FEATURE

Introducing Microprocessors Part 8

Dealing with programming, including flow charts and languages.

MIKE TOOLEY

The general learning objective for Part Eight is that readers should understand simple assembly language programs used to control external devices connected to the parallel port of a microprocessor-based system.

The specific objectives for Part Eight are as follows:

6.1 Languages

6.1.1 Explain the need for programming languages and distinguish between high level and low level languages.

6.1.2 State the desirable characteristics of programming languages for each of the following applications:

real-time control systems
data processing
systems and application software.

6.2. Assembly Language Programming

6.2.1 Describe the logical procedure which must be adopted in order to create a satisfactory program.

6.2.2 Describe algorithms and draw flowcharts relating to simple problems.

6.2.3 Identify and use common flowchart symbols.

6.2.4 Explain what is meant by assembly language.

6.2.5 Describe, with typical examples, the use of mnemonics in assembly language programs.

6.2.6 Write, hand-assemble, enter, test and debug simple programs using a subset of the instruction set of any common 8-bit CPU. to:

a) add two eight-bit data values from RAM and place the result in a third RAM location

b) operate an external relay or LED in a pre-defined on/off sequence.

Programming Languages

In order to simplify the process of producing working programs, the software developer may use one (or more) of a number of programming languages to simplify the task of producing a working program. The choice of language depends essentially upon several factors including the application concerned, the degree of familiarity which the programmer has with the language concerned, and the availability of the necessary development software for the microprocessor system to be used.

Languages which are well suited to producing software in fields such as data processing are generally not well suited to producing software for such applications as real-time control. Furthermore, a programmer who is competent in a language such as Pascal may be very much out of his or her depth with Forth.

Happily, a range of languages is available to most modern microcomputers and the final choice of language will take into account such factors as compactness (i.e. size of program code generated), speed of execution, ease of use, portability (i.e. ability to transfer code easily from one system to another), and ease of maintenance.

The desirable characteristics of languages for three typical applications (real-

	Real-time control	Data processing	Applications software
Speed of execution?	MUST be very fast	not generally critical	as fast as possible
Size of code	MUST be very compact	not generally critical	should be reasonably compact
Portability	not generally critical	should be reasonably portable	MUST be highly portable
Availablity of data structures	not generally critical	MUST offer a range of powerful data structures	should offer a range of data structures
Example language	Assembly language	Pascal	C

Table 1. Characteristics of programming languages for three typical applications.

time control, data processing and applications software) are listed in Table 8.1

High and Low-Level Languages

Programming languages are often classified as "high level" or "low level". High level languages are generally those which are "procedure oriented" and are written in structured English such that programs are easily readable. Each program statement in a high level language will normally have a recognizable function and, furthermore, will be equivalent to several assembly language instructions.

Low level languages are those which are "machine oriented" and are thus close to the binary "machine code" which is executable by the microprocessor. Assembly language is an example of a low level language which uses mnemonic operational codes (opcodes) and symbolic addresses (instead of actual memory locations). The individual program statements in a program written in a low level language may not in themselves, be particularly meaningful and therefore comments are generally added to clarify the action of the statements.

Assembly Language Programming

In Part Three we briefly mentioned that assembly language was a low-level language in which the instructions are presented in mnemonic code for later translation into the binary code acceptable to the microprocessor. Readers will doubtless recall that this process is normally carried out by means of an assembler program.

The assembler acts upon a text file written in mnemonic assembly language code (known as "source code") and generates a binary code (known as "object code") within the microcomputer's memory. Thereafter, the object code constitutes a directly executable program i.e. we simply load the Instruction Pointer or Program Counter with the entry (start) address of the code and execution commences.

Some assemblers produce intermediate programs in hexadecimal format such that the mnemonic source code is first translated into a hexadecimal file. This file may be subsequently stored on disk (as a "hex, file") or loaded into the microcomputer's memory ready for execution.

Alternatively, where programs are extremely short, it is possible to dispense completely with the services of an assembler and resort to "hand assembly". This, somewhat tedious process, involves first writing the program in assembly language mnemonics and then translating each instruction (operation code and operand) into hexadecimal code which is then loaded into an appropriate region of memory prior to execution. Hand assembly requires the services of a machine code "monitor" or "debugger". Alternatively a rather more specialized "hexadecimal code loader" may be used.

At this point, it is worthwhile reminding readers of the simple example

Address (hex)	Contents (hex)	(binary)
1800	3E	00111110
1801	01	00000001
1802	06	00000110
1803	02	00000010
1804	80	10000000

which we used in Part Three. We wished to add together two bytes of immediate data (stored in RAM as part of the program) using our hypothetical microprocessor (IMP). This task involved three instructions. The first loaded the first operand (in this case a byte of immediate data) into the accumulator (A). The second loaded the second byte of data into the B register. Finally, the third instruction added together the contents of the A and B registers and placed the result back into the accumulator.

Assuming that the data bytes have hexadecimal values of 01 and 02 respectively the program takes the following form:

```
LD A,01
LD B,02
ADD B
```

Its hexadecimal representation may be found by referring to the instruction set as follows:

LD A,01 is represented by 3E (the opcode) followed by the byte of immediate data (in this case 01)

LD B,02 is represented by 06 (the opcode) followed by the byte of immediate data (in this case 02)

ADD B is represented by 80 (the opcode) and there is no operand

The hexadecimal representation of the program is thus:

```
3E 01
06 02
80
```

Assuming that the program is to commence at an address of 1800H, the contents of IMP's memory would be as shown in the table below:

After execution of the program the Instruction Pointer (Program Counter) will have reached 1805H and the A and B registers will contain 03 and 02 respectively. Note that, if we wished to test the program it would be necessary to halt the microprocessor at address 1805 otherwise

it would continue to execute whatever code it came across. This is a potentially dangerous situation as the microprocessor cannot distinguish between random data and program code (the former may cause the system to lockup in an endless loop or even overwrite the program with spurious data).

Now let's consider a more complex example. Suppose that we wish to add together two eight-bit data values stored in RAM (not as part of the program) and place the result into a third RAM location. We will assume that, in both cases, the bytes of data are stored in memory locations 1900H and 1901H and that the result is to be deposited at address location 1902H. To make life easier, we will ignore the possibility of an overflow occurring (as would be the case if the sum of the two bytes were to exceed 255 decimal or FFH).

The assembly language program, and corresponding hexadecimal machine code, will be different for different microprocessors. Indeed the programmer may have to adopt slightly different techniques due to the constraints imposed by the instructions and addressing modes (i.e. methods of locating the data used by an instruction) available with the microprocessor concerned.

The following routines for the Z80 and 6502 illustrate this point:

Z80 Code

```
LD A, (1900H) ;get first byte
LD B,A    ;transfer to B
LD A, (1901H)    ;then get the second byte
ADD B    ;find their sum
LD (1902H),A    ;and store the result
```

6502 CODE

```
CLC ;first clear carry flag
LDA $1901 ;get second byte
ADC $1900 ;and add to first
STA $1902 ;then store the result
```

Problem 8.1

Use implied addressing (with register pair HL acting as a pointer) to produce an alternative Z80 program which will have the same effect. (NB: A subset of the Z80 Instruction Set appeared on Data Card 4.)

Assembly Language Programming Technique

Regardless of the processor involved, a number of techniques can be used to improve the overall efficiency of a program and also make it easier to maintain. Many of these techniques are easy to implement and merely require a little forethought and self- discipline on the part of the programmer.

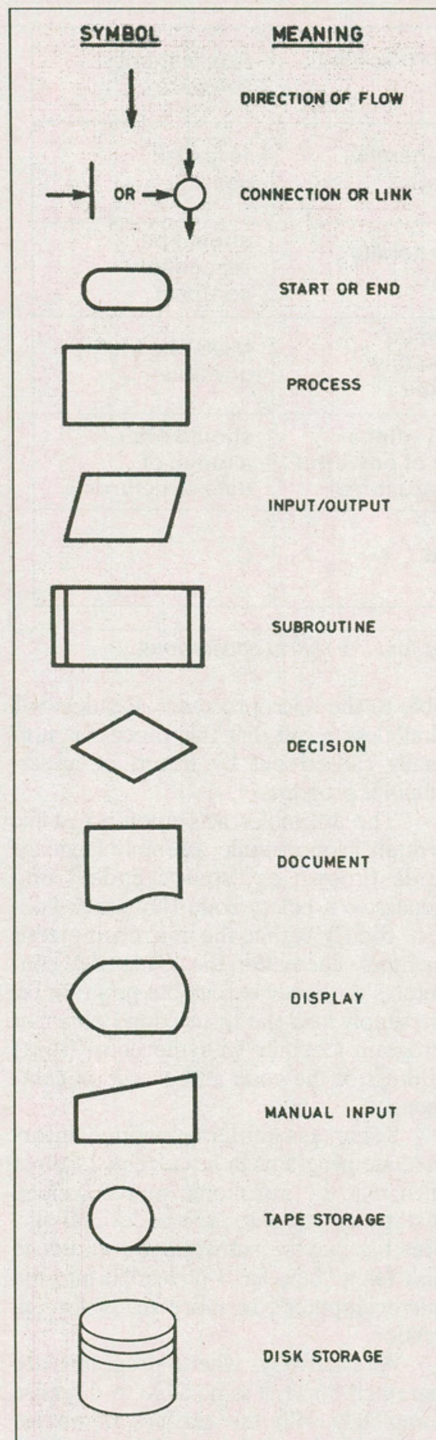

Fig. 8.1. Flowchart symbols.

Programs will invariably comprise a number of smaller modules each having an identifiable function. The overall structure of the program should be defined at a very early stage and no attempt should be made at coding any of the modules required by the program until the overall program structure has been finalized.

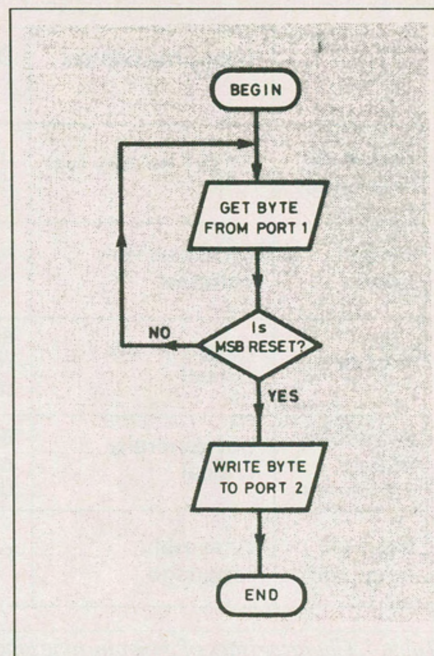

Fig. 8.2. Flowchart for a simple I/O program.

An algorithm is a method of describing the sequence of operations which should be followed in order to solve a problem. An algorithm is often expressed using a diagram to show the sequence of events. This diagram is known as a flowchart and a standard set of symbols (Fig. 8.1). These symbols indicate the type of process involved and the flowchart is annotated with brief explanatory comments which are inserted within the symbols to which they refer.

The overall structure and flow of a program should be defined using one, or more, flowcharts at an early stage. Alternatively (or in addition to a flowchart representation) the sequence of the program may be described by a series of statements written in a form of structured English. In any event, the overall flow of the program should be sequential, there should be only one entry and one exit point, and all transfers of control (i.e. jumps and calls) should be explicit.

As an example of using flowcharts and structured English statements, consider the case of a simple routine which reads a set of switches connected to an input port, loops until the switch connected to most significant bit (MSB) is closed and then transfers the byte read from the switches to an output port.

A flowchart for the process is shown

```
; READ BYTE FROM PORT1, LOOP UNTIL MSB RESET,
; THEN TRANSFER BYTE TO PORT2
;
; EXIT: A = (PORT1), BC = PORT2, ZF = reset
;
; REGISTERS AFFECTED: A, B, C, F
;
GETBYTE:    LD    BC,PORT1        ; Get byte from
            IN    A,(C)           ; PORT1
            BIT   7,A             ; Is MSB reset?
            JR    Z,GETBYTE       ; No, keep trying
            LD    BC,PORT2        ; Yes, send byte
            OUT   (C),A           ; to PORT2
            RET
```

Fig. 8.3. Simple Z80 I/O subroutines.

in Fig. 8.2. Alternatively, we could express the problem in terms of the following structured English statements:

Begin
Repeat
Get byte from PORT 1
Until MSB of byte is reset

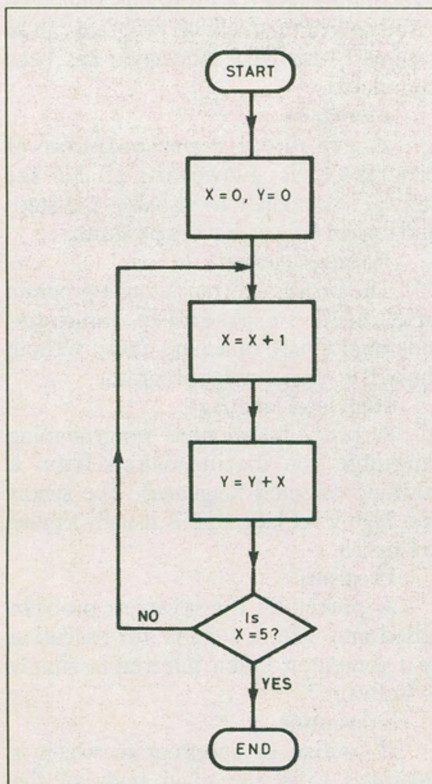

Fig. 8.4. See Problem 8.3.

Output byte to PORT 2
End

Armed with one or other of the foregoing algorithms, it is a relatively simple matter to develop the code. A particular solution based on the Z80 microprocessor, is shown in Fig. 8.3.

Problem 8.2

Sketch a flowchart to describe the steps in finding the sum of two data values (taken from memory) and place the result back into memory.

Problem 8.3

The flowchart in Fig. 8.4 indicates a process. Determine the values of the variables X and Y upon exit.

Subroutines

The fragment of code shown in Fig. 8.3 constitutes a subroutine. This is a section of code which may be called from various points in the main program (using the CALL instruction) and returned to (by means of the corresponding RETurn) instruction. If desired, both the CALL and RETurn instructions can be made conditional on the contents of the flag register. Furthermore, a subroutine may have several conditional RETurn statements.

The CALL instruction saves the old value of the Instruction Pointer (or Program Counter) in the stack before replacing it with the value of the subroutine start address. On returning from the subroutine, the Instruction Pointer (or Program Counter) is loaded with the value saved on the stack so that the main program can be resumed at the point at which it was left.

Parameters can be easily passed to and from subroutines by simply placing them in one or more of the CPU registers. Alternatively, parameters may be passed using the stack or by reserving an area of memory in which parameters can be deposited before making the call

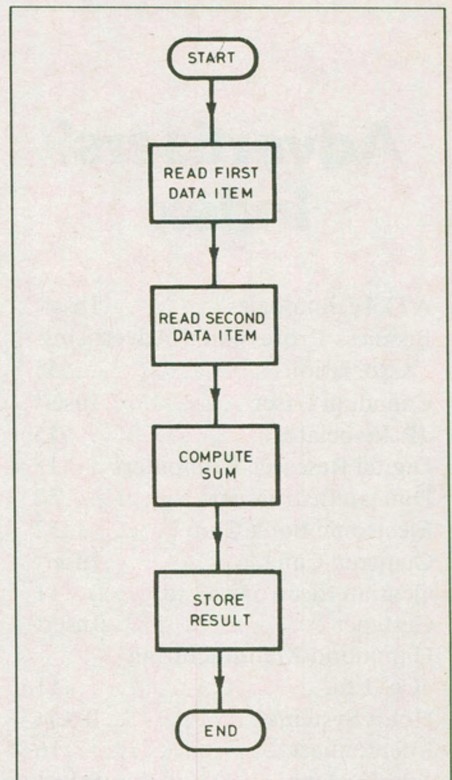

Fig. 8.5. Answer to Problem 8.2.

and recovered after the call has been made. This allows the passing of a much greater number of parameters than would be possible using just the CPU registers.

Care must be taken to preserve the contents of any CPU registers that may be modified as a result of executing a subroutine call and that are required in subsequent processing. It is thus essential to have a knowledge of the effect of a subroutine on the CPU registers (in any event, this should be clearly indicated in the source code). Furthermore, subroutines should be designed so that they minimize usage of the CPU registers, thus keeping things simple for the programmer and reducing any potential overhead associated with storing and retrieving register contents.

The use of subroutines makes programs easy to maintain and allows modules to be easily transferred into other programs without having to rewrite an entire program. This is an important point and one which can save the programmer a great deal of time.

Programming I/O Devices

Readers may recall that we concluded last month's instalment by describing a representative output driver arran-

gement based on a programmable parallel I/O device. We also stated that the external devices could be easily operated by simply writing an appropriate data byte to the port in question. As an example a binary value of 11000111 (hex. C7) written to Port A will illuminate the three LEDs connected to PA0, PA1 and PA2 and operate the relays connected to PA6 and PA7. To turn the LEDs and relays off, a binary value of 00000000 (hex.00) should be sent to Port A.

Readers may recall from Part Six that a microprocessor employing memory mapped I/O (such as the 6502) can simply write data to an output port using an instruction of the form; STA address (the accumulator must first be loaded with the requisite byte of immediate data). In the case of a microprocessor which uses port I/O (such as the Z80), the accumulator is again first loaded with the requisite byte a data and then an output instruction of the form OUT (port), A is used.

In either case, it will usually be necessary to configure the programmable I/O device (this will often be a 6502 PIA or 6522 VIA in the case of 6502 CPU or a Z80-PIO or 8255 PPI in the case of a Z80 CPU) before I/O can commence. The configuration routine will be very much dependant upon the hardware configuration and type of I/O device fitted. @SUBHEAD = Problem 8.4

A microprocessor based system is fitted with one input and one output port. The input port is connected to eight switches and the output port is connected to eight LEDs. Devise a simple assembly language program which will continuously read the switches and operate the respective LEDs in each of the following cases:

(a) Using a 6502 CPU memory mapped with the following port addresses: Input, 8002H Output, 8005H

(b) Using a Z80 CPU employing port I/O with the following port addresses: Input, FBH Output, FDH

Answers to Problems

8.1 Either of the following programs would prove satisfactory:

(a) LD HL, 1900H
LD, B, (HL)
INC HL
LD, A, (HL)
INC HL
ADD B
LD (HL), A

(b) LD HL, 1900H
LD A, (HL)
INC HL
ADD (HL)
INC HL
LD (HL), A

Note that the program in (b) is one byte shorter than that in (a)

8.2. See Fig. 8.5

8.3. X = 5, Y = 15

8.4. (a) LDA $8002; get byte from switch bank

STA $8005; and send it to the LED

(b) IN A, (FBH); get byte from switch bank

OUT (FDH), A; and send it to the LED

Glossary for Part Eight

Algorithm

The sequence of steps (presented in a clearly understandable form) which describe the procedure used to solve a problem.

Call

An instruction to jump to a subroutine. A jump to the specified address is performed, but the contents of the Instruction Pointer (or Program Counter) is saved so that the (calling) program can be resumed when the subroutine has been completed.

Flowchart

A graphical representation of program logic. Flowcharts enable the software developer to visualize the steps and logical flow within the program.

Hand assemble

The process of translating a program presented in assembly language mnemonics into machine code without the aid of an assembler program.

High level language

A problem oriented programming language (as distinguished from a machine oriented language). The syntax of a high level language is usually similar to English.

Program

A procedure for solving a problem coded into a form suitable for execution by a computer. Often referred to simply as "software".

Subroutine

A routine or subprogram which is separated from the main body of the program and which is executed by means of a CALL instruction (or its equivalent in a high level language). At the conclusion of the subroutine, control reverts to the main (calling) program at the point at which it was left. ■