Introducing Z-80 Assembly Language Programming

Ian R. Sinclair

Introducing Z-80 Assembly Language Programming

Ian R. Sinclair

Newnes Technical Books

Newnes Technical Books
is an imprint of the Butterworth Group
which has principal offices in
London, Boston, Durban, Singapore, Sydney, Toronto, Wellington

First published 1983

© **Butterworth & Co. (Publishers) Ltd, 1983**
Borough Green, Sevenoaks, Kent TN15 8PH, England

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, including photocopying and recording, without the written permission of the copyright holder, application for which should be addressed to the Publishers. Such written permission must also be obtained before any part of this publication is stored in a retrieval system of any nature.

This book is sold subject to the Standard Conditions of Sale of Net Books and may not be re-sold in the UK below the net price given by the Publishers in their current price list.

British Library Cataloguing in Publication Data

Sinclair, Ian R.
Introducing Z-80 assembly language programming.
1. Assembly language (Computer program language)
2. INTEL Z80 (Computer)
I. Title
001.64'25 QA76.73.A8

ISBN 0-408-01338-9

Typeset by Tunbridge Wells Typesetting Services Ltd
Printed in England by Whitstable Litho Ltd., Whitstable, Kent

Preface

There has never been any shortage of information on any of the major microprocessor chips, but most of the books that have been previously available on the 6502 and the Z-80 have been rather specialised. Specialism has taken two forms, of which one is explaining assembly language with a view to the book being used as a reference work by writers of interpreters and compilers. The other option is a book which is specific to the use of one machine, such as the TRS-80 or the ZX-81. This book is designed as an introduction to assembly language programming of the Z-80, assuming only that the reader has some experience of using a language such as BASIC, and enough background knowledge of microprocessors to understand words such as 'address' and 'data'. Some reference to specific hardware is needed, because a Z-80 on its own is of no use and most writers of assembly language will be writing for their own computer, but every care has been taken to avoid giving details that apply to one machine in such a form that the reader would not realise that these details applied to that one machine.

The Z-80 has now been manufactured for a considerable time (several years; a long time in IC terms), and its use is by now very well established. As a result, no book can possibly quote examples of wholly original subroutines, and I would not claim any originality for any of the examples shown here. Most of them have, however, been taken from my own assembly language programs, written either for the TRS-80 or on MENTA, and the few that are not of my own making and for which I know a source have been acknowledged.

Unlike many authors of books on Z-80 assembly language programming, I have laid very little emphasis on arithmetical routines. These are of little interest to the beginner, and generally concern only the writer of a compiler or interpreter – who is not likely to be reading this book. For that reason also, I have emphasised the other aspects of Z-80 programming which are more likely to be used by the hobby user or the user interested in machine control. I have avoided long examples as far as possible, because it is much more useful for the newcomer to assembly language to design his/her own program than to wade through anyone else's work. Once general methods are known, then practice, using this book as a guide, is the best path to achievement in assembly language skills.

I am grateful to a large number of people who have contributed to this book, particularly to Phil Chapman of Newnes Technical Books for the initial idea and continual encouragement, and to Barry Savage of Dataman Designs, who very kindly lent me the MENTA assembler unit to keep in touch with Z-80 assembly language when I replaced my TRS-80 with a BBC machine. I must also mention the tolerance of my wife, faced with a room full of buzzing printers, and the cheerful co-operation of countless members of the TRS-80 Users' Group, particularly Laurie Shields.

Ian Sinclair

此为试读,需要完整PDF请访问: www.ertongbook.com

Contents

1

Why and how

Any system that uses a microprocessor, from a simple controller to a large computer, will require the microprocessor to be controlled by a program, because the action of the microprocessor is completely governed by a program. A microprocessor is programmed by applying sets of electrical signals to it, and each different set of signals can be represented by a number, so that a microprocessor program can be written down as a series of numbers. This set of numbers is called 'machine code' or 'object code', and each number represents a set of signals that will be applied to the microprocessor.

Computers are not generally programmed in this way by the users. Using programs that consist of strings of numbers is not the simplest way of programming, and machine code is referred to as a 'low-level' language. It is, in fact, very close to the most primitive method imaginable, which is to control the action of the microprocessor by the settings of switches. Users and programmers of computers need to be able to ignore the details of what the microprocessor is doing, and concentrate on designing programs to solve specific problems, so that 'high-level' languages such as COBOL, FORTRAN, ALGOL, BASIC and Pascal have been devised. These languages use 'keywords', each of which corresponds to a long program in machine code, to put together what would be a very long and difficult program if it were written directly in machine code. When the language is interpreted, meaning that each keyword (such as PRINT, INPUT, GOTO) is dealt with by the computer in turn as the program runs, this causes the program to run much slower than if the program is compiled. Compiling means that the entire program is turned into machine code before the program is run, so that the conversion is done once and once only.

There are two particularly important reasons for using machine code. One is speed. When a computer uses an interpreted high-level language, the speed of interpretation may be such that actions like the animation of graphics or the sorting of string arrays take place much too slowly because of the interpretation process. For example, a loop such as:

```
10 FOR n = 1 TO 10
20 PRINT a
30 NEXT n
```

will require the interpretation of the keyword PRINT to be carried out ten times, and the address of variable a to be found ten times. On the other hand, a compiler would collect the PRINT and variable-finder routines, put the address of the variable into the PRINT routine, and create a loop in machine code which would carry out the action (not the creation of code, followed by part of the action) ten times. When no compiler exists for a machine, which is true of many small computers (though not of the TRS-80 or Video Genie, which can use the ACCEL compiler) this speed-up action must be carried out by writing the program directly in machine code. If the machine code routines of the interpreter exist in ROM, then it may be possible to make use of them, but the process of interpretation, finding the routine which corresponds to the word PRINT, for example, is cut out, which means that the time-consuming part of the process is eliminated.

The other reason for resorting to machine code is that any high level language is restricted by its keywords. The keywords of a language form a type of menu from which the programmer can choose, but inevitably this menu is restricted. There will be actions which are either impossible or which can only be carried out in a very clumsy way using the keywords. The most important actions of this type are inputs and outputs — the computer is generally restricted to a few standard methods, and when experimental interfacing methods are used, there will very often be no keyword which corresponds to the action that is required. It is rather unusual, to say the least, to find that computer X can read cassettes that were recorded by computer Y, and yet the methods of recording are essentially similar; only the machine code routines differ.

For every part of the system that is under the control of the mircroprocessor, however, control by machine code is possible. This aspect of machine code has assumed less importance in recent years because more recent designs of computers have paid more attention to inputs and outputs, and the requirement for specialised machine code routines is less than it once was. Nevertheless, the use of robot arms, modems, networks and other forms of interfacing will always demand some use of machine code to ensure correct operation.

The Z-80 system

The Z-80 is the most widely used 8-bit microprocessor. Eight-bit in this sense means that program or data signals sent to the microprocessor use eight voltages applied to eight corresponding pins on the body of the microprocessor. The total number of pins on the body is kept to a reasonable number (40) by using these eight pins, the data pins, both for

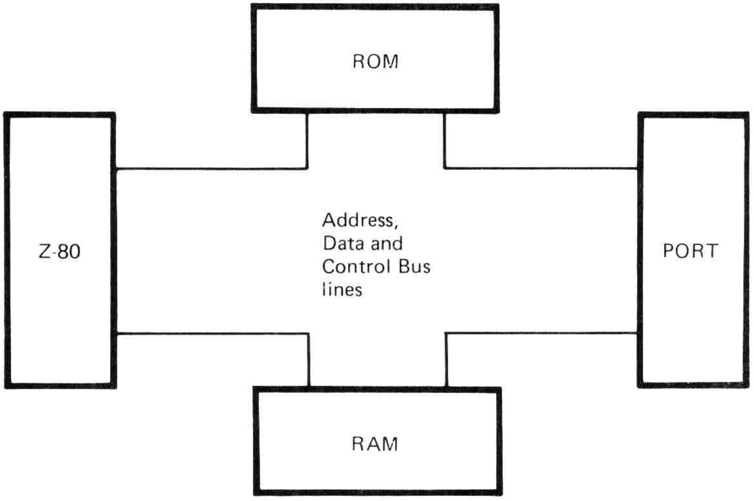

Fig. 1.1. A block diagram for any Z-80 microprocessor system

inputs and for outputs. This is possible because the microprocessor deals with actions in sequence, one by one, so that simultaneous input and output is impossible in any case.

Figure 1.1 shows a generalised block diagram of a Z-80 microprocessor system. The blocks marked ROM and RAM are memory. The ROM will contain at least the essential input and output programs that enable the Z-80 to be further programmed, and the RAM will be used to store programs or data which will be lost when the power is switched off. For most of our purposes, we shall be concerned with RAM into which the number codes that make up a machine-code program are placed, but the ROM is even more important because without some programs present in ROM it would be impossible to program the RAM. The routines in the ROM will therefore, at the very least, provide for input from the keyboard and output to the video screen, so that we can enter items and also see the effect. These inputs and outputs are carried out by passing signals through the block marked 'Port'. The port, which may be one single chip on the board or a number of connected chips, provides for connections, isolation and timing. The connections will be inputs and outputs, and these will have to be isolated from the microprocessor itself because they can be dealt with only at certain times, fixed by the program that controls the inputs and outputs. A port usually provides for storing one set of signals until they can be used either by the microprocessor or by the other systems (peripherals) that are attached to the port.

All the sections of the microprocessor system are connected by lines which are grouped as buses. The significance of a bus is that a large number of chips are made to share one common set of interconnections. The difference between independent connections and buses is illustrated in Fig. 1.2. A bus allows signals to be sent between any two units that are connected to the bus by using a common path, rather than by having a separate path for each possible signal route.

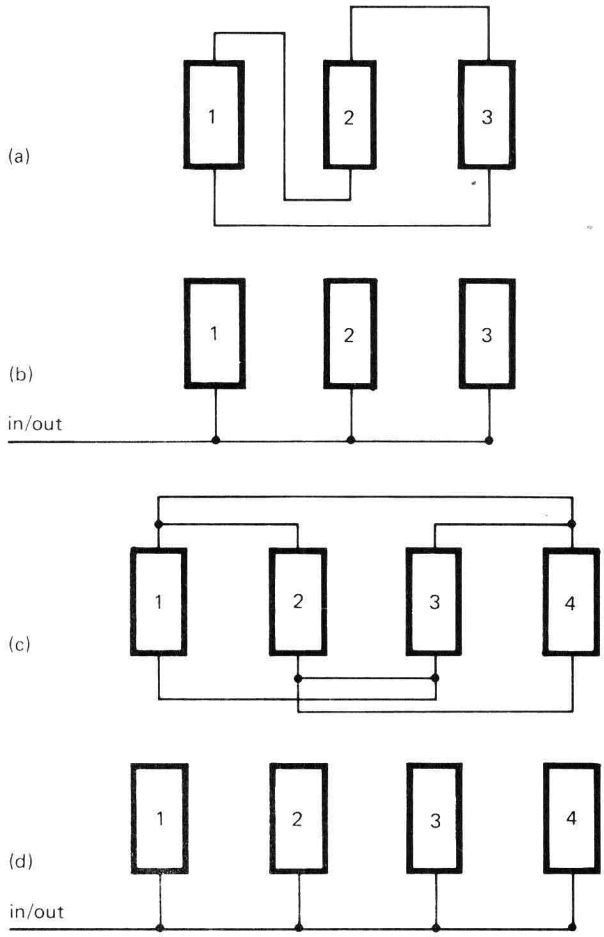

Fig. 1.2. Independent connections and bus connections. Connecting each unit of a system to each other independently (a,c) requires a large number of circuit paths. A bus method (b,d) is simpler and easier to use, provided that two signals are not sent at the same time

Words such as 'sent' and 'path' tend to lead us to think of a set of signals as something being moved from one place to another. The signals that are used in microprocessor systems, however, are electrical voltages, and a better description of their action would be 'sharing'. When a connecting line is used by any unit of a microprocessor system to 'send' a signal the electrical voltage of each line in the bus is set to one of two possible levels. Any other unit connected to the same lines can sense these voltage levels, but the voltages are not in any way removed from the sending unit and shifted to the receiving unit, any more than playing a record takes the groove from the disc and puts it into your ear! The value of a bus system is that the interconnections can be permanent, but the units are controlled so that when one unit places voltages on the lines of the bus, all of the other units on that bus will have the same set of voltage available — but they can be controlled so that only one (or more) selected units can use these voltages.

The Z-80 buses

The Z-80 buses are described as the data bus, the address bus and the control bus. The data bus consists of eight connecting lines which are bidirectional, meaning that they can carry signals from the microprocessor to other units or from the other units to the microprocessor. As we noted earlier, inputs and outputs can never be simultaneous, so that this use of the same set of connections for both purposes is not a disadvantage; it is in fact a considerable advantage in terms of the reduction of the number of connections that have to be made.

The address bus consists of 16 lines which are used for selecting units of memory. Every different arrangement of voltages on the address lines corresponds to a different portion of the memory being activated and connected to the data lines. This allows the microprocessor to 'write' — place a signal pattern on the data lines to be copied into the selected part of memory, or 'read' — when the signal placed on the data lines by the memory location is copied by the microprocessor.

The control bus consists of lines which are used to control the units attached to the data and address buses, or to allow the microprocessor to be controlled by external signals. Two important control bus lines of the Z-80 are the read and the write lines, which are connected to all the memory chips, and which send out signals from the microprocessor that will determine whether the selected part of memory is to be written to or read from. Writing implies that the content of the memory will be changed; reading always leaves the memory unchanged. Another control line is used to signal when the address bus is free to accept a

此为试读，需要完整PDF请访问：www.ertongbook.com

memory address; this signal is used to distinguish between the use of the address bus for memory and its use for the port or ports.

Among the control inputs are the interrupts and the reset. The two interrupt inputs allow the action of the microprocessor to be interrupted by signals which will then force the microprocessor to carry out a special piece of program whose starting address will be specified in some way. More details of this process are given in Chapter 7. The reset input, as the name suggests, clears all the data from the microprocessor, and forces the address bus lines to the number zero (that is, the signals on the lines are all at zero voltage).

In general, the signals that are present at the pins of the Z-80 are of less interest to the assembly language programmer than to the hardware designer but, since many of the hardware actions depend on software and vice versa, the hardware actions are dealt with in rather more detail in Chapter 2.

Binary number system

The signals which are present on the buses use only two voltage levels, known as logic levels 0 and 1 — the words 'logic level' often being omitted. Level 0 means a low voltage, somewhere between 0 and + 0.8 V, and level 1 means a higher voltage, between 3.5 V and 5.0 V. The wide tolerances of voltage at each level should ensure that no voltage levels in the system are ever ambiguous; all should be either at level 0 or at level 1. The use of two voltage levels leads naturally to the use of a numbering system which has two digits only, 0 and 1. This numbering system is the binary code.

Our conventional (denary) number system is based on digits 0 to 9, with the next whole number being written as 10, meaning one ten and zero units. Similarly, the number 365 means 3 hundreds (100, or 10^2) plus 6 tens (10, or 10^1) plus 5 units (1, or 10^0). Each *place* in the number shows how many units are to be counted and also the multiplier (1, 10, 100, etc.). This system of writing numbers is one of the many contributions of the ancient Arab world to mathematics and science, and it rapidly superseded the clumsy and arithmetically useless Roman number system. By using the position of each digit to indicate its power of ten, its significance or importance in the number is indicated. The '5' in 365 is the least significant digit; change it to 6 or to 4 and the change is of one part in 365, almost negligible. The '3' in 365 is the most significant digit; change it to 2 or to 4 and the change is of one hundred parts in 365 — certainly not negligible. Binary numbers can be written in the same form.

When we count using only the digits 0 and 1, however, the next

Place:	7	6	5	4	3	2	1	0
Value:	128	64	32	16	8	4	2	1

Fig. 1.3. Position values, or powers of two

number following 1 has to be 10, meaning 1 two and 0 units. The place of a digit now indicates the power of two rather than the power of ten; Fig. 1.3 shows a table of powers of two up to 2^7, embracing all the powers that will be used in Z-80 programming. Any number that can be written in denary (scale of ten) can also be written in binary (scale of two), but the binary version will contain more digits than the denary version if the number is greater than 1! The sequence of the first 16 numbers in a binary count is shown in Fig. 1.4.

```
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
```

Fig. 1.4. The first sixteen numbers (0 to 15) of a binary count

Conversion of a binary number into its denary equivalent is straightforward, using the table of powers of two shown in Fig. 1.3. Wherever there is a 1 in the binary number, its place value (the value of that power of two) is written down, and these numbers are then added to give the denary number. Conventionally, the Z-80 works with sets of eight binary digits, and this set is called a byte. One byte can range from 00000000 (zero) to 11111111 (denary 255), so that the numbers that the Z-80 handles can be in sets of this size range. This does not preclude the use of whole numbers greater than 255, nor of fractions and negative numbers, provided that they can be represented in byte form, of which more later. The important point is that one byte is the maximum that the Z-80 can read in or write out at one time. Figure 1.5 shows the ancient and useful method of converting a denary number into binary form.

Using single-byte units means that special provisions have to be made for numbers that are not positive whole numbers (integers). Since the binary system makes no provision for anything other than the digits 0

Method: Divide the number by two. Write the result and the remainder, which must be 1 or 0. Repeat with the result, until the last remainder, which is 1, is found. Write the remainders in order, starting with the last one.
This is the binary number equivalent to the denary number.

Example: 124

```
        2)124        remainders

          62             0
          31             0
          15             1
           7             1
           3             1
           1             1
           0             1
```

Binary number is 1111100 (read upwards)
In eight-bit form, this is 01111100

Fig. 1.5. Converting a denary number to binary

and 1, with no + or − signs, the sign of a number, positive or negative, has to be indicated by making use of one of the bits of a byte. The bit used is the most significant bit, and the meaning allocated to it is that, if this bit is 0, the number is taken as being positive, and if this bit is 1, the number *may* be taken as negative. Note *may* — if we decide that the most significant bit will not be used as a sign bit in our programs, then we are free to do so.

Signed numbers

The use of the most significant bit as a sign bit follows from the method of creating the negative equivalent of a positive binary number. The procedure is shown in detail in Fig. 1.6; the binary number is inverted first, writing a 1 for each 0 and a 0 for each 1 in the number. Following inversion, 1 is added so as to produce the negative form, known as the 'twos complement'. This is always a number whose most significant bit

Write the binary number, single byte.
Invert each byte, writing 1 for 0 and 0 for 1.
Add 1 to the result. This is then the negative form of the original number.

Example:

```
        Byte is     01101101
        Invert      10010010
        Add 1       10010011   this is the negative form
```

Fig. 1.6. Creating the negative form of a binary number. The number must use seven bits only, if its negative form is to take up one byte

must be 1, and this, or any other number in which the most significant bit (msb) is used as a sign bit, is called a 'signed binary number'. For single-byte numbers, the range of values that a signed number can have is −128 to + 127, as distinct from the 0 to 255 range of the same bytes considered as unsigned numbers.

The sequence of a binary countdown from positive values through zero to negative values looks rather odd in consequence of the use of twos complement; a portion of such a countdown is illustrated in Fig. 1.7.

11111011	−5
11111100	−4
11111101	−3
11111110	−2
11111111	−1
00000000	0
00000001	+1
00000010	+2
00000011	+3
00000100	+4
00000101	+5

Fig. 1.7. Part of a binary countdown with its denary equivalent

The important part is that the negative form of a binary number bears little resemblance to the positive form; + 5 in binary is 00000101 and −5 is 11111011. This system has, however, the very considerable advantage of permitting the microprocessor to use the same circuits to carry out both addition and subtraction. Novice programmers are often perplexed by a number such as 10011011, which if regarded as unsigned is 155 denary but which if regarded as signed is 101 denary. The curious point is that only very seldom is it important to distinguish between the meanings unless you are designing mathematical programs, because the microprocessor will treat the number in the same way no matter how you think of it. In practice, the programmer has to be aware of when the microprocessor will treat a number as being signed and when it will treat the number as being unsigned. Since we do not generally use binary numbers in the course of our programming, the number of times that the programmer is involved is negligible.

Integers and floating-point numbers

Most computers make provision for integers, meaning positive whole numbers. These are used, for example, for line numbers (except by the Camputers LYNX) and for address numbers, and they consist of two bytes each. The range of numbers is 0 to (256 × 256)−1, which is 65 535; alternatively if the numbers are thought of as signed, the range is −32 768 to + 32 767. A greater range can be obtained by using a larger number of bytes.

Numbers which are not defined as integers are treated as floating-point numbers, which are stored in mantissa-exponent form. This is analogous to the 'standard' or 'scientific' notation that is used by most calculators, in which a number is expressed in denary as a value between 0 and 10 multiplied by a integer power of ten. Numbers greater than 1 will use a zero or positive power of ten, numbers less than 1 will use a negative power of ten. Using this scheme, we can write 261 700 as 2.617×10^5, and 0.00114 as 1.14×10^{-4}. If you are unfamiliar with this method of writing denary numbers, then skip the next section — it's not for you!

Binary floating-point numbers are written in the form .XXXXXXXX for the modulus (eight bits following a binary point in this example) and with a separate exponent, the size of the power of two.

1. If number is greater than 1, divide by the next higher power of 2
2. Write in exponent – mantissa form.
3. Convert mantissa to binary fraction by subtracting negative powers of two.

Example: 24 which is greater than 1, so divide by next higher
power of 2, which is 32. This gives 24 as 0.75×2^5
Now $2^{-1} = 1/2$, which is 0.5
0.75 is greater than 0.5, so write .1B, and subtract to get 0.25
$2^{-2} = 1/4 = 0.25$; in binary this is .01B, and subtracting from
the 0.25 that remained leaves zero. This ends the conversion.
The fraction 0.75 denary is .11 binary.
The complete number has a mantissa of .11000000 and exponent of 10000101

Fig. 1.8. The mantissa-exponent form for a floating-point number

The scheme which is followed is that of expressing the number as a modulus whose value lies between .10000000 and .11111111 and an exponent whose value is 10000000 added to the actual value of the exponent (all figures in binary). The binary fraction is obtained by the conversion of the denary fraction to binary, which is illustrated in Fig. 1.8, having first put the number into appropriate form by dividing it by the next higher power of 2. For example, the number 56.6 is put into binary fraction form by first dividing by the next higher power of 2, which is 64. This gives the number 56.6 as being equal to $(56.6/64) \times 2^6$ (since 64 is 2^6). The fraction, 56.6/64 is 0.884375 in denary fractions, so that the mantissa of this number would be the binary equivalent of 0.884375 denary.

Looking at another example, the denary fraction 0.0246 can be converted by dividing by 2^{-5} (which is 0.03125), so that the form of the number is a mantissa of 0.0246/0.03125, in denary, and an exponent of 2^{-5}.

The exponent number, 6 in the first example, would be added to 128 (denary) to give 134 denary, in the second case, 128−5 is 123. These exponents in binary therefore become 10000110 for the 56.6 example, and 01111010 for the 0.0246 example. This allows the system to distinguish between the numbers which started greater than 1 and these which started as less than 1.

The fractional part of the number which in denary is always between 0.5 and 1 can be expressed as a number of bytes of binary. The greater the number of bytes that is used to express this fraction, the more precise the arithmetic, because denary fractions seldom convert exactly to binary fractions. Binary fractions are exact only when the denary number that is being converted is a negative power of two (Fig. 1.9) like

Power of two	*Denary*	*Binary*
−1	1/2	.1
−2	1/4	.01
−3	1/8	.001
−4	1/16	.0001
−5	1/32	.00001
−6	1/64	.000001
−7	1/128	.0000001
−8	1/256	.00000001

Fig. 1.9. Table of negative powers of two

2^{-1} (which is $\frac{1}{2}$, 0.5), 2^{-2} ($\frac{1}{4}$, 0.25), 2^{-3} ($\frac{1}{8}$, 0.125) and so on. All other numbers convert inexactly, even when four bytes are used to express the fraction as is normal in modern computers. This can lead to errors in the course of arithmetic when numbers are converted to binary fractions and back again. We encounter a similar situation when we work with numbers such as $\frac{1}{3}$ in decimal. By using four bytes to hold the fractional part of the number, however, the accuracy should be to enough places of decimals (in the denary equivalent) for all but the most stringent purposes. To avoid mistakes, though, it is essential to round off numbers to the number of decimal places that is needed.

Binary-coded decimal (BCD)

BCD, meaning binary-coded decimal (or denary), is a method of coding denary numbers using binary codes but without converting the complete number to binary form. BCD is used extensively when numbers have to be displayed on seven-segment displays, because each unit of such a display is used for one digit. Since a denary digit can range in value from 0 to 9, binary 0000 to 1001, a set of four binary digits is needed to represent each single denary digit.

A number such as 255 is represented in BCD by the binary codes for

此为试读，需要完整PDF请访问：www.ertongbook.com