction set 11, 46
interface 9
interference 21
internal bus 11
interrupt 12
  handler, editor 31, 77
  handler, monitor 25, 77
  parameters 75
  register 76
  signal 12
  sources 76
  structure 75
  vectors 75, 78
INX 48

JC 48
JM 48
JMP 48
JNC 49
JNZ 49
JP 49
JPE 49
JPO 49
jump instruction 12
JZ 49

K (= 1024) 15
KBUF 80
KDEL 79
keyboard 9
  buffer 75
  faults 22
  handlers 79
  interface 70
  routines 79
  tests 20
KFLAG 79
KGRAF 80
kill command (K) 35
KLUC 80
KR 80
KSTAT 60, 79

label field
  assembler directives 38
  assembler instructions 38
LDA 49

LDAX 49
LED (light-emitting diode) 19
LEFT$ 62
LEN 62
LET 60
LHLD 49
line-edit command (E) 34
lines of text 31
line numbers in editor 33
LIST 56
LOAD 58
load-from-tape command (L)
  editor 35
  monitor 28
loading from tape in BASIC 58
logical operations 11
LXI 49

machine code 37
machine-code buffer 53
machine instructions 12, 13
malfunctions 21
mains 16
  fuse 16
  plug 16, 18
  supply 16
memory 9
  address 13
  faults 21
  locations 11
  map 15, 25, 86–87
  modify command (M) 28
  subsystem 10, 69
  tests 20
microcomputer board 9, 10
microprocessor (8085) 41, 68
MID$ 62
'Missing opnd.' error message 54
MON 55
monitor program 15, 24
  behaviour 25–26
  commands 26–30
  command format 26–27
  entry command from editor (M) 35
  prompt (<) 25
  structure 25
  utility subroutine 25, 31
MOV 49
multiplexed address/data bus 68
musical instructions 83
MVI 49

nested subroutines 14
NEW 56
NEXT 63
NOP 49
numeric variables 59

odd parity 46
opcode field 38
opcode specification in assembler 42
OPEN # 64
operand 13
operand address specification in assembler 40
operand field 39
  assembler directives 38
  assembler instructions 38
operation code (opcode) 13, 92
ORA 49
ORG 39
ORI 50
origin pseudo-operation (ORG) 39
OUT 50

parts checklist 16
PCHL 50
PC register 11, 13, 42
peripherals 9
peripheral board 16, 18
  connection 19
  faults 22
  tests 21
plug (mains) 18
POP 50
popping (from stack) 14
POP PSW 50
power, loss of 21
power supply
  circuitry 74
  unit 10
PRCOL 79
PRERR 79
PRINT 63
PRINT # 64
print command (P)
  editor 35
  monitor 28
printed-circuit board 9
printer connections 74
PRMES 79
PRNL 79
processing unit (processor) 9, 10
  structure 10
prompts
  BASIC (*) 55
  editor (#) 31, 32
  monitor (>) 20
programs 9
  listing 53
  stopping 57
  writing 58
program counter 11, 13, 42
PRSP 79
pseudo-operations 38–39
pseudo-opcode field 38
PUSH 50
pushing (onto stack) 14
PUSH PSW 50
PW 61

query command (Q)
  editor 35
  monitor 28

RAL 50
RAM 15, 24, 25
  error messages 21
  memory 32
  test 20
  usage 76, 87
RANDOM 61
RAR 50
RC 50
read-only memory (ROM) 15, 24
read-write memory (RAM) 15, 24, 25
registers 11
  array 11
  examine/modify command (X) 29
  specification in assembler 41
REM 59
'Rep. label' error message 54
reset
  control line 15
  signal 12
  switch 12
RET 50
RETURN 62

REWIND 58
rewind command (R)
editor 35
monitor 29
RIGHT$ 62
RIM 50, 76
RLC 51
RM 51
RNC 51
RND 62
RNZ 51
ROM 15, 24
error messages 21
entry points 87
ROW 60
RP 51
RPE 51
RPO 51
RRC 51
RST 51
RST5.5 interrupt 77
RST6.5 interrupt 77
RST7.5 interrupt 77
RUN 56
RZ 51

SAVE 58
save-on-tape command (S)
editor 35
monitor 29
saved status buffer 75, 78
saved status RAM 25
saving on tape in BASIC 58
SBB 51
SBI 51
SCROLL 60
self-testing procedure 16, 19–21
serial-line
connections 74
handlers 82
interface 10, 33, 73
setting-up HEKTOR 16
SHLD 51
SIM 51, 76, 77
SIN 82
single-step command (1) 30
SIZE 60
SOBAUD 82
SOUND 64
source lines in assembler 37
SOPAR 82
SORT routine 84
SOUT 82
SPHL 52
SSET 82
SSTOP 82
STA 52
stack 13, 14, 75
RAM 25
stack pointer 14, 25, 45
overflow 75
STAT 60
STAX 52
STC 52
STOP 57, 59
STR$ 62
string variables 60
study guide 6
SUB 52
subroutines 13
SUI 52
SW 60
switch-on tests 20
symbol table 42, 52
syntax errors 57
system hardware 68
system parameter RAM 25
system RAM 32, 75, 87
system software 15, 75
system variables 60

TAB 64
temporary register 11, 12
terminator
editor 33
monitor 26
test command (T) 29
text buffer 31, 32
text lines 31
THEN 63
TIME 61
timer interrupts 77
top of stack 14
transformer 18
TRAP interrupt 77
tune machine 82
TV 9, 16
cable 16, 18
connection 18
display problems 21
handlers 78
interface 10, 70, 72
interface test 20
monitor set 18, 19
output routines 78
types of set 16
TVT 78
typing in program lines 56

'Undef. symbol' error message 54
unsigned integer numbers 13
utility subroutines (monitor) 25
user program 15
user RAM test 20

VAL 62
vectors 75–77
input/output handler 75
interrupt handler 77
VERIFY 58
verify tape command
editor 35
monitor 29
video socket 19

warm start (BASIC) 29
word 13
word length 13

XCHG 52
XRA 52
XRI 52
XTHL 52

& concatenation in BASIC 61
# editor prompt 31
# in BASIC instructions 64
. in editor commands 32
. in BASIC instructions 55
* BASIC prompt 55
; in assembly language 37
> monitor prompt 25
/ in editor commands 32
\ in assembler options 54

>, >=, =, <, <=, <>	relational operators in BASIC	61
1 single-step command in monitor		30
8085 microprocessor		41, 68
8085 instruction types		43
8155 RAM/IO device		70, 75

TM222 The digital computer

1 Computers and data

2 The structure of the digital computer

3 Working with HEKTOR: Introduction to the microcomputer

4 Working with HEKTOR: The 8085 microprocessor

5 Working with HEKTOR: Assembly language processing

6 Introduction to software development

7/8 Principles of input/output operations

9/10 Working with HEKTOR: Input/output programming

11 Introduction to high-level languages

12 Working with HEKTOR. BASIC programming

13 Project

14 The PDP 11 computer systems

15 Operating systems

16 File and case studies

TM222 HEKTOR User Manual

The Open University Press

0 335 17125 7