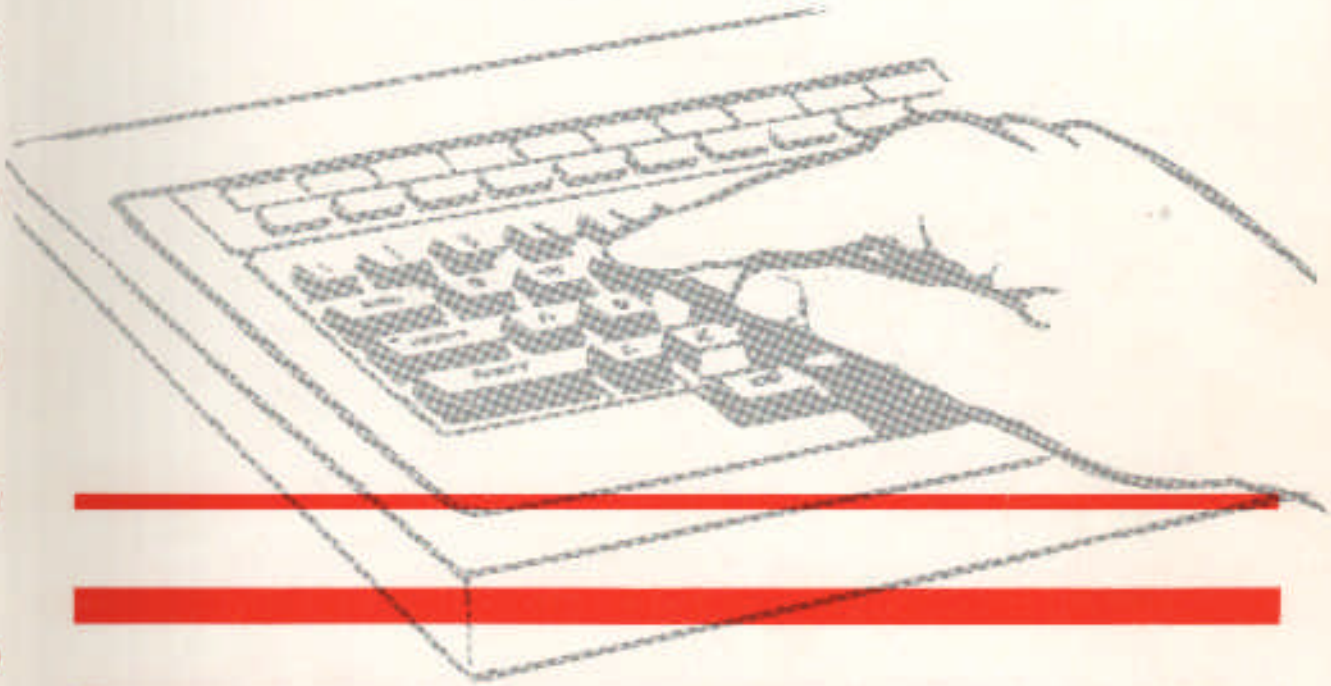
I E Morley

Machine Code Programming for the

NASCOM 1 and 2



By: G.R. Wilson

Published By: interface

Machine Code Programming For the NASCOM 1 and 2

By: G.R. Wilson

interface components

Interface Components Limited. Oakfield Corner, Sycamore Road, Amersham, Bucks. HP6 6SU. Tel: (02403) 22307

MACHINE CODE PROGRAMMING
for the NASCOM 1 & 2

PREFACE

This book is intended for the novice NASCOM programmer. The approach is to take the reader step-by-step through most of the impressive Z80 instruction set, illustrating the different instruction groups with short programs.

Programming home computers is essentially a creative activity, requiring both imagination and a knowledge of the computer. It is hoped that this book will provide the reader with the 'nuts and bolts' so that he can create programs for the use and entertainment of both himself and others.

Graham Wilson

MACHINE CODE PROGRAMMING
for the NASCOM 1 & 2

MACHINE CODE PROGRAMMING
for the NASCOM 1 & 2

A Step-by-step Guide

G.R.Wilson

CONTENTS

APPENDICES

MACHINE CODE PROGRAMMING
for the NASCOM 1 & 2

1.0 NUMBER SYSTEMS

There are four number systems with which familiarity is required: decimal, binary, hexadecimal, and binary coded decimal.

1.1 Decimal

Counting numbers:

```
0
1
2
3
4
5
6
7
8
9
10  <-------  Here, add 1 to next
11            column and restart
12            the digit sequence.
:
:
:
100
:
:
```

Interpretation:

eg.

```
100 10   1      Column 'weights'
 3   9   5
 |   |   |
 |   |   |-------> 5 x 1   = 005
 |   |-----------> 9 x 10  = 090
 |---------------> 3 x 100 = 300
                             ---
                             395
```

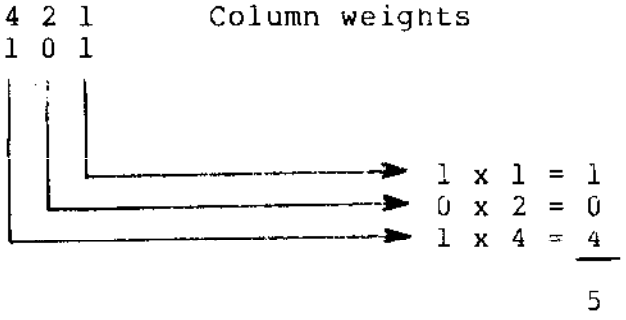
1.2 Binary

Counting numbers:

```
 0 <=> The two Binary digITs
 1      or 'bits'.
10 ---Here,add 1 to the
11      next column and
100     restart the
101     bit sequence.
 :
 :
```

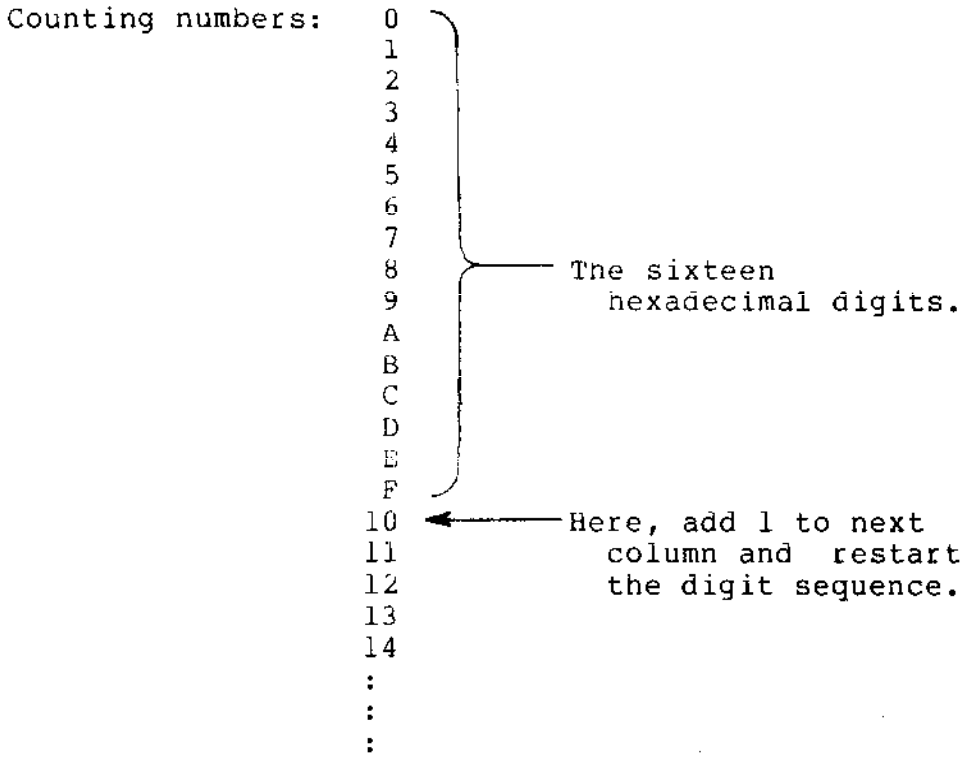
Interpretation:

eg.



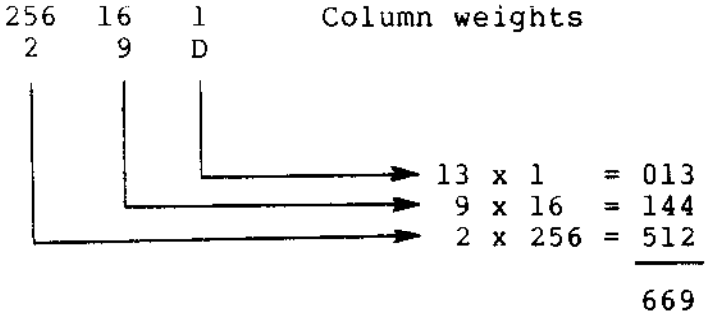
ie. 101b = 5d where b signifies a binary number
and d signifies a decimal number.

1.3 Hexadecimal



Interpretation:

eg.

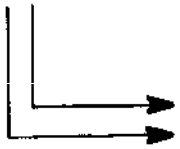


ie. 29DH=669d

1.3.1 Use of Hexadecimal

Consider:	1 0 0 1	1 1 0 1	=157d
	1001	1101	a) Divide into groups of four bits, starting at right-hand end.
	9	D	b) Write hexadecimal equivalent of each group of four bits.

Observe that 9D



13 x 1 = 013
9 x 16 = 144
157d

Thus, 10011101 is more conveniently written as 9D. This gives rise to fewer errors when writing numbers.

1.4 Note the number of different patterns of 0 and 1 that are possible using the following numbers of bits:

No. of bits	No. of different patterns possible.
4	16
8	256
10	1024 (=1K)
16	65536 (=64K)

1.5 Representation of negative binary numbers

Imagine a car mileometer being wound backwards. After reading 0000 the next number shown will be 9999. It is possible therefore to regard 9999 as representing the number one-less-than-zero, or -1.

If the milometer were in hexadecimal it would roll back from 00 to FF, which is thus regarded as representing -1. This method of representation is referred to as being 'complementary'.

Hex	Decimal
7F	+127
7E	+126
7D	+125
:	:
:	:
03	+3
02	+2
01	+1
00	0
FF	-1
FE	-2
FD	-3
:	:
:	:
:	:
:	:
82	-126
81	-127
80	-128

The largest number is +127 and the most negative number is -128. (Together with zero this gives a total of 256 numbers as seen on page 1-4.)

Note that all positive numbers have the left-most bit of their binary equivalent equal to 0, while all negative numbers have it equal to 1.

The weights of an 8 bit '2s complement' number are:

-128 64 32 16 8 4 2 1

eg. 8A

1000 1010

-118

1.6 Binary Coded Decimal,BCD

Since much of the input and output of a system is in decimal, some method of coding the decimal digits 0 to 9 is desirable. The most common method is simply to use the pure binary equivalents for the digits. Thus:

Decimal	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

eg. To represent 1981 in BCD:

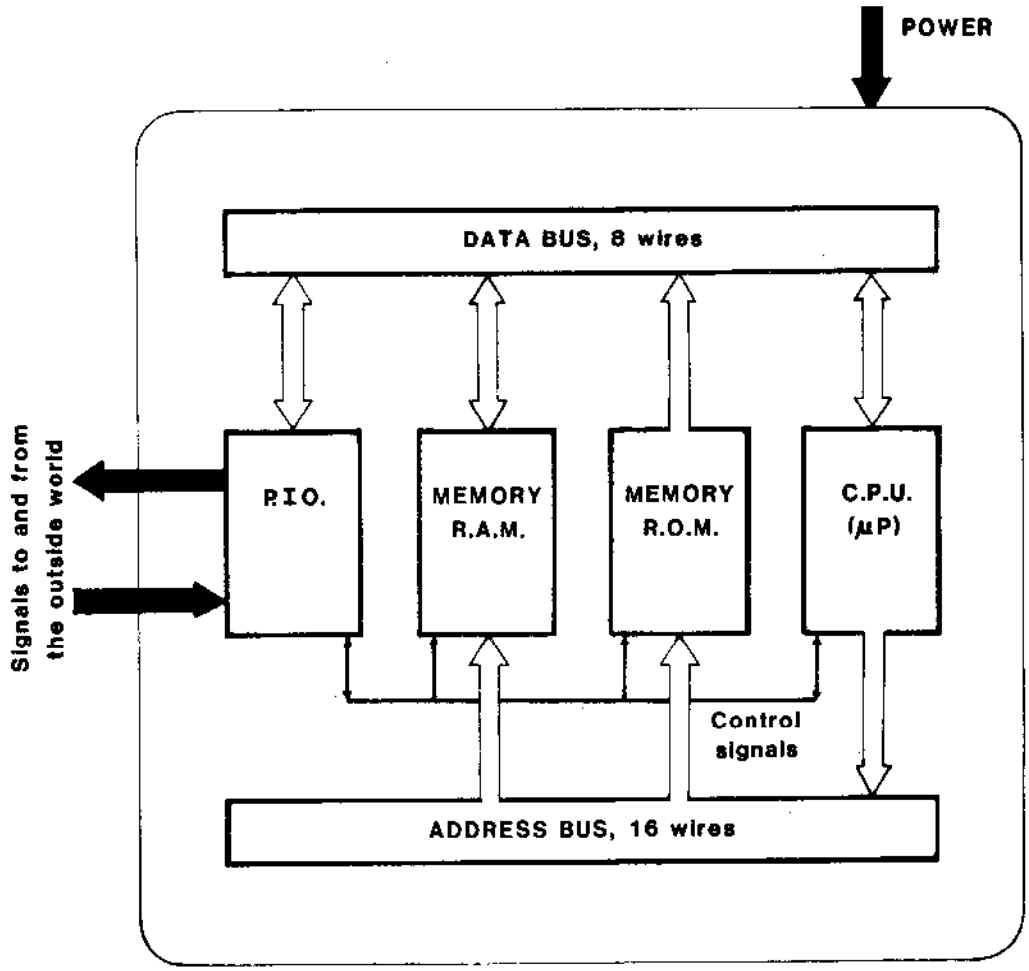
```
0001 1001 1000 0001
  1    9    8    1
```

1.7 The first few numbers in the different systems.

Hex	Dec	Bin	BCD
00	00	00000000	00000000
01	01	00000001	00000001
02	02	00000010	00000010
03	03	00000011	00000011
04	04	00000100	00000100
05	05	00000101	00000101
06	06	00000110	00000110
07	07	00000111	00000111
08	08	00001000	00001000
09	09	00001001	00001001
0A	10	00001010	00010000
0B	11	00001011	00010001
0C	12	00001100	00010010
0D	13	00001101	00010011
0E	14	00001110	00010100
0F	15	00001111	00010101
10	16	00010000	00010110
11	17	00010001	00010111
12	18	00010010	00011000
13	19	00010011	00011001
14	20	00010100	00100000

2.0 COMPONENTS OF A SIMPLE MICROCOMPUTER

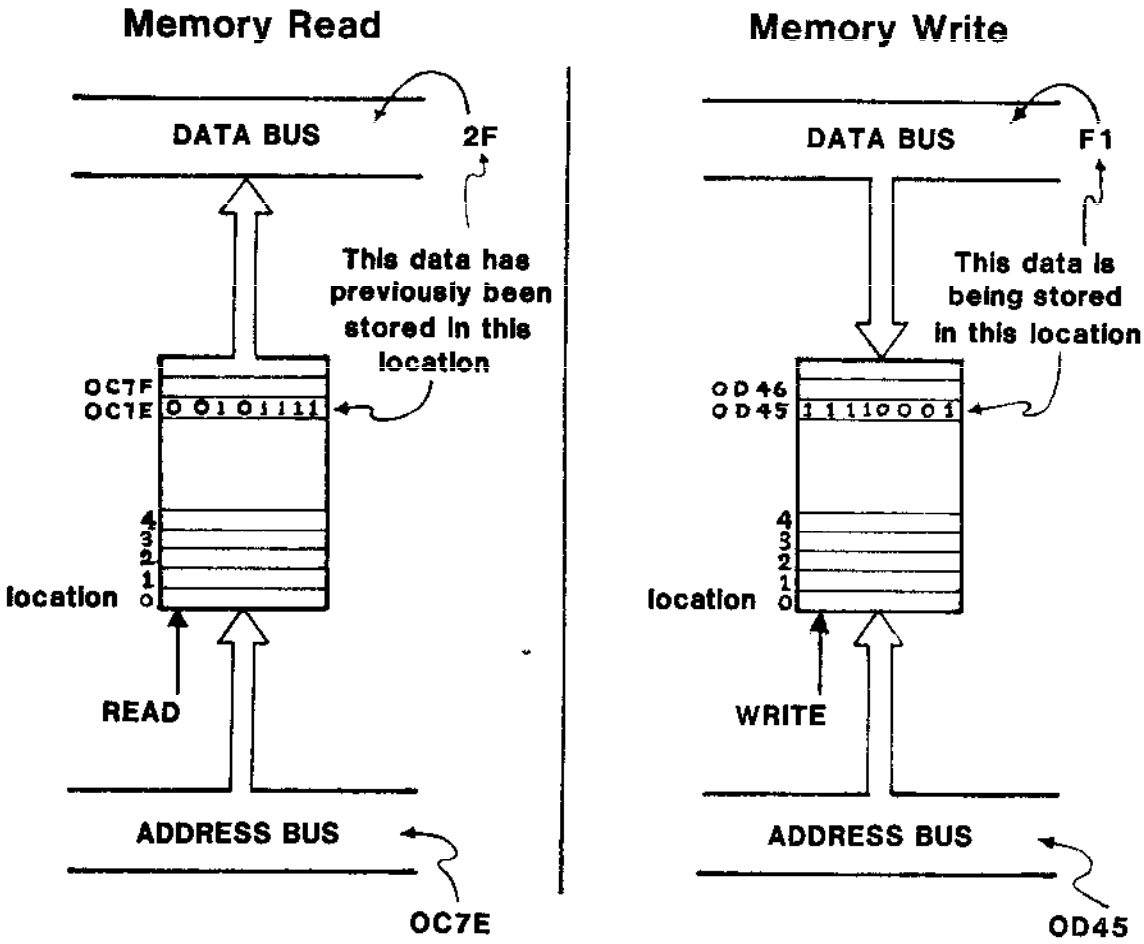
The components of a simple microcomputer are shown in the figure below. They are the central processing unit (CPU), which is the microprocessor itself, two forms of memory (RAM and ROM), and a peripheral input-output device (PIO). These are interconnected by two buses, the data bus and the memory bus. The data bus comprises eight wires onto which eight bit 'bytes' of data may be placed by one component for transfer to another. The address bus comprises upto sixteen wires onto wnich the CPU may place a sixteen bit address for transfer to the other components. In addition there are a few control signals which allow the CPU to control tne other components; for example, the RAM needs a control signal to tell it whether to read or write.



A Minimal Microcomputer

2.1 Random Access Memory

This is more descriptively called read-write memory since it can receive data from the data bus and store it, ie. data may be written into it, also it can place data on the data bus, ie. data may be read from it. These two operations are shown in the figure below.



RAM operations

The RAM may be thought of as upto 65,536 individual eight bit (ie. single byte) registers. Each register may be individually accessed by placing its address, ie. its location number, on the address bus.

When the control signal indicates a memory read operation, the eight bits of data stored at the addressed location are placed on the data bus. In the diagram there is 0C7E on the address bus and 2F stored in location 0C7E. Thus 2F will appear on the data bus.

When the control signal indicates a memory write

operation, the eight bits of data on the data bus are written into the memory location whose address is on the address bus. In the diagram, F1 is shown on the data bus and 0D45 on the address bus. Thus, F1 will be written into location 0D45.

2.2 Read-only Memory

The operation of the ROM is exactly the same as the read operation of the RAM. The data in ROM is entered by a special process prior to it being used in the microcomputer.

2.3 Use of RAM and ROM

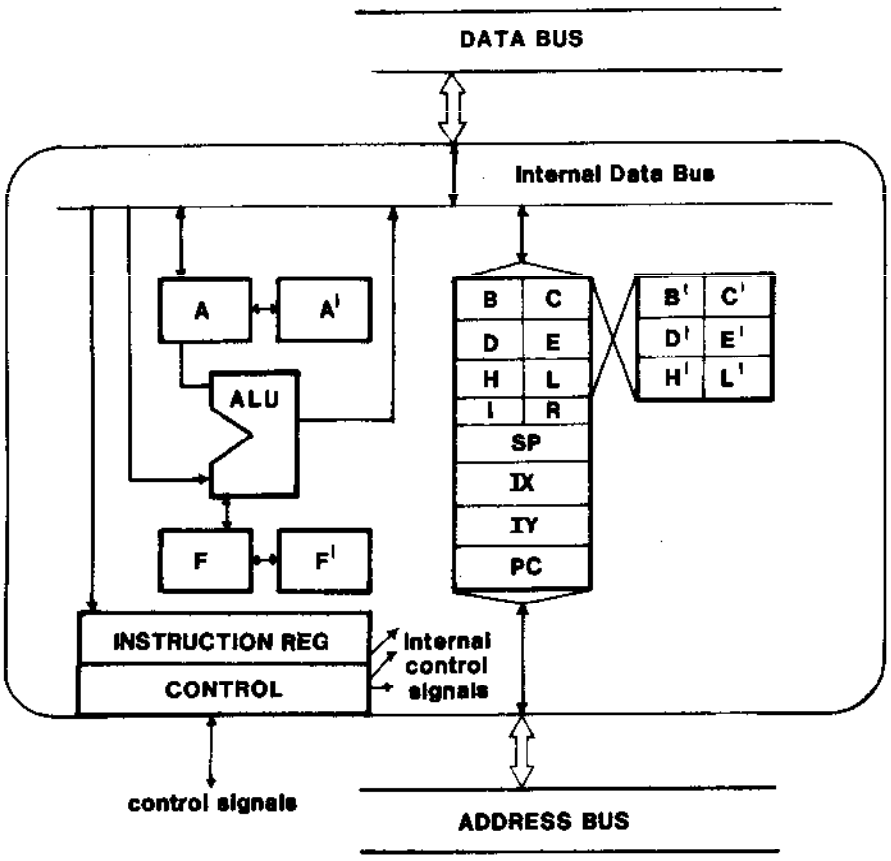
The most useful characteristic of ROM is that its contents do not dissappear when its power supply is removed. It is therefore used to hold information which is always required by the microcomputer. In the NASCOM a program called NAS-SYS is held in ROM and it is this program which allows commands to be entered from the keyboard. Another desirable characteristic of ROM is that its contents cannot be accidentally overwritten.

On the other hand, RAM is volatile, ie. its contents do dissappear when its power supply is removed. On first applying power, RAM locations contain random patterns of 0 and 1.In the NASCOM, as in other machines, the RAM is used to store the program and data entered by the user.

NAS-SYS, and other ROM-based programs, may be supplied in either ROM or EPROM. The latter type, known as erasable, programmable read-only memory, has a quartz window in the package and may be erased by exposure to intense ultra-violet light. New data may then be written into them by means of a special device known as a 'programmer'.

2.4 The Central Processing Unit

A diagram of the CPU of the Z80 is shown below. The CPU is essentially a collection of registers for temporary storage of data, an arithmetic and logic unit or ALU which carries out the arithmetic and logic operations on data, and the necessary control circuits.



Z80 Microprocessor

An instruction is read from the memory via the data bus into the Instruction Register. The instruction itself is then executed by the control circuits. The instructions all cause simple operations to occur, such as transferring data from a register to a memory location, or adding the numbers in two of the registers.

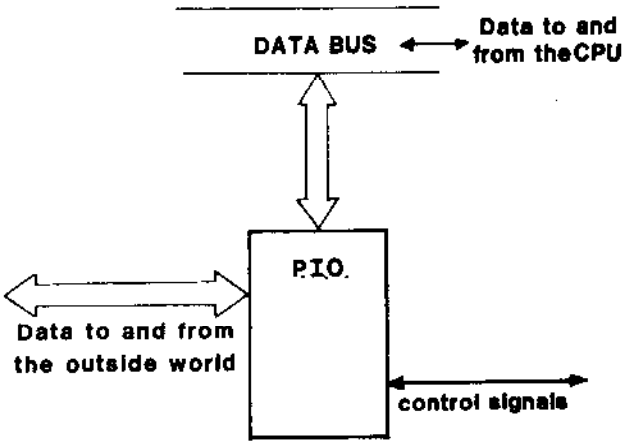
The 'working registers' are the Accumulator, A, the Flag register,F, and the six general purpose registers, B,C,D,E,H,L. All these registers have a corresponding auxilliary register, designated by a ' and all are eight bits wide.

The Program Counter, PC, holds the address of the memory location containing the next program instruction. Having sixteen bits, it can address upto 64K different locations, (see page 1-4). It will normally be updated automatically by the control circuits.

The Index Registers, IX and IY, and the Stack Pointer, SP, are sixteen bit registers whose uses are described in subsequent chapters, as is the function of the eight bit Interrupt Vector register, I. The Memory Refresh register, R, facilitates the use of a particular type of memory and is of no importance to the programmer.

2.5 Peripheral Input-Output

The PIO simply facilitates the transfer of data between the 'outside world' and the CPU via the data bus. The direction of transfer may be either way. Its use is described in Chapter 11.



Peripheral Input-Output Device

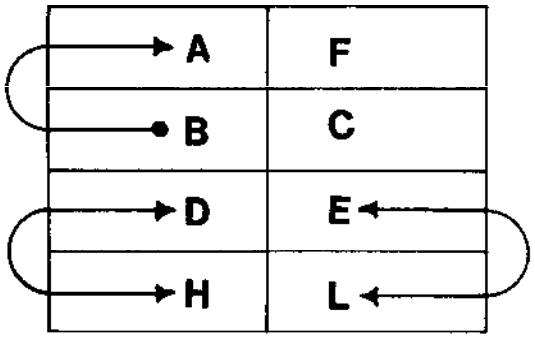
3.0 WRITING AND EXECUTING A PROGRAM

3.1 The Instruction Set

A computer instruction refers to an individual operation which can be performed as a single entity by the logic circuits inside the cpu.

The instruction set is the complete list of the computer instructions. It can be broken down into groups, each instruction within a group referring to a particular type of operation. The major groups of instructions are as follows:

3.1.1 Moving data from one cpu register to another.

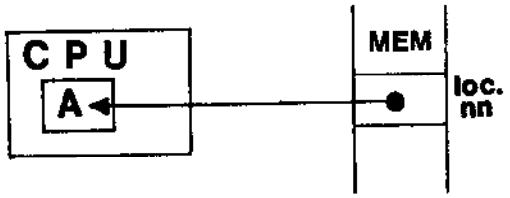


eg1 Transfer the contents of register B to register A.

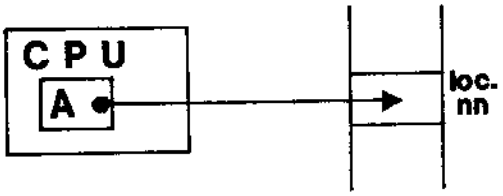
eg2 Exchange the contents of register pairs DE and HL.

3.1.2 Moving data between a cpu register and memory.

Note here the convention of using n to represent any single byte number, and nn to represent a two byte number.

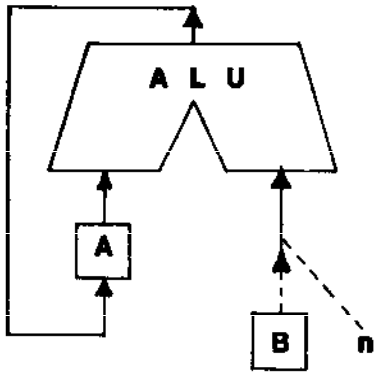


eg1 Transfer the contents of memory location nn into register A.



eg2 Transfer the contents of register A to memory location nn.

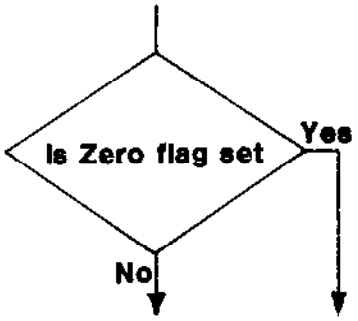
3.1.3 Arithmetic and Logic Operations



eg1 Add the number in register B to the number in register A; result in register A.

eg2 Subtract the number n from the number in register A; result in register A.

3.1.4 Program Counter Modification;Jumps



eg If the result of the previous operation was zero, continue from another part of the program, otherwise continue with the next instruction in sequence.

3.2 Writing a Program

A computer program is a list of instructions which when executed in sequence will cause the computer to carry out the required function.

Instructions reside in either the RAM or the ROM. Each instruction consists of a pattern of 0 and 1 spread over one, two, three, or even four consecutive memory locations. The required patterns of 0 and 1 are written on paper as hexadecimal numbers for convenience and they can be entered into the RAM via the NASCOM keyboard, using the keys 0 to 9,A to F.

When the program is written as a list of hexadecimal numbers it is said to be written in machine code. The machine code for some typical instructions is given below.

80 Add the number stored in register B to the number stored in register A; result in register A.

45 Transfer the number in register L to register B.

37 Set the Carry Flag to 1.

The pattern of 0 and 1 represented by the numbers in the

left-hand column, when placed in the Instruction Register (see page 2-4) of the cpu, cause the operations described alongside to be carried out.

The list of instructions, in machine code, is what has to be entered into successive memory locations in order to load the program.

Writing a program in machine code is very laborious as there are very many codes to remember. As an aid to programming each instruction can be written in mnemonic form. The mnemonics are easy to remember. After writing the program in this form it can be translated into machine code prior to loading it into the computer memory. The full list of mnemonics is given in the instruction set.

Some examples of instructions in mnemonic form, together with their machine code equivalent and a description of the instruction are given below.

3E n	LD A,n	Load register A with number n.
C6 n	ADD A,n	Add the number n to the number already in register A.
76	HALT	Halt.

These numbers represent the pattern of 0 and 1 which cause the desired operation.

These are short forms for describing the desired operation. They are called mnemonics.

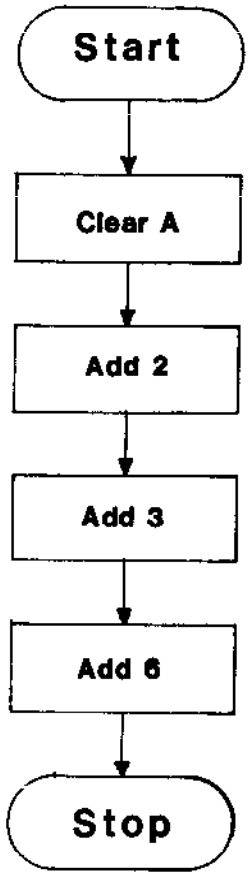
These are the desired operations.

Note that many instructions require more than one byte. These three simple instructions can be used to write a simple program.

MACHINE CODE PROGRAMMING
for the NASCOM 1 & 2

Program 3.1
Program Requirement: To add the numbers 2,3,and 6, the sum to be in register A.

The programming begins with the drawing of a flowchart. The flowchart gives a clear picture of the steps needed to achieve the programming objective.



Flowchart for Program 3.1

Each rectangle contains a description of one process, ie. a step towards achieving the programming objective. In this simple example each process consists of only one computer instruction. When it appears that the flowchart is logically correct, the program itself can be written.

PROGRAM 3.1

LOC'N	CONTENTS	OP CODE	ARGUMENTS	COMMENTS
0D00	3E 00	LD	A,00H	Clear reg A by loading it with 00H.
0D02	C6 02	ADD	A,02H	Add 2 into reg.A
0D04	C6 03	ADD	A,03H	Add 3
0D06	C6 06	ADD	A,06H	Add 6
0D08	76	HALT		Halt,sum in reg.A

(1) These columns are filled in first, following the steps indicated by the flowchart.

(2) The origin of the program, ie.the address of the first byte of the program is then decided. In this example (and the following ones) the origin is taken to be 0D00.

(3) The mnemonic code is translated into machine code using the instruction set. (Note that in the instruction set the machine code for LD A,n is given as 3E 20 since n in the instruction set is always taken to be 20H. The present program requires that n is 00H, so the code is 3E 00. Similarly, ADD A,n is given as C6 20 and this becomes C6 02, C6 03, and C6 06 in the program.)

(4) Finally the location of the first byte of each instruction is entered.The program is now ready to be entered into the NASCOM.

3.3 Loading the Program

Following power-up, the tv screen of a non-expanded NASCOM 1 will be filled with random characters. Pressing the Reset button, RS, causes the cpu to start executing the program beginning at location 0000H and this clears the screen. Expanded NASCOM 1 and NASCOM 2 have a power-on reset circuit which automatically initiates the program at 0000H. The program beginning at this location is called NAS-SYS, and it is always present in the computer, residing in the EPROM (or ROM) integrated circuits on the circuit board. The first part of NAS-SYS clears the screen, puts up a heading, and then waits for a command from the keyboard.

MACHINE CODE PROGRAMMING
for the NASCOM 1 & 2

To load the program, key the sequence:

MD00 (nl)	Open memory location 0D00.
0D00 3E 00 (nl)	NAS-SYS indicates the current contents of location 0D00. (This may be any two hex digits). The sequence 3E 00 (nl) causes 3E to be written into location 0D00, and 00 into location 0D01.
0D02 C6 02 (nl)	Write C6 into loc.0D02, 02 into loc.0D03.
0D04 C6 03 (nl)	C6 into loc.0D04, 03 into loc.0D05.
0D06 C6 06 (nl)	C6 into loc.0D06, 06 into loc.0D07.
0D08 76.(nl)	76 into loc.0D08 and terminate the loading sequence with the . before nl.

Note that the underlined characters are output by NAS-SYS. (nl) indicates the NEWLINE key on NASCOM 1 or the ENTER key on NASCOM 2. The spaces shown must be keyed in.

To check that the program is correctly loaded:

MD00 (nl)
0D00 3E(nl)
0D01 00(nl)
0D02 C6(nl)
0D03 02(nl)
0D04 C6(nl)
0D05 03(nl)
0D06 C6(nl)
0D07 06(nl)
0D08 76.(nl)

If an error is found, simply key in the correct code before (nl).

To execute the program:

ED00	Command to cause the instructions to be executed beginning with the instruction in loc. 0D00. NAS-SYS response.Halt led lit.

The program has been executed, taking only a few millionths of a second, and the cpu has halted in reponse to the HALT command in loc.0D08.

3.4 Single-stepping through a Program

It is possible to execute a program one instruction at a time. This is called single-stepping through a program and it allows the computer operations to be observed individually. To do this with Program 3.1, press RS to restart the computer (observe that the Halt led turns off) and then key the following sequence:

SD00 (nl)	Command to single step from loc.0D00
1000 0D02 00ff hhll ddee	0D02 is the content of the program counter and indicates the location of tne next instruction to be executed. 00ff shows that reg.A contains 00 and reg.F contains ff. Thus, reg.A has been loaded with 00 by the first instruction of the program.
(nl) 1000 0D04 02ff hhll ddee	The number 02 has been added to the content of reg.A, and the next instruction will be taken from loc.0D04.
(nl) 1000 0D06 05ff hhll ddee	The number 03 has been added to reg.A. Next instruction from loc.0D06.
(nl) 1000 0D08 0Bff hhll ddee	The number 06 has been added to reg.A. (05+06=0B in hexadecimal).

The next program instruction is HALT. This will not be executed when single stepping.

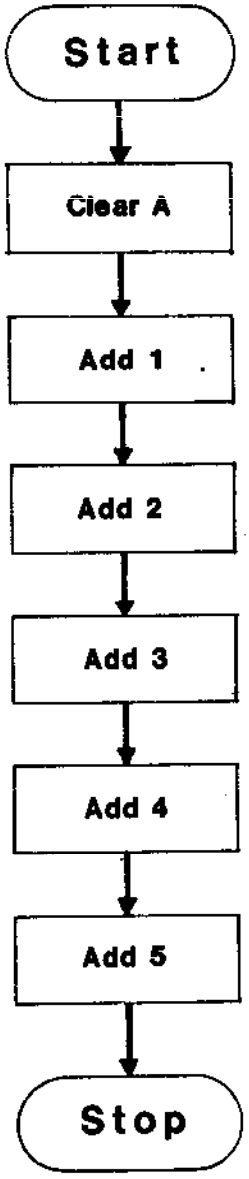
A description of the display of cpu registers is given in Appendix b; a program to write register headings on the screen is also given. This will be found useful when single-stepping through programs.

4.0 LOOPS

A program loop is a very common programming technique. It is illustrated in the following program.

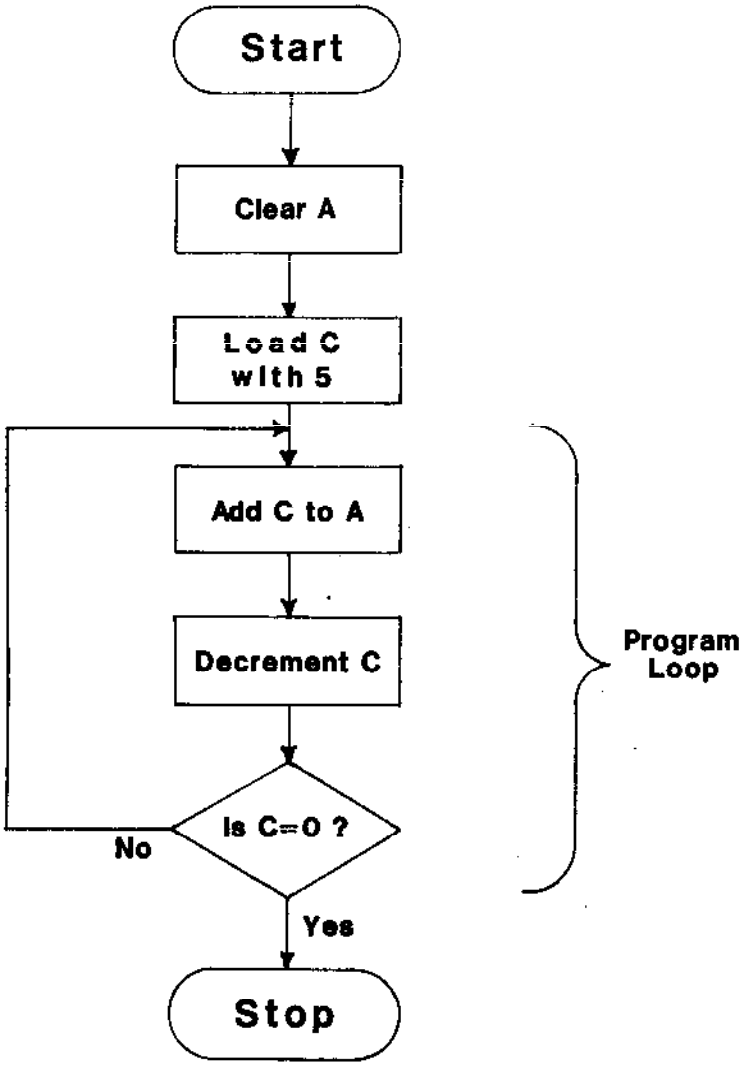
Program 4.1
Program Requirement: To add the numbers 1,2,3,4 & 5 using a loop.

The method of Program 3.1 could be extended, thus:



A Possible Flowchart for Program 4.1

While this is perfectly possible for five numbers, it would clearly be very laborious for, say, fifty numbers. A better approach is to make a program loop, as follows.



Flowchart for Program 4.1 using a loop

Here, register C is used to count the number of times that the loop has been executed. It is initially set to 5 and decremented (ie. reduced by 1) each time through the loop until it reaches zero. When at zero the decision box (the diamond shape in the flowchart) changes the program flow to send it to the instruction to stop.

PROGRAM 4.1
Program Requirement: To add the numbers 1 to 5 into register A, making use of a loop.

LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0D00	3E 00		LD	A,00H	Clear A.
0D02	0E 05		LD	C,05H	Initialise C.
0D04	81	AGAIN	ADD	A,C	Add C to A.
0D05	0D		DEC	C	Decrement C.
0D06	C2 04 0D		JP	NZ,AGAIN	Jump to AGAIN if Not Zero.
0D09	76		HALT		

Note that in this program the instruction JP NZ,AGAIN is used. This will cause the program counter to be loaded with 0D04 (=AGAIN) if the result of decrementing register C was Not Zero. Otherwise the HALT instruction will be executed. 0D04 is the actual address of the memory location given the label AGAIN. This is a very convenient device as it allows the jump location to be written in terms of the label rather than the actual memory location, which may not be known at the time of writing the instruction. Also note that in the machine code for this instruction, the low order byte is written first, followed by the high order byte.This is the convention for all such Z80 instructions.

Load the program into the computer memory and check for correct entry as was done for Program 3.1 then single-step through the program as shown below.

SD00 (nl)
1000 0D02 00ff hh11 ddee bbcc — Reg.A has been loaded with 00.Next instruction from loc.0D02.

-(nl)
1000 0D04 00ff hh11 ddee bb05 — Reg.C now loaded with 05. Next instruction from loc.0D04.

-(nl)
1000 0D05 05ff hh11 ddee bb05 — Reg.C added to reg.A. Next instruction from loc.0D05.

-(nl)
1000 0D06 05ff hh11 ddee bb04 — Reg.C decremented. Next instruction from loc. 0D06.

```
-(nl)
1000 0D04 05ff nhll ddee bb04
```

Reg. C not yet zero so the next instruction will be taken from loc.0D04. Second pass through loop begins.

```
-(nl)
1000 0D05 09ff nhll ddee bb04
-(nl)
1000 0D06 09ff nhll ddee bb03
-(nl)
1000 0D04 09ff nhll ddee bb03
```

Third pass through loop begins.

```
-(nl)
1000 0D05 0Cff nhll ddee bb03
-(nl)
1000 0D06 0Cff nhll ddee bb02
-(nl)
1000 0D04 0Cff nhll ddee bb02
```

Fourth pass through loop begins.

```
-(nl)
1000 0D05 0Eff nhll ddee bb02
-(nl)
1000 0D06 0Eff nhll ddee bb01
-(nl)
1000 0D04 0Eff nhll ddee bb01
```

Fifth pass through loop begins.

```
-(nl)
1000 0D05 0Fff nhll ddee bb01
-(nl)
1000 0D06 0Fff nhll ddee bb00  Z
-(nl)
1000 0D09 0Fff nhll ddee bb00
```

Zero flag set.

Since reg. C is now zero, next instruction will be taken from loc.0D09. The loop is terminated.

5.0 MOVING DATA BETWEEN CPU REGISTERS

The simplest movements of data are transfers from one cpu register to another. Any of the general purpose cpu registers may be loaded with the contents of any other, eg:

```
78    LD A,B          Move contents of reg.B to reg.A.

4C    LD C,H          Move contents of reg.H to reg.C.
```

The general form is:

```
      LD regd,regs    where regd is the destination register
                        and regs is the source register.
```

A full list of these instructions is given in the Instruction Set.

A cpu register may also be loaded with an immediate value,eg:

```
06 20 LD B,20H        Load reg.B with the number 20H.
```

The general form is:

```
      LD regd,n       where regd is the register and n is an
                      8 bit number.
```

These instructions are illustrated in the following program.

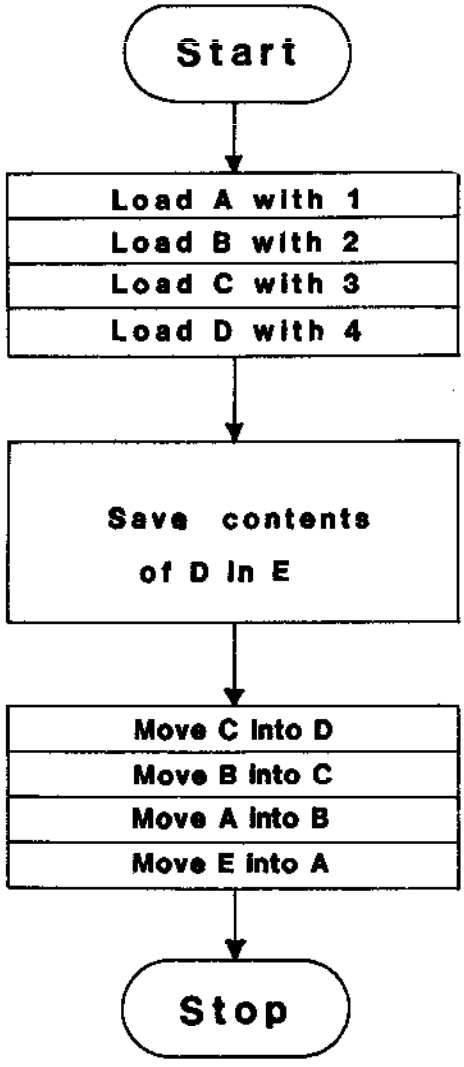
MACHINE CODE PROGRAMMING
for the NASCOM 1 & 2

PROGRAM 5.1
Program Requirement: Load registers A,B,C,D with 1,2,3,4 respectively then move reg.A into reg.B, reg.B into reg.C, reg.C into reg.D, and reg.D into reg.A.

ie.

Reg	Initially	Finally
A	1	4
B	2	1
C	3	2
D	4	3

Flowchart:



Flowchart for Program 5.1

PROGRAM 5.1

LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0D00	3E 01		LD	A,01H	Load A with 01.
0D02	06 02		LD	B,02H	Load B with 02.
0D04	0E 03		LD	C,03H	Load C with 03.
0D06	16 04		LD	D,04H	Load D with 04.
0D08	5A		LD	E,D	Save D in E.
0D09	51		LD	D,C	Move C to D.
0D0A	48		LD	C,B	Move B to C.
0D0B	47		LD	B,A	Move A to B.
0D0C	7B		LD	A,E	Move E to A.
0D0D	76		HALT		

Single-stepping through the program will show the data transfers as they occur.

6.0 MOVING DATA BETWEEN A CPU REGISTER AND MEMORY

There are four methods of moving data between a cpu register and external memory. These are:

Register Indirect Addressing

1. Transfer between any cpu register and a memory location pointed to by register pair HL, ie.register pair HL holds the 16 bit address of the required memory location.

2. Transfer between register A and a memory location pointed to by either register pair BC or DE. This is similar to method 1 but is restricted to register A.

Direct Addressing

3. Transfer between register A and a memory location whose 16 bit address is given in the instruction itself.

Indexed Addressing

4. Transfer between any cpu register and a memory location pointed to by either of the 16 bit index registers, IX and IY. This is similar to method 1 but uses IX or IY instead of HL.

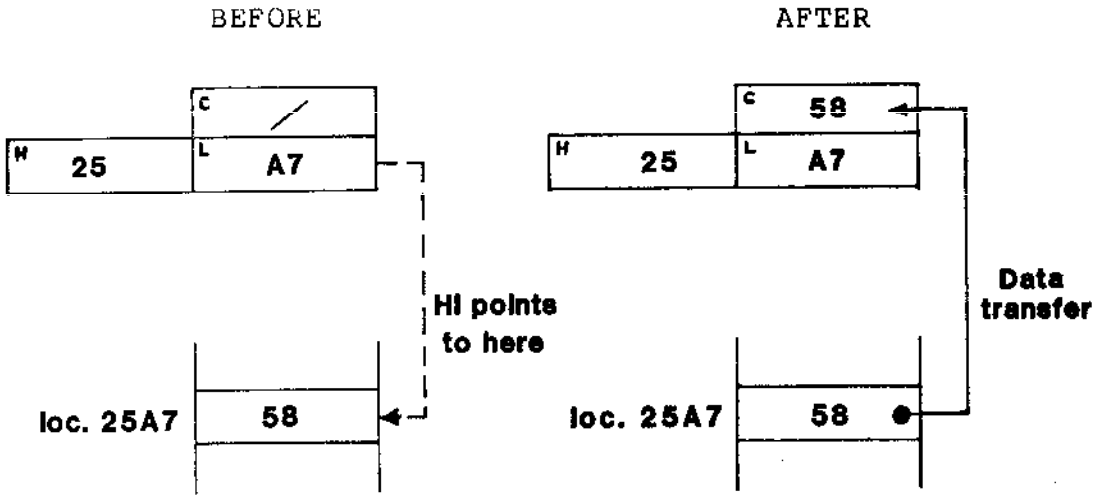
These addressing modes are illustrated in the programs which follow.

MACHINE CODE PROGRAMMING
for the NASCOM 1 & 2

6.1 Using HL as a pointer.

The contents of register pair HL (ie. registers H and L taken together to form a 16 bit word) are used to point to the memory location.

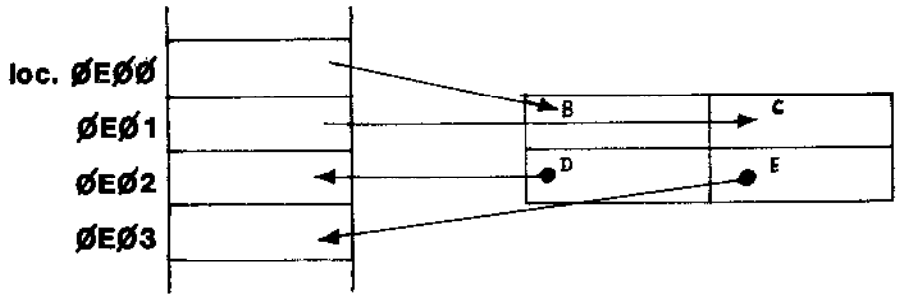
eg. LD C,(HL) Load reg. C with the contents of the memory location whose address is in the register pair HL.



LD (HL),C would cause the data transfer to be in the opposite direction, ie. from register C to the memory location whose address is in the register pair HL.

Program 6.1
Program Requirement: Load reg. B from loc.0E00, load reg.C from loc.0E01 , load loc.0E02 from reg.D,and load loc.0E03 from reg.E.

ie.



The program makes use of the INC HL instruction which simply increments the contents of register pair HL by 1.

PROGRAM 6.1

LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0D00	16 7D		LD	D,7D	
0D02	1E 7E		LD	E,7E	
0D04	21 00 0E		LD	HL,0E00	Set up pointer to first datum.
0D07	46		LD	B,(HL)	Load B from loc.0E00.
0D08	23		INC	HL	Increment HL so that it points to next datum.
0D09	4E		LD	C,(HL)	Load C from loc.0E01.
0D0A	23		INC	HL	Increment HL.
0D0B	72		LD	(HL),D	Load loc.0E02 from reg.D.
0D0C	23		INC	HL	Increment HL.
0D0D	73		LD	(HL),E	Load loc.0E03 from reg.E.
0D0E	76		HALT		
::::					
0E00	B1				Data
0E01	C1				Data
0E02	00				Reserved for contents of reg.C
0E03	00				Reserved for contents of reg.E

Note that when an instruction includes two bytes of data or an address, as in LD HL,0E00 above, the low order byte is placed first.

Load the program (not forgetting the data area from loc.0E00) and single-step through it. Observe that register B is loaded with B1, register C with C1. At the end of the program examine memory locations 0E00 to 0E03 (key ME00 (nl),etc) and note that 0E02 and 0E03 contain the same as registers D and E respectively, while 0E00 and 0E01 are unchanged.

When referring to memory locations, it is more convenient to use a label for the address rather than the address itself. Thus, location 0E00 might be given the label DAT. Then the program could be written:

```
0D04    21 00 0E            LD      HL,DAT
                        program as before
0D0E    76
::::
::::
0E00    B1          DAT     B1
::::
```

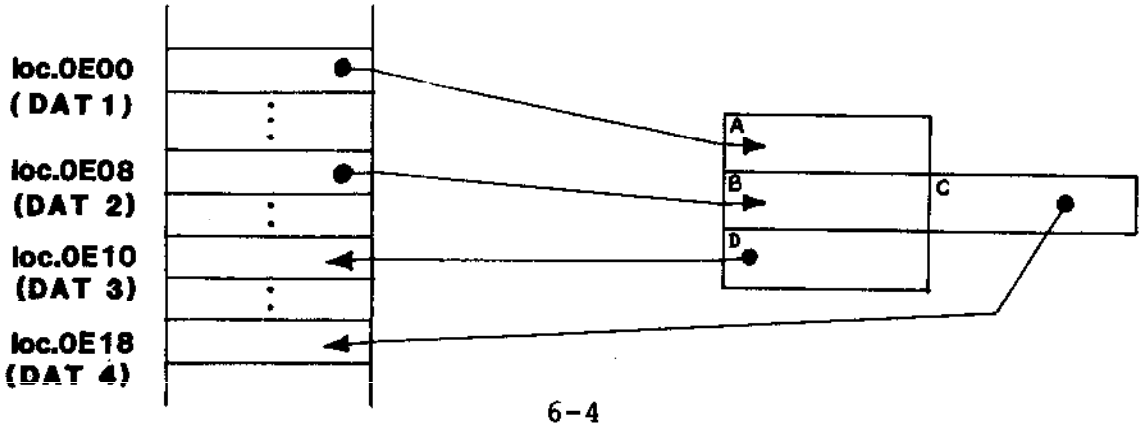
Location 0E00 is now labelled as DAT and the instruction LD HL,0E00 has become LD HL,DAT since DAT is the label given to loc.0E00.

The only rule to observe is that if a label appears in the argument column, then it must also appear elsewhere in the program in the label column. In this way the label becomes associated with a particular numerical address, just as DAT in the example above becomes associated with 0E00.

The addressing method used in Program 6.1 is convenient when the memory locations for the data are consecutive, since it is only necessary to increment HL in order to point to the next byte. If the data are not in consecutive memory locations, the register pair HL must be loaded with a new address each time.(This is tedious and a better method is described in section 6.3.) The following program illustrates this point (and also shows the use of labels).

Program 6.2
Program Requirement: Load reg.A from location DAT1(=0E00), load reg.B from location DAT2(=0E08), load location DAT3(=0E20) from reg.C, and load location DAT4(=0E18) from reg.D.

ie.



PROGRAM 6.2

LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0D00	0E CC		LD	C,CC	Load C with CC.
0D02	16 DD		LD	D,DD	Load D with DD.
0D04	21 00 0E		LD	HL,DAT1	HL points to loc.DAT1
0D07	7E		LD	A,(HL)	Move DAT1 into A.
0D08	21 08 0E		LD	HL,DAT2	HL points to loc.DAT2
0D0B	46		LD	B,(HL)	Move DAT2 into B.
0D0C	21 20 0E		LD	HL,DAT3	HL points to loc.DAT3.
0D0F	71		LD	(HL),C	Move C into DAT3.
0D10	21 18 0E		LD	HL,DAT4	HL points to DAT4.
0D13	72		LD	(HL),D	Move D into DAT4.
0D14	76		HALT		
::::					
::::					
0E00	19	DAT1	19		
::::					
0E08	27	DAT2	27		
::::					
0E18	00	DAT4	00		
::::					
0E20	00	DAT3	00		

Load the program and data area and single-step through it. Observe the movement of data into registers A and B. Then examine memory locations 0E18 and 0E20 and observe that they contain the same as registers D and C respectively.

6.2 Using BC and DE as pointers

These instructions are very similar to those used in Program 6.1 and 6.2. Register pairs BC or DE are used as pointers, instead of HL, and these transfers always involve register A.

eg. LD A,(BC) Load reg.A with the data in the memory location whose address is in reg. pair BC.

LD A,(DE)

LD (BC),A — Load the memory location whose address is in reg. pair BC with the data in reg. A.

LD (DE),A

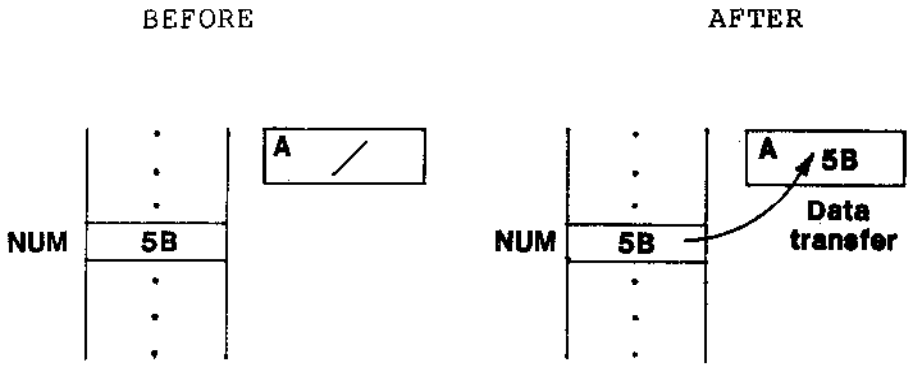
6.3 Using Direct Addressing

In direct addressing the actual address of the memory location (nn) is contained within the instruction. The transfers must always involve register A.

eg. LD A,(nn) — Load register A from location nn.

LD (nn),A — Load location nn from register A.

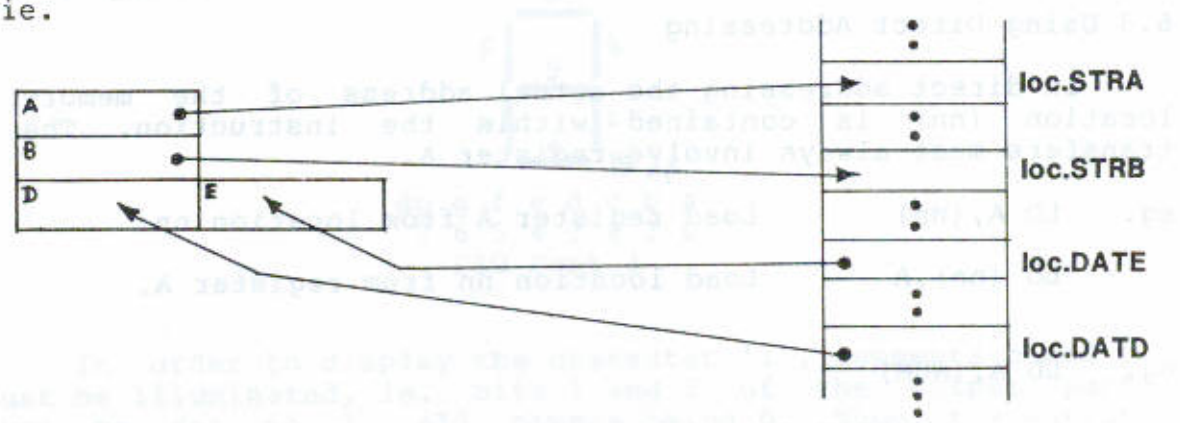
eg. LD A,(NUM)



This addressing mode is most often used to transfer a byte from memory into the cpu. Once the byte is in register A it may be transferred to any other cpu register using a LD reg,A instruction.

Program 6.3
Program Requirement: To store the contents of register A in STRA(=0E00), register B in STRB(=0E08), and to load register D from DATD(=0E18), register E from DATE(=0E10).

ie.



PROGRAM 6.3

LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0D00	3E AA		LD	A,AA	Load A with AA.
0D02	06 BB		LD	B,BB	Load B witn BB.
0D04	32 00 0E		LD	(STRA),A	Save A in STRA.
0D07	78		LD	A,B	Move B into A.
0D08	32 08 0E		LD	(STRB),A	Transfer to STRB.
0D0B	3A 18 0E		LD	A,(DATD)	Move DATD to A.
0D0E	57		LD	D,A	Transfer to D.
0D0F	3A 10 0E		LD	A,(DATE)	Move DATE into A.
0D12	5F		LD	E,A	Transfer to E.
0D13	76		HALT		
::::					
0E00	00	STRA	00		
::::					
0E08	00	STRB	00		
::::					
0E10	1E	DATE	1E		
::::					
0E18	1D	DATD	1D		

Load the program and single-step through it. Then examine memory locations 0E00 and 0E08. Check that the required data transfers have taken place.

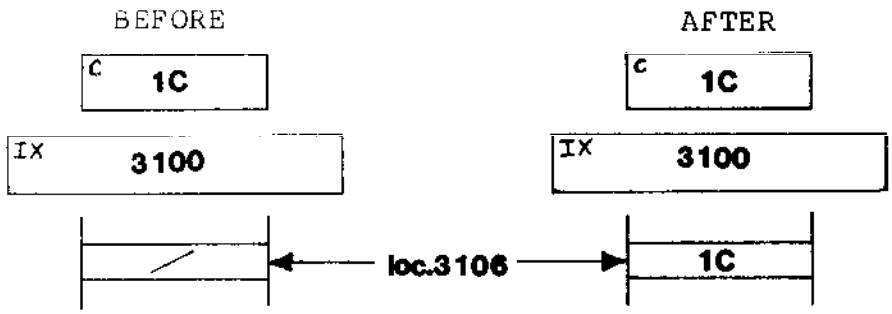
6.4 Using the Index Registers

The index registers IX and IY are 16 bits wide and are used as pointers similar to the use of register pair HL. However, an 8 bit displacement may be added to the contents of the index register to obtain the effective address.

The instructions have the form:

LD (IX+disp),reg

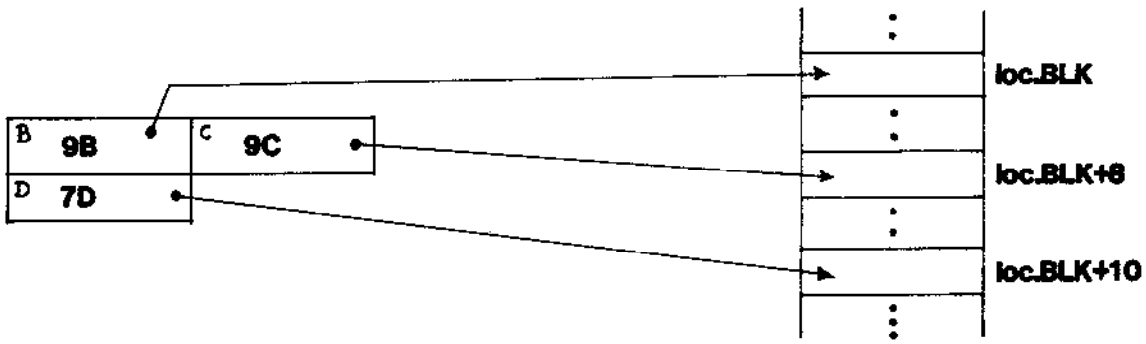
eg. LD (IX+6),C



The instruction LD (IX+6),C obtains the sum 3100+6 and loads the contents of register C (=1C) into memory location 3106.

Program 6.4
Program Requirement: Load register B with 9B and register C with 9C, and register D with 7D. Then transfer the contents of register B to location BLK(=0E10), transfer register C to location BLK+8(=0E18), and transfer register D to location BLK+10(=0E20).

ie.



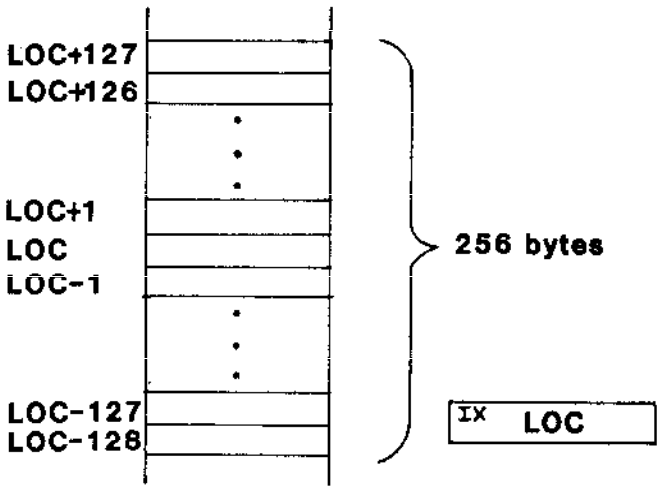
PROGRAM 6.4

LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0D00	06 9B		LD	B,9B	Load B with 9B.
0D02	0E 9C		LD	C,9C	Load C with 9C.
0D04	16 7D		LD	D,7D	Load D with 7D.
0D06	DD 21 10 0E		LD	IX,BLK	IX points to loc.BLK.
0D0A	DD 70 00		LD	(IX),B	Move reg.B to loc.BLK.
0D0D	DD 71 08		LD	(IX+8),C	Move reg.C to loc. BLK+8.
0D10	DD 72 10		LD	(IX+10),D	Move reg. D to BLK+10
0D13	76		HALT		Halt.
::::					
::::					
::::					
0E10	00	BLK	00		Reserved for data from reg.B.
::::					
0E18	00	BLK+8	00		Reserved for data from reg.C.
::::					
0E20	00	BLK+10	00		Reserved for data from reg. D.

Note: At location 0D04 IX is loaded with BLK(=0E10) and subsequent memory references are made relative to this location by including the displacements 08 and 10 in the instructions.

Single step through the program and then examine locations 0E10, 0E18, and 0E20 to check that the data transfers have taken place. Location 0E10 should contain 9B, 0E18 should contain 9C, and 0E20 should contain 7D.

Since the displacement is an eight bit byte it may take any value between 80H(=-128d) and 7F(=127d). This allows ready access to data within a block of 256 bytes, as shown below:

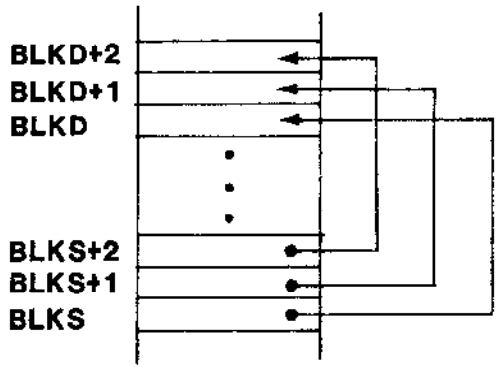


6.4.1 The use of index registers to process data blocks.

The use of index registers is most efficient when dealing with contiguous locations in memory, ie. a block. This is illustrated in the following program.

Program 6.5
Program Requirement: To move the three bytes in locations BLKS to BLKS+2 to locations BLKD to BLKD+2.

ie.



In the program IX points to the source location and IY points to the destination location.

PROGRAM 6.5

```
LOC'N  CONTENTS    LABEL   OP CODE  ARGUMENTS   COMMENTS
0D00   DD 21 00 0F         LD       IX,BLKS     IX points to
                                                BLKS.
0D04   FD 21 00 0E         LD       IY,BLKD     IY points to
                                                BLKD.
0D08   DD 46 00            LD       B,(IX)      Load B with
                                                first data
                                                byte,
0D0B   FD 70 00            LD       (IY),B      ..and transfer.
0D0E   DD 23               INC      IX          Increment
0D10   FD 23               INC      IY          ..both
                                                pointers.
0D12   DD 46 00            LD       B,(IX)      Get second
                                                byte,
0D15   FD 70 00            LD       (IY),B      ..and transfer.
0D18   DD 23               INC      IX
0D1A   FD 23               INC      IY
0D1C   DD 46 00            LD       B,(IX)      Get third byte,
0D1F   FD 70 00            LD       (IY),B      ..and transfer.
0D22   76                  HALT                 Halt.
::::
::::
::::
0E00   00          BLKD    00                   Locations
0E01   00                  00                   reserved
0E02   00                  00                   for data.
::::
::::
0F00   1A          BLKS    1A                   Data
0F01   2B                  2B                   to be
0F02   3C                  3C                   transferred
```

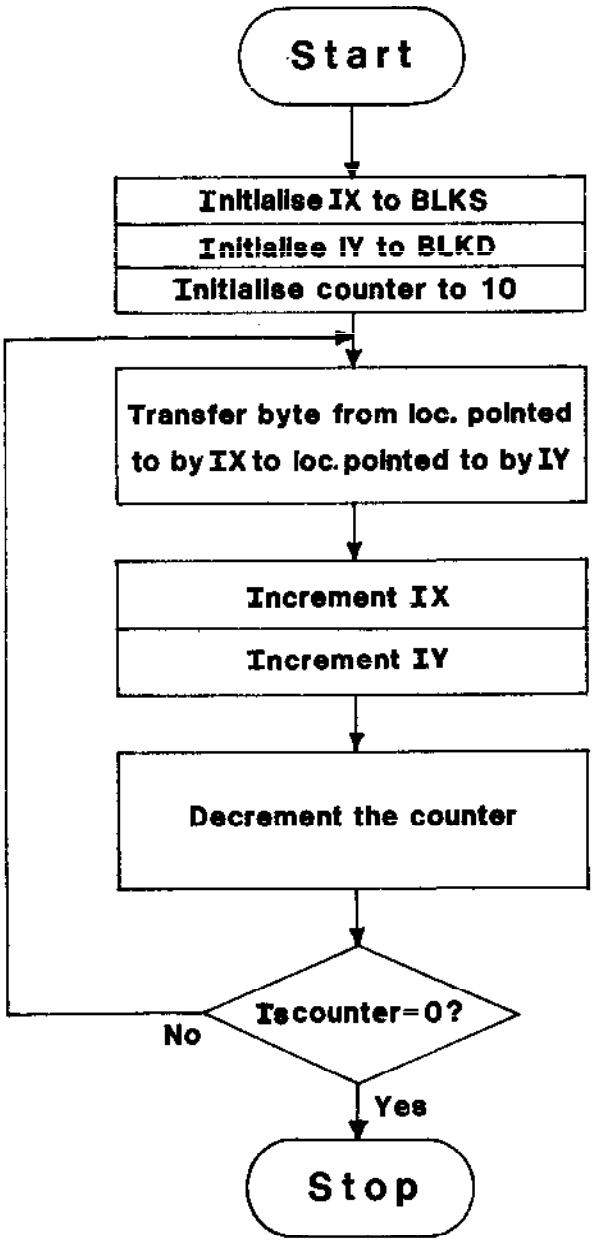
Single step from location 0D00 taking particular note of the contents of IX and IY Then examine locations 0E00 to 0E02 and check that the data transfers have taken place.

The sequence:

```
LD B,(IX)
LD (IY),B
INC IX
INC IY
```

is repeated for each byte to be transferred. Clearly, for a block of many bytes this approach is inefficient and a loop is called for, as follows:

Program 6.6
Program Requirement: To transfer sixteen bytes from data area beginning with BLKS to data area beginning with BLKD.



Flowchart for Program 6.6

PROGRAM 6.6

```
LOC'N  CONTENTS    LABEL   OP CODE  ARGUMENTS  COMMENTS
0D00   DD 21 80 0E         LD       IX,BLKS    Point to BLKS.
0D04   FD 21 00 0E         LD       IY,BLKD    Point to BLKD.
0D08   0E 10               LD       C,10H      Set C to 10H.
0D0A   DD 46 00    LOOP    LD       B,(IX)     Load byte.
0D0D   FD 70 00            LD       (IY),B     Transfer byte.
0D10   DD 23               INC      IX
0D12   FD 23               INC      IY
0D14   0D                  DEC      C
0D15   C2 0A 0D            JP       NZ,LOOP
0D18   DF 5B               .MRET                Return to NAS-
                                                SYS.
::::
::::
::::
0E00               BLKD                         Sixteen
:                                               :
:                                               :
0E0F                                            bytes.
::::
::::
0E80               BLKS                         Sixteen
:                                               :
:                                               :
0E8F                                            bytes.
```

The instruction .MRET (DF 5B) is a special NAS-SYS subroutine to return program control to NAS-SYS. It is used instead of the HALT instruction and is more convenient since it avoids having to reset the NASCOM. NAS-SYS subroutines are referred to in more detail in Chapter 10.
Load the program, tabulate BLKS (locations 0E80 to 0E8F) and also BLKD (locations 0E00 to 0E0F). Then execute the program and tabulate BLKD again.
ie. key the sequence:

```
TE80 E8F
```

BLKS

```
TE00 E0F
```

BLKD	BLKD before transfer.

```
ED00
TE00 E10
```

BLKD	BLKD after transfer.

Check that BLKD after transfer is the same as BLKS.

6.5 Other data transfer instructions

1. Block transfer instructions.

   The Z80 has some special instructions which greatly facilitate block transfers.

2. 16bit moves.

   The LD instructions described so far move a single byte. There are several instructions for moving two bytes. These involve register pairs BC, DE, and HL, the stack pointer SP, and the index registers IX and IY.

3. Exchange instructions.

   These allow the exchange of data between the main cpu general purpose registers and the primed set.

   ```
   eg.  EX AF,AF'     Exchange registers AF and AF'.
        EXX           Exchange registers BC,DE,HL
                      with BC',DE',HL' respectively.
   ```

7.0 ARITHMETICAL AND LOGICAL OPERATIONS

The arithmetical operations include ADD, SUBtract, and ComPare, while the logical operations include OR, AND,ComPLement, and XOR.

7.1 Arithmetical Operations

Arithmetic may be carried out using either pure binary or decimal numbers.

7.1.1 Binary Addition and Subtraction

eg.	ADD A,B	Add contents of register B to the contents of register A.
	ADD A,(HL)	Add the contents of the memory location pointed to by register pair HL to register A.
	SUB C	Subtract the contents of register C from the contents of register A.
	SUB (HL)	Subtract the contents of the memory location pointed to by register pair HL from the contents of register A.

One of the operands must always be register A and the result always goes into register A. The second operand may be fetched using any of the addressing modes described in Chapter 6. The simple addition of two 8 bit numbers is shown in the following program.

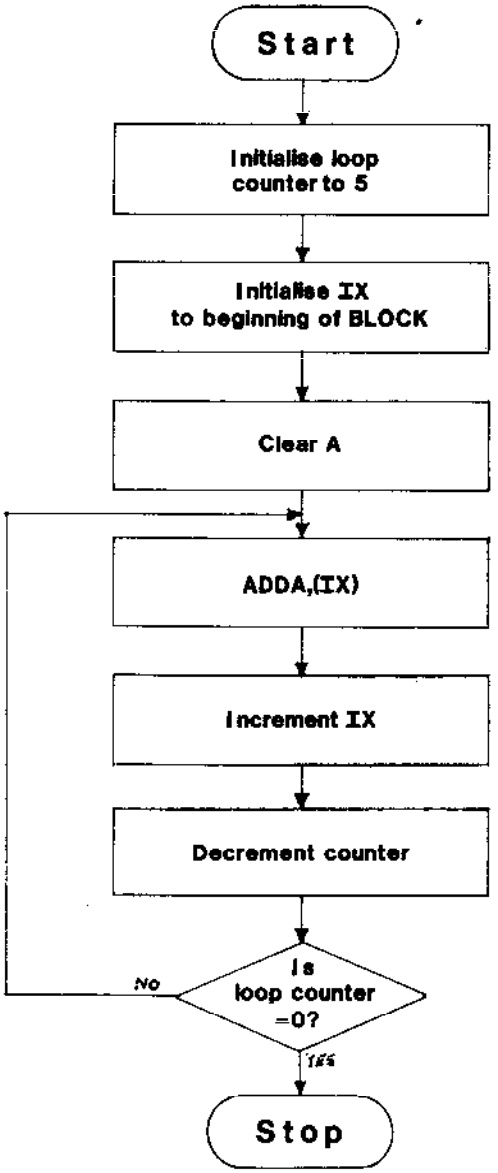
Program 7.1
Program Requirement: To add the numbers in registers B and C, result in register A.

LOC'N	CONTENTS	OP CODE	ARGUMENTS	COMMENTS
0D00	97	SUB	A	Clear A.
0D01	06 13	LD	B,13H	Load B with 13H.
0D03	0E 27	LD	C,27H	Load C with 27H.
0D05	80	ADD	A,B	Add B to A.
0D06	81	ADD	A,C	Add C to A.
0D07	76	HALT		

Single-stepping through this program will show the addition taking place. Remember that 13H + 27H = 3AH.

The next program shows how a block of numbers may be added.

Program 7.2
Program Requirement: To add a block of five numbers, sum in register A.



Flowchart for Program 7.2

In the program register B is used as the loop counter and index register IX points to successive data bytes.

PROGRAM 7.2

```
LOC'N  CONTENTS     LABEL   OP CODE  ARGUMENTS   COMMENTS
0D00   06 05                LD       B,5H        Initialise B to
                                                 5.
0D02   DD 21 00 0E          LD       IX,BLOCK    IX points to
                                                 start of data
                                                 block.
0D06   97                   SUB      A           Clear A.
0D07   DD 86 00     LOOP    ADD      A,(IX)      Add byte to A.
0D0A   DD 23                INC      IX          Point to next
                                                 data byte.
0D0C   05                   DEC      B           Decrement
                                                 counter.
0D0D   C2 07 0D             JP       NZ,LOOP     Loop done?
0D10   DF 5B                .MRET                Return to
                                                 keyboard.
::::
::::
0E00   04           BLOCK   04                   4d
0E01   10                   10                   16d
0E02   02                   02                   2d
0E03   01                   01                   1d
0E04   20                   20                   32d
```

By single-stepping through the program the sum can be seen to build up in register A. Note that the sum is 37H(=55d).

The data bytes could be negative, eg. if the first data byte (in loc.0E00) is changed to F9H(=-7d) the sum 2CH(=44d) will be obtained.

If the sum is greater than 7FH(=+127d) or less than 80H(=-128d), it will be out of the range of the 8 bit register, and the overflow flag will be set to 1.

The overflow flag,P/V, is a single bit in the Flag register (register F of the cpu). Its state is normally used to control the program sequence. (The zero flag, Z, has previously been used in the JP NZ,LOOP instruction.) The P/V flag may be used to indicate an error due to overflow in the previous program by adding the instructions shown below.

```
        ADD         A,(IX)
        JP          PE,ERROR            Jump to ERROR if
                                        overflow flag is
                                        set.
        INC         IX
        :
        :
ERROR   .ERRM                           Here if overflow.
        .MRET
```

The code .ERRM (DF 6B) is a NAS-SYS command to display 'Error' on the screen. The program thus becomes:

PROGRAM 7.3
Program Requirement: As for Program 7.2, but with overflow detection.

```
LOC'N  CONTENTS    LABEL   OP CODE  ARGUMENTS  COMMENTS
0D00   06 05               LD       B,5H
0D02   DD 21 00 0E         LD       IX,BLOCK
0D06   97                  SUB      A
0D07   DD 86 00    LOOP    ADD      A,(IX)
0D0A   EA 20 0D            JP       PE,ERROR   Jump to ERROR
                                                if overflow has
                                                occurred.
0D0D   DD 23               INC      IX
0D0F   05                  DEC      B
0D10   C2 07 0D            JP       NZ,LOOP
0D13   DF 5B               .MRET
::::
0D20   DF 6B       ERROR   .ERRM
0D22   DF 5B               .MRET
::::
::::
::::
0E00   38          BLOCK   38
0E01   F9                  F9
0E02   4B                  4B
0E03   05                  05
0E04   10                  10
```

Execute the program first and observe that the Error message is displayed. Single-stepping through the program shows the contents of register A successively as 38, 31, 7C, 81 at which point there is a jump to ERROR since 81H is -127d which is overflow.

7.1.2 INCrement and DECrement

The increment instruction simply adds 1 to the existing value, while the decrement instruction subtracts 1.

Any of the addressing modes explained in Chapter 6 may be used. Examples can be found in the various programs in this guide.

7.1.3 NEGate

NEG forms the twos complement of the contents of register A, ie. the number in A is multiplied by -1. This is illustrated in the following program.

Program 7.4
Program Requirement: To form the negative of a number.

LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0D00	0E 00		LD	C,00H	
0D02	79	LOOP	LD	A,C	
0D03	ED 44		NEG		
0D05	47		LD	B,A	
0D06	0C		INC	C	
0D07	C3 02 0D		JP	LOOP	

Single-step through the program and observe that, after the LD B,A , registers B and C are respectively -x and +x in the twos complement notation described in Chapter 1.

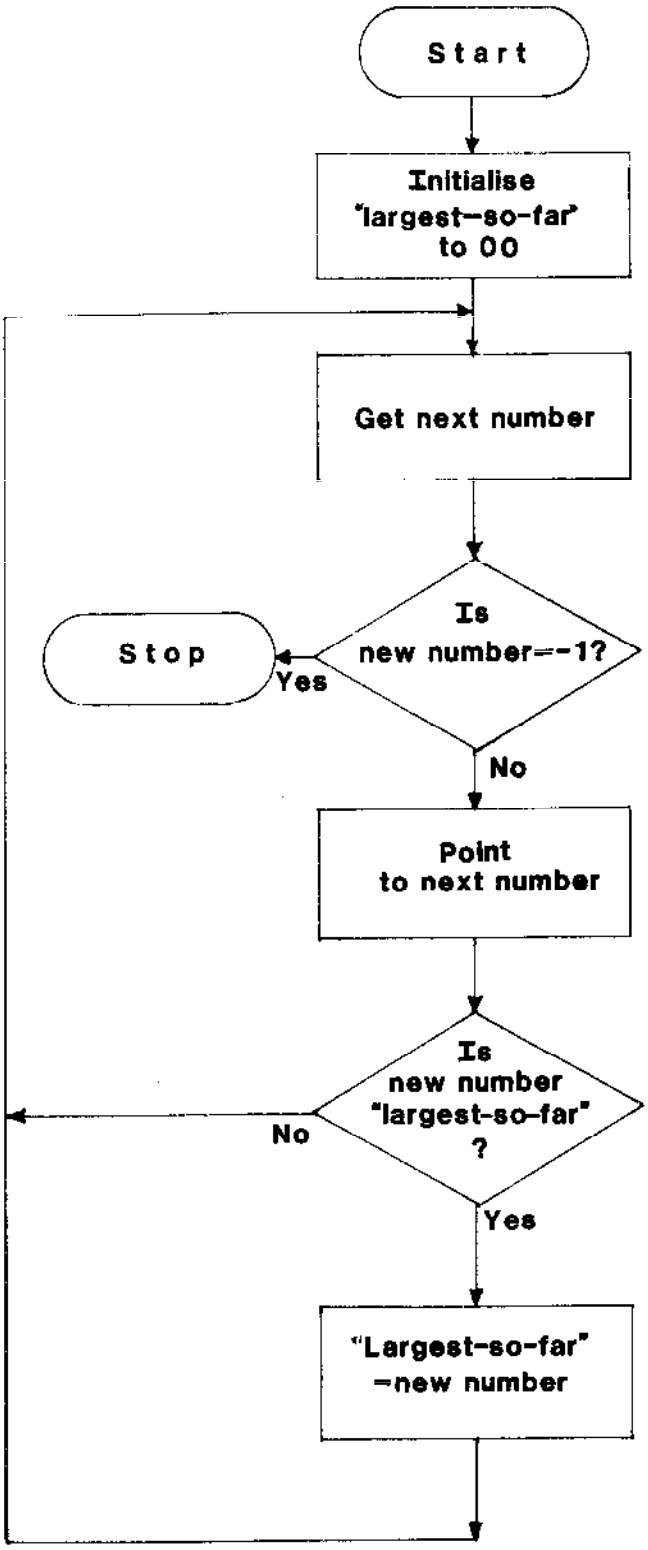
7.1.4 ComPare

This instruction is identical to the SUB instruction except that the contents of register A are unchanged. The flags in the Flag register, however, are changed. These flags are then normally used by a subsequent instruction to determine whether or not a jump should occur.

Program 7.5
Program Requirement: To find the largest number in a block of positive numbers. The last number in the block is -1.

Note that rather than specifying the length of the block and counting the bytes to determine the end, the block is delimited by a number (-1) which will never occur as a data byte. This allows the data block to be changed without having to make any change in the program.

The largest-number-so-far is kept in register B, and register pair HL points to successive data bytes in the block.



Flowchart for Program 7.5

PROGRAM 7.5

```
LOC'N  CONTENTS   LABEL   OP CODE  ARGUMENTS   COMMENTS
0D00   21 00 0E           LD       HL,LIST     HL points to
                                                LIST.
0D03   06 00              LD       B,00H       Clear B.
0D05   7E         LOOP    LD       A,(HL)      Get new number.
0D06   FE FF              CP       FF          Compare with -
                                                1.
0D08   CA 14 0D           JP       Z,END       Jump if equal.
0D0B   23                 INC      HL          Bump pointer.
0D0C   B8                 CP       B           Is new >
                                                largest?
0D0D   FA 05 0D           JP       M,LOOP      If no, jump.
0D10   47                 LD       B,A
0D11   C3 05 0D           JP       LOOP
0D14   DF 5B      END     .MRET
::::
::::
::::
0E00   05         LIST    05
0E01   49                 49
0E02   32                 32
0E03   48                 48
0E04   70                 70
0E05   61                 61
0E06   FF                 FF
```

Single-step through the program and observe that the largest-number-so-far is in register B.

7.1.5 Multiple Precision Arithmetic

This section may be ommitted at a first reading.

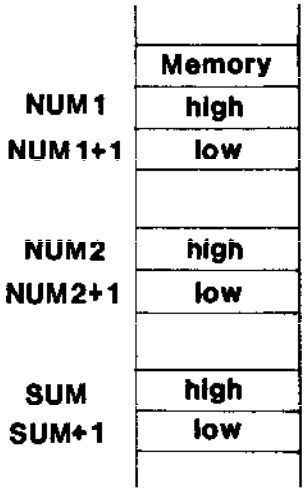
The arithmetic operations described so far involve only single byte numbers thereby limiting the number range to +127 to -128d. If a larger number range is required the number must occupy more than one byte and this is then termed multiple precision arithmetic.

Double byte, ie. 16 bit numbers provide a number range from +32,768 to -32,768, and four byte, ie. 32 bit, numbers provide a number range greater than 2,000,000,000.

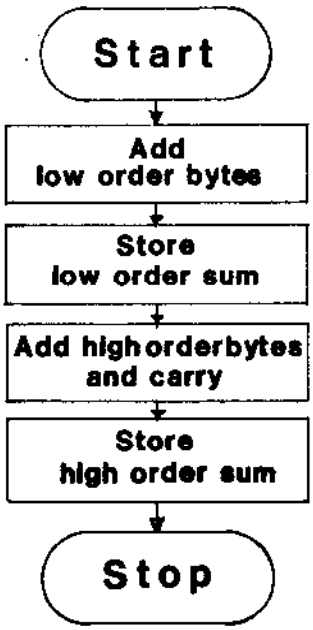
The Z80 has some instructions for performing 16 bit arithmetic. However, before considering them, the general principles involved in handling multi-byte numbers are illustrated in Program 7.6 using only the single-byte arithmetic instructions.

Program 7.6
Program Requirement: To add two 16 bit numbers, using only single-byte arithmetic instructions.

Let the number be stored so that the high order byte of the first number is stored in location NUM1, and the low order byte in location NUM1+1. Similarly, the second number is stored in locations NUM2 and NUM2+1, and the sum in SUM and SUM+1.
ie.



Addition proceeds by first adding the two low order bytes and placing the sum in location SUM+1. Any carry out from bit 7 produced by this operation will be stored in the carry flag,C. When the two high order bytes are added, the carry also has to be added. This is accomplished by the ADC (add with carry) instruction.



Flowchart for Program 7.6

PROGRAM 7.6

```
LOC'N  CONTENTS    LABEL    OP CODE  ARGUMENTS   COMMENTS
0D00   DD 21 01 0E          LD       IX,NUM1+1   Point to low order byte of NUM1.
0D04   FD 21 03 0E          LD       IY,NUM2+1   Ditto for NUM2.
0D08   21 05 0E             LD       HL,SUM1+1   Ditto for SUM.
0D0B   DD 7E 00             LD       A,(IX)      Get low byte of NUM1.
0D0E   FD 86 00             ADD      A,(IY)      Add low byte of NUM2.
0D11   77                   LD       (HL),A      Store low byte of SUM.
0D12   2B                   DEC      HL
0D13   DD 7E FF             LD       A,(IX-1)
0D16   FD 8E FF             ADC      A,(IY-1)
0D19   77                   LD       (HL),A
0D1A   DF 5B                .MRET
::::
::::
::::
0E00   4A          NUM1     4A
0E01   38          NUM1+1   38
0E02   2A          NUM2     2A
0E03   F8          NUM2+1   F8
0E04   ..          SUM      ..
0E05   ..          SUM+1    ..
```

Single-step through the program and note that first of all 38 and F8 are added to produce 30 (and Carry =1), then 4A,2A,and the carry are added to produce 75, ie. 4A38H+2AF8H=7530H (or 19000d+11000d=30000d). The sum is stored in locations 0E04 and 0E05.

The numbers, of course, may be negative: change NUM2 to EC78 and repeat to obtain 4A38H+EC78H=36B0H
(or 19000d+ -5000d=14000d).

Sixteen bit subtraction may be obtained by modifying Program 7.6:-
change loc. 0D0E ADD A,(IY) to FD 96 00 SUB (IY) and
change loc. 0D16 ADC A,(IY-1) to FD 9E FF SBC A,(IY-1).

Set the numbers back to 4A38 and 2AF8 and run the modified program to obtain 4A38H-2AF8H=1F40H (or 19000d-11000d=8000d). Again, change the numbers to 4A38 and EC78 to obtain 4A38H-EC78H=5DC0H (or 19000d--5000d= 24000d).

The sixteen bit add and subtract instructions utilise register pair HL as an accumulator, ie.all the instructions involve HL.

PROGRAM 7.7
Program Requirement: To add two 16 bit numbers, using the 16 bit arithmetic instructions.

PROGRAM 7.7

LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0D00	2A 00 0E		LD	HL,(NUM1)	Put NUM1 into HL.
0D03	ED 5B 02 0E		LD	DE,(NUM2)	Put NUM2 into DE.
0D07	19		ADD	HL,DE	Add
0D08	22 04 0E		LD	(SUM),HL	Store result.
0D0B	DF 5B		.MRET		
::::					
::::					
::::					
0E00	38	NUM1	38		NUM1 is
0E01	4A		4A		4A38
0E02	F8	NUM2	F8		NUM2 is
0E03	2A		2A		2AF8
0E04	..	SUM	..		
0E05	..		..		

Note that the double byte numbers have to be stored low byte first. Single-step through the program and observe that 4A38H+2AF8H=7530H.

Program 7.8
Program Requirement: To subtract two 16 bit numbers, using the 16 arithmetic instructions.

Subtraction requires a little more care since all the instructions involve the carry flag. In this program the carry flag is set to 0 immediately before the subtraction; this is achieved by the instruction OR A.

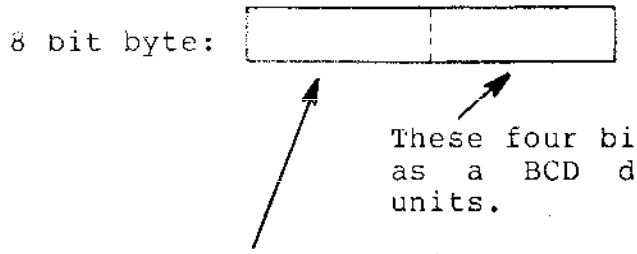
PROGRAM 7.8

LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0D00	2A 00 0E		LD	HL,(NUM1)	Put NUM1 into HL.
0D03	ED 4B 02 0E		LD	BC,(NUM2)	Put NUM2 into BC.
0D07	B7		OR	A	Clear carry flag.
0D08	ED 42		SBC	HL,BC	Subtract
0D0A	22 04 0E		LD	(DIFF),HL	
0D0D	DF 5B		.MRET		
::::					
::::					
::::					
0E00	38	NUM1	38		NUM1 is
0E01	4A		4A		4A38
0E02	F8	NUM2	F8		NUM2 is
0E03	2A		2A		2AF8
0E04	..	DIFF	..		
0E05	..		..		

Single-step through the program and observe that 4A38-2AF8=1F40.

7.1.6 Decimal Arithmetic

The decimal digits 0 to 9 are represented in Binary Coded Decimal (BCD) simply by their corresponding pure binary equivalent, as described in Chapter 1.6. Since only four bits are required, a single byte can hold two digits as shown below.



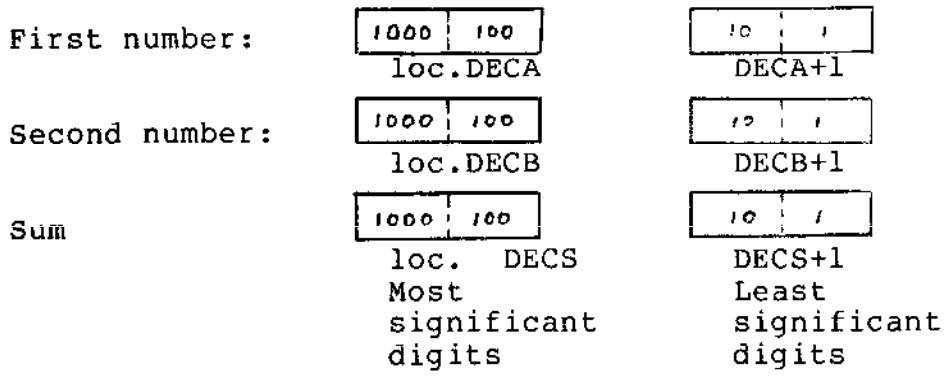
Program 7.6
Program Requirement: To add 47d and 25d.

```
LOC'N  CONTENTS    LABEL   OP CODE   ARGUMENTS   COMMENTS
0D00   06 47               LD   .    B,47
0D02   3E 25               LD        A,25
0D04   80                  ADD       A,B
0D05   27                  DAA
0D06   32 00 0E            LD        (SUM),A
0D09   DF 5B               .MRET
::::
::::
::::
0E00   ..          SUM     ..
```

Single-stepping through the program will show that after the ADD A,B instruction, register A holds 6C since 47H+25H=6CH. The Decimal Adjust A (DAA) instruction converts the hexadecimal sum to decimal yielding 72d.

Program 7.6 performs the addition of two-digit numbers and if the sum is greater than 99 overflow occurs. Decimal numbers of arbitrary length may be processed simply by using more bytes to hold the number. This is illustrated in the following example.

Program 7.7
Program Requirement: To add two four-digit numbers.



The most significant two digits of the first number are stored in memory location DECA and the least two significant two digits in DECA+1. The second number is similarly stored, and two bytes at locations DECS and DECS+1 are reserved for the sum.

PROGRAM 7.7

LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0D00	3A 01 0E		LD	A,(DECA+1)	Get ls byte of 1st no.
0D03	47		LD	B,A	Save in B.
0D04	3A 03 0E		LD	A,(DECB+1)	Get ls byte of 2nd no.
0D07	80		ADD	A,B	Add
0D08	27		DAA		Adjust.
0D09	32 05 0E		LD	(DECS+1),A	Store result.
0D0C	3A 00 0E		LD	A,(DECA)	Get ms byte of 1st no.
0D0F	47		LD	B,A	
0D10	3A 02 0E		LD	A,(DECB)	Get ms byte of 2nd no.
0D13	88		ADC	A,B	Add with carry.
0D14	27		DAA		Adjust.
0D15	32 04 0E		LD	(DECS),A	Store result.
0D18	DF 5B		.MRET		
:::					
::::					
::::					
0E00	29	DECA	29		
0E01	47		47		
0E02	52	DECB	52		
0E03	95		95		
0E04	..	DECS	..		
0E05	..		..		

Note the use of the ADC instruction in location 0D13. Any carry out from the tens digit is automatically stored in the carry flag. When the hundreds digits are added the carry

must also be added. This is done by the ADC,add with carry, instruction.

If subtraction is being performed the corresponding instruction is SBC.

The program may be extended to allow arithmetic with decimal numbers of arbitrary length.

7.2 Logical Operations

7.2.1 Logical AND

Definition:

```
    0         0         1         1
AND 0     AND 1     AND 0     AND 1
    -         -         -         -
    0         0         0         1
```

ANDing 8 bits:

This is done bit by bit,eg.

```
          0 0 0 0 1 1 1 1 = 0F
      AND 0 1 1 0 0 0 1 1 = 63
          -----------------
          0 0 0 0 0 0 1 1 = 03
```

This example shows how the AND can be used to mask out selected bits of a byte. The 'mask' in this example is 0F and consequently only the lowest four bits of the other byte are allowed to drop through. This is often used to set unwanted bits in a byte to zero.

7.2.2 Logical OR

Definition:

```
    0         0         1         1
OR  0     OR  1     OR  0     OR  1
    -         -         -         -
    0         1         1         1
```

ORing 8 bits:

This is done bit by bit,eg.

```
          0 0 0 0 0 0 1 1 = 03
       OR 0 0 1 0 1 1 0 0 = 2C
          -----------------
          0 0 1 0 1 1 1 1 = 2F
```

This example shows how the OR may be used to merge two bit patterns.

7.2.3 Logical NOT (CPL)

Definition: NOT 0=1 NOT 1=0

Complementing 8 bits,eg.

```
          0 0 1 0 1 0 0 1 = 29
    CPL   1 1 0 1 0 1 1 0 = D6
```

7.2.4 Logical XOR

Definition:

```
    0         0         1         1
XOR 0     XOR 1     XOR 0     XOR 1
    -         -         -         -
    0         1         1         0
```

XORing 8 bits:

This is done bit by bit:

```
egl.        0 0 1 0 1 1 1 1 = 2F
        XOR 0 0 0 0 0 1 1 0 = 06
            -----------------
            0 0 1 0 1 0 0 1 = 29


eg2.        0 0 0 0 1 1 1 1 (a)
        XOR 1 0 0 1 0 1 1 0 (b)
            ---------------
            1 0 0 1 1 0 0 1 (c)
```

This example shows that bits XORed with 0 are unchanged, while bits XORed with 1 are complemented. Thus the most significant nibble (four bits) of (c) is the same as the corresponding nibble of (b), while the least significant nibble of (c) is the complement of the corresponding nibble of (b). The XOR is very useful when a bit has to be toggled, ie.alternated between 0 and 1.

Program 7.8
Program Requirment: To illustrate AND,OR,CPL, and XOR.

```
LOC'N  CONTENTS   LABEL   OP CODE  ARGUMENTS  COMMENTS
0D00   3E 0F              LD       A,0F
0D02   E6 63              AND      63
OD04   F6 2C              OR       2C
0D06   EE 06              XOR      06
0D08   2F                 CPL
0D09   DF 5B              .MRET
```

By single-stepping through the program, the result of each logical operation can be seen in register A.

Thus:

```
      0 0 0 0 1 1 1 1 = 0F
AND   0 1 1 0 0 0 1 1 = 63
      ---------------
      0 0 0 0 0 0 1 1 = 03
 OR   0 0 1 0 1 1 0 0 = 2C
      ---------------
      0 0 1 0 1 1 1 1 = 2F
XOR   0 0 0 0 0 1 1 0 = 06
      ---------------
      0 0 1 0 1 0 0 1 = 29
CPL
      ---------------
      1 1 0 1 0 1 1 0 = D6
```

8.0 SKEW OPERATIONS

There are three types of skew operations;
1) logical shift left or right,
2) arithmetic shift left or right,
3) rotation left or right.

8.1 Logical Shifts

Logical shifts simply shift the pattern of bits in the same way as a shift register. No account is taken of the fact that the pattern of bits may represent a number.

8.1.1 Shift Right Logical

eg. SRL D shift contents of register D to the right.

Note that a 0 is shifted into bit 7 and that bit 0 moves into the Carry flag.

Program 8.1
Program Requirement: To shift register D right logical.

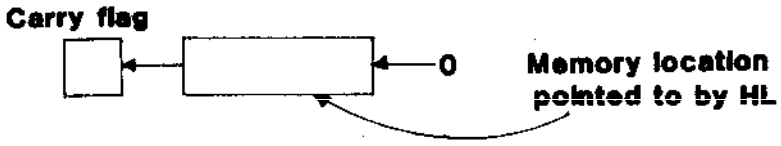
LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0D00	16 C0		LD	D,C0	D=11000000
0D02	CB 3A	SHIFT	SRL	D	
0D04	C3 02 0D		JP	SHIFT	Repeat.

Single step through the program and observe that register D takes on the following values in sequence:

C0 = 11000000
60 = 01100000
30 = 00110000
18 = 00011000
0C = 00001100
06 = 00000110
03 = 00000011
01 = 00000001
00 = 00000000
00 = 00000000, etc.

8.1.2 Shift Left

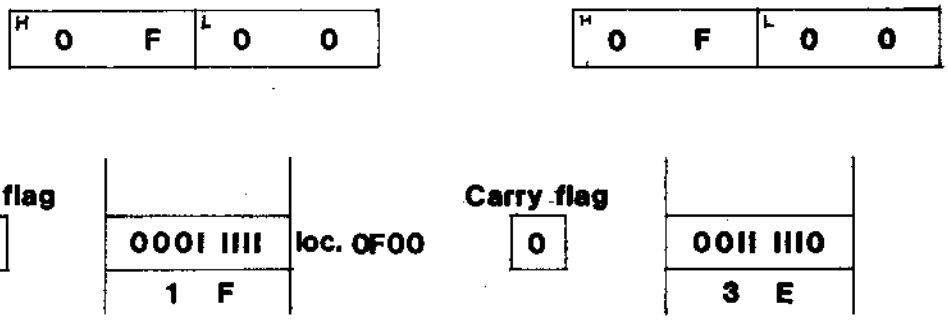
eg. SLA (HL) shift contents of memory location to the left.



Note that a 0 is shifted into bit 0 and that bit 7 moves into the carry flag.

Program 8.2
Program Requirement: To shift memory location 0F00 to the left.

ie. BEFORE AFTER



PROGRAM 8.2

LOC'N	CONTENTS	LABEL	OP CODE	ARGMENTS	COMMENTS
0D00	21 00 0F		LD	HL,0F00	
0D03	36 1F		LD	(HL),1F	
0D05	CB 26		SLA	(HL)	
0D07	DF 5B		.MRET		To NAS-SYS

Single step through the program and then examine location 0F00.

Note that 3E is twice 1F; the shift has caused a multiplication by 2. This shift is both a logical and an arithmetical shift. Its use to multiply by numbers other than 2 is illustrated in the following example.

Program 8.3
Program Requirement: To multiply a number in memory by 5.

It is only necessary here to note that 5x may be obtained by adding x to 4x(x shifted left twice).

PROGRAM 8.3

```
LOC'N  CONTENTS   LABEL   OP CODE  ARGUMENTS   COMMENTS
0D00   3A 00 0E           LD       A,(NUM)     Get x
0D03   57                 LD       D,A         Save in D
0D04   CB 27              SLA      A           A=2x
0D06   CB 27              SLA      A           A=4x
0D08   82                 ADD      A,D         Add 4x & x
0D09   32 01 0E           LD       (FIVEX),A
0D0C   DF 5B              .MRET                To NAS-SYS
::::
::::
::::
0E00   0A         NUM     0A                   10d
0E01   ..         FIVEX   ..                   For 5x
```

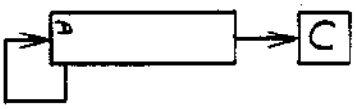
By single stepping through the program it will be seen that the result 32H (=50d) is formed in register A.

8.2 Arithmetic Shifts

The arithmetic shift left, SLA, has been previously described. When shifting an arithmetic quantity right, the sign bit must be preserved as well as being shifted right.

8.2.1 Shift Right Arithmetic

eg. SRA D — shift contents of register D right, arithmetic.




```
Initial content of register D:   10001001  = 89H = -119d
After 1 shift:                   11000100 = C4H = -60d
After 2 shifts:                  11100010 = E2H = -30d
After 3 shifts:                  11110001 = F1H = -15d
After 4 shifts:                  11111000 = F8H = -8d
After 5 shifts:                  11111100 = FCH = -4d
```

Note that the effect of an arithmetic shift right is a division by 2 (with any remainder lost). The sequence may be seen by stepping through the following program.

PROGRAM 8.4 Program Requirement: To shift register D right arithmetic.

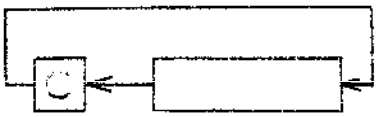
```
LOC'N  CONTENTS   LABEL   OP CODE  ARGUMENTS   COMMENTS
0D00   16 89              LD       D,89
0D02   CB 2A      LOOP:   SRA      D
0D04   C3 02 0D   JP      LOOP
```

8.3 Rotates

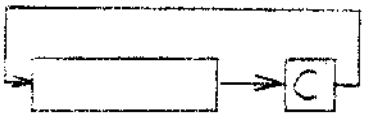
Rotates are similar to shifts except that bits shifted out from one end of the byte are shifted in at the other end. Rotated data may or may not go through the Carry flag.

ROTATE THROUGH CARRY

RL arg
RLA

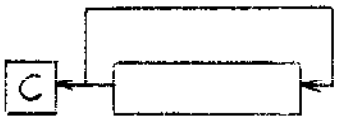


RR arg
RRA

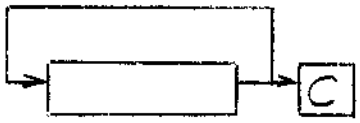


ROTATE CIRCULAR

RLC arg
RLCA



RRC arg
RRCA



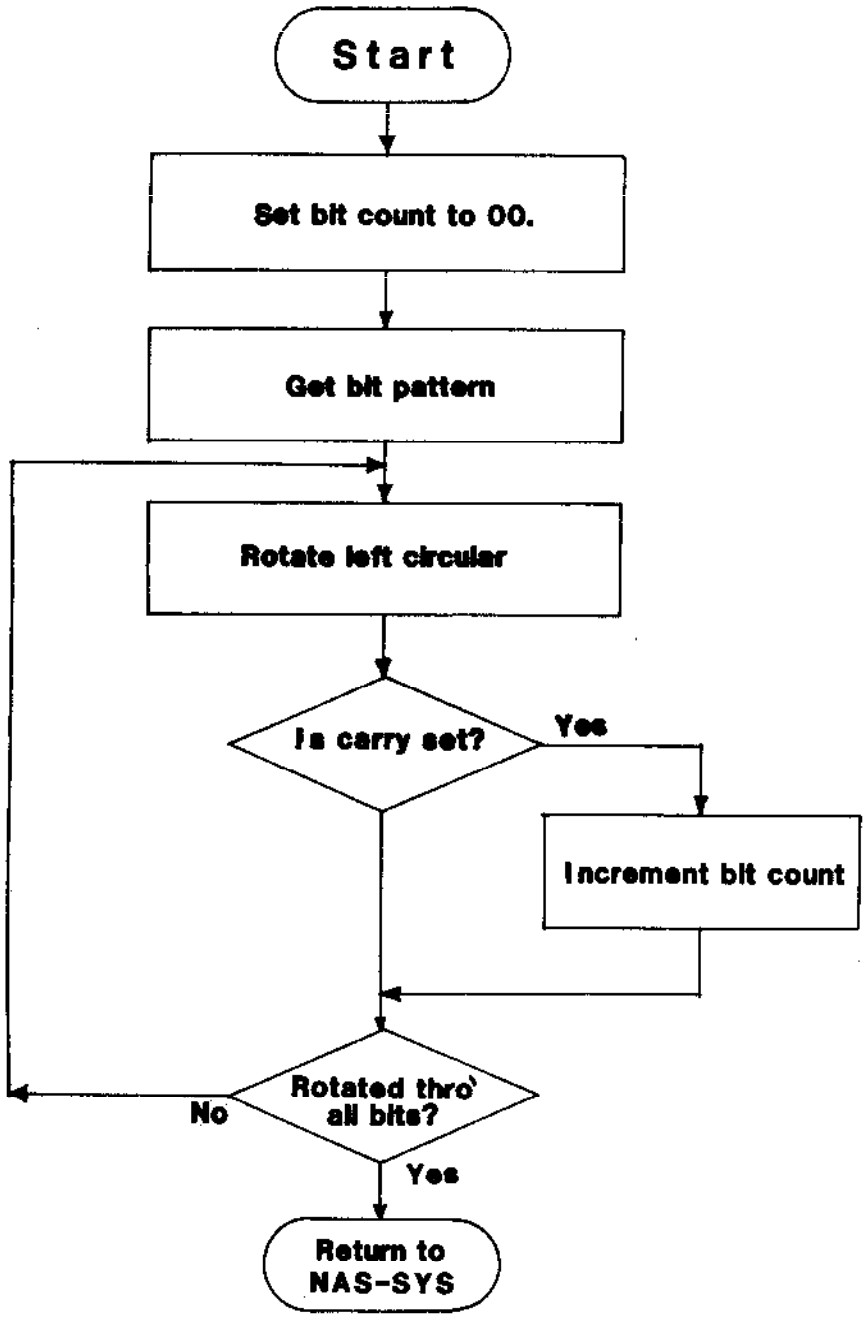
Note that if the argument is register A, the instruction may be written in two ways:

```
viz.      CB 17     RL A
          17        RLA
```

Both have the same effect on register A but whereas the first code will cause all the relevant flags to be affected, the second code affects only the Carry flag. (Flags are described in Chapter 9.)

Program 8.5
Program Requirement: To count the number of 1s in a byte stored in memory. Ie. If the byte is A3 = 10100011, then the program is to return the quantity 4 since there are 4 bits set in the byte.

Flowchart:



Flowchart for Program 8.5

PROGRAM 8.5

LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0D00	AF		XOR	A	Clear A
0D01	06 08		LD	B,8	Counter
0D03	21 00 0E		LD	HL,PATN	
0D06	CB 06	LOOP:	RLC	(HL)	Rotate
0D08	D2 0C 0D		JP	NC,NO1	Jump if bit is 0.
0D0B	3C		INC	A	Else increment bit count.
0D0C	05	NO1:	DEC	B	
0D0D	C2 06 0D		JP	NZ,LOOP	
0D10	32 01 0E		LD	(BITS),A	BITS holds result.
0D13	DF 5B		.MRET		To NAS-SYS.
::::					
::::					
::::					
0E00	A3	PATN:	A3		Data
0E01	..	BITS:	..		Result.

Execute the program and then examine location 0E01 for the result.

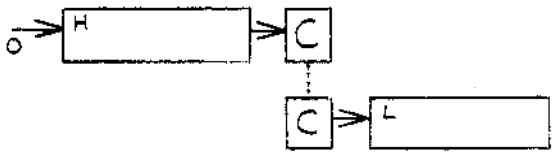
8.4 Sixteen bit Shifts

Although the instruction set does not include any 16 bit shifts, it is nonetheless possible to program such shifts. Register pair HL may be effectively shifted left simply by adding it to itself: ADD HL,HL. This may be extended to other register pairs:

```
EX      DE,HL
ADD     HL,HL
EX      DE,HL
```

Register pair HL may be shifted right by:

```
SRL     H
RR      L
```



and similarly for BC and DE.

9.0 MODIFICATION OF THE PROGRAM SEQUENCE

Several instructions modify the program sequence by changing the program counter. The JumP instructions JP START and JP NZ,loc have been used in previous programs.

Jump instructions may be conditional or unconditional.

9.1 Unconditional Jumps

eg. C3 m n JP nm This simply causes a jump to location nm from where the next instruction will be taken.

18 e JR e This is a jump relative to the program counter. It causes a jump to location PC+e, where PC is the current content of the program counter; e is a single byte number in the range -128d to +127d so that jumps are limited to within this range. Note that the current content of the program counter is always the address of the first byte following the JR instruction.

The relative jump, JR, is generally preferable to the absolute jump, JP, since the jump will be to the correct location wherever the program is placed in memory. It also requires only two bytes instead of three. However, it takes longer to execute and calculation of the displacement, e, requires some care. (This last point does not apply, of course, if an assembler program is being used.)

9.2 Conditional Jumps

These jumps will only occur if the specified condition is met. Those of most common use are given below.

```
* JP Z,nm    Jump to location nm if result was Zero.
* JP NZ,nm   Jump to location nm if result was Non-Zero.
  JP P,nm    Jump to location nm if result was Positive.
  JP M,nm    Jump to location nm if result was Minus.
* JP C,nm    Jump to location nm if result set the Carry flag.
* JP NC,nm   Jump to location nm if result did Not set the
             Carry flag.
```

* These instructions may also be relative jumps.

While many instructions affect the state of the flags and so will determine whether a subsequent conditional jump will occur, it must be remembered that this does not apply to all instructions. Thus, virtually all the LD instructions do not affect the flags. Should the flags be required following such an instruction, an instruction which does affect them but does not change the data should be inserted, eg. AND A sets the flags according to the byte in register A but does not change the contents of register A.

9.2.1 The Flag Register

Register F is simply a collection of flip-flops which are changed by many of the Z80 instructions. A description of the Flag register is given below.

Flag register format:

S	Z	X	H	X	P/V	N	C

S...Sign
Z...Zero
H...DAA half-carry
P/V.Parity/Overflow
N...DAA add/subtract
C...Carry
X...not used

Sign flag S is set if an operation caused bit 7 of the result to be set. Since positive two's complement numbers have bit 7 equal to 0 and negative numbers have bit 7 equal to 1, the Sign flag indicates the sign of the result. Arithmetic, logic, and skew operations affect this flag.

Zero flag Z is set if an operation caused the result to be zero. Arithmetic, logic, and skew operations affect this flag.

Carry flag C is set if an arithmetic operation produces a carry (or no borrow) from bit 7 of the result. It is also affected by the skew operations.

Parity/Overflow flag If an arithmetic operation causes overflow then P/V will be set. Overflow occurs when the addition or subtraction of two numbers produces a result outside the number range -128d to +127d. This flag also indicates parity following logical operations.

H & N flags These are used by the Z80 to carry out decimal arithmetic and are of litle importance to the programmer.

The operation of the Sign and Zero flags is straightforward, as shown in the following program.

PROGRAM 9.1

LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0D00	97		SUB	A	Clear A.
0D01	C6 27		ADD	27H	Result 27.
0D03	D6 30		SUB	30H	Result -3.
0D05	76		HALT		

Single step through the program and note that the A and F registers take on the following values:

	S	Z	X	H	X	P/V	N	C	Flags set
After D00, A = 00 F =	0	1	x	x	x	0	1	0	ZN
After D01, A = 27 F =	0	0	x	x	x	0	x	0	
After D03, A = FD F =	1	0	x	x	x	0	1	1	SNC

The Carry and Overflow flags require a little attention. The Carry flag, C, may be regarded as a ninth bit at the left-most end of every byte, while the Overflow flag, P/V, is most commonly used to indicate arithmetic overflow. It does not always follow that a carry into C (ie. out of bit 7) indicates an arithmetic overflow or vice versa. This is a consequence of two's complement arithmetic.

Although the microprocessor arithmetic operates on bit patterns as though they are signed binary numbers (ie. in two's complement), the programmer may often regard the bit patterns as unsigned numbers in the range 0 to 255d, as for example when using multiple-precision arithmetic. Consider the following examples:

eg 1.	Meaning if unsigned	Meaning if signed	Binary arithmetic
7F	+127	+127	0111 1111
ADD 02	+002	+002	0000 0010
	+129	+129	C=0←1000 0001
			S=1←┘
			P/V=1

There is no carry produced so C=0. If the programmer interprets the bit patterns as signed binary numbers, then the addition of two positive numbers has produced a negative result, (-127d in fact) and this is signalled by P/V=1. However, if the programmer interprets the bit patterns as unsigned binary numbers, then the result is correct.

```
eg2.     FF     +255          -001          1111 1111
     ADD 02     +002          +002          0000 0010
                +257          +001    C=1←0000 0001
                                      S=0←┘
                                      P/V=0
```

A carry out has been produced, C=1, but overflow has not occurred, P/V=0. If the programmer is using signed binary numbers the result is correct; if using unsigned numbers the C flag indicates that the result is greater than 255.

```
eg3.     01     +001          +001          0000 0001
     SUB 02     +002          +002          0000 0010
                -001          -001    C=1←1111 1111
                                      S=1←┘
                                      P/V=0
```

Here, P/V=0 signifies no overflow of signed binary numbers. Since this is a subtraction, C=1 signifies "no borrow" required from a higher order byte.

```
eg4.     81     +129          -127          1000 0001
     SUB 02     +002          +002          0000 0010
                +127          -129    C=0←0111 1111
                                      S=0←┘
                                      P/V=1
```

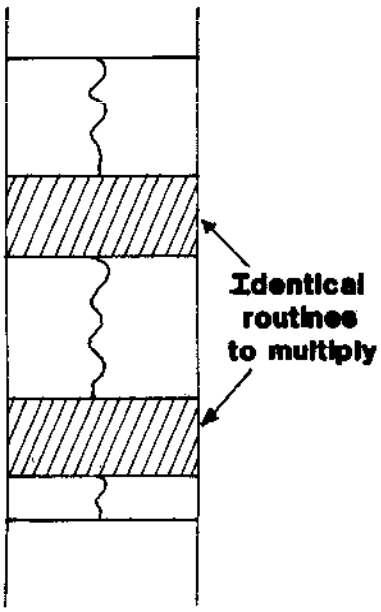
Here, the P/V flag indicates signed number overflow.

Short programs to illustrate these example are left as an exercise for the reader.

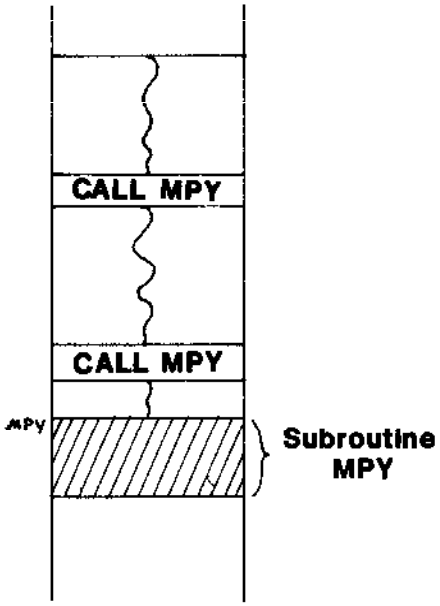
9.3 Jump to Subroutine,CALL

eg. CALL mn — Jump to the subroutine begining at location mn and when finished continue with the instruction immediately following the CALL.

Some programs make use of a particular sequence of instructions more than once, for example, it may be required to multiply two numbers at various points in the program. Such programs may be written in either of the two following ways.



Program with instruction sequence to perform multiplication written out in full twice.



Similar program with multiply routines replaced by CALL MPY. The instruction sequence to perform multiplication has been written as the subroutine, MPY.

When the cpu encounters the instruction CALL MPY it will jump to location MPY and execute the instruction in that and following locations. When the instruction RETurn is encountered in the subroutine, the cpu will jump back to the instruction immediately following the CALL MPY. As many calls to the subroutine as required may be written into the "main" or "calling" program.

Program 8.3 (page 8-2) multiplies a number by 5; it will now be written as a subroutine.

PROGRAM 9.2

```
LOC'N  CONTENTS   LABEL   OP CODE  ARGUMENTS  COMMENTS
0D00   06 01              LD       B,1
0D02   78                 LD       A,B
0D03   CD 00 0E   LOOP:   CALL     X5
0D06   4F                 LD       C,A
0D07   04                 INC      B
0D08   C3 03 0D           JP       LOOP
::::
       ;Multiply contents of A by 5, product in A.   Uses  D.
::::
0E00   57         X5:     LD       D,A
0E01   CB 27              SLA
0E03   CB 27              SLA
0E05   82                 ADD      A,D
0E06   C9                 RET
```

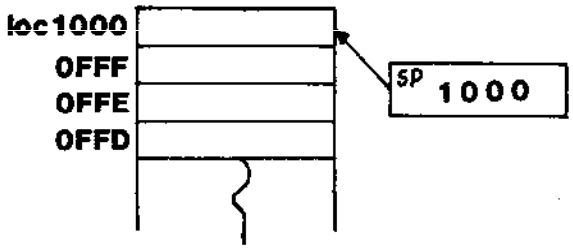
Single step through the program and note that when the instruction CALL X5 is executed, the Program Counter takes on the value 0E00, the location of the beginning of the subroutine. Observe that register B holds x and register C holds 5x where x is 1,2,3,etc. The start of the 5 times table in hexadecimal is given below.

```
5 x 1 = 05
5 x 2 = 0A
5 x 3 = 0F
5 x 4 = 14
5 x 5 = 19
5 x 6 = 1E,etc.
```

9.4 Saving Data on the Stack

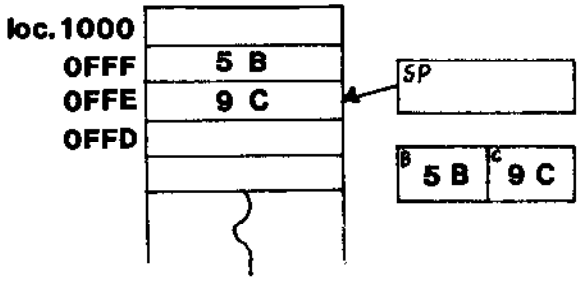
The stack is an area of memory defined by the programmer and used to store data temporarily. It is a very convenient mechanism. Data is stored on the stack by the PUSH instruction and retrieved by the POP instruction. The data POPped from the stack is always the last data that was PUSHed onto it, ie. the stack operates as a Last In, First Out (LIFO) buffer store.

Suppose the programmer defines the stack area as memory locations 1000H and below. Then the Stack Pointer, SP, must be loaded initially with 1000H. The situation is then as shown below.



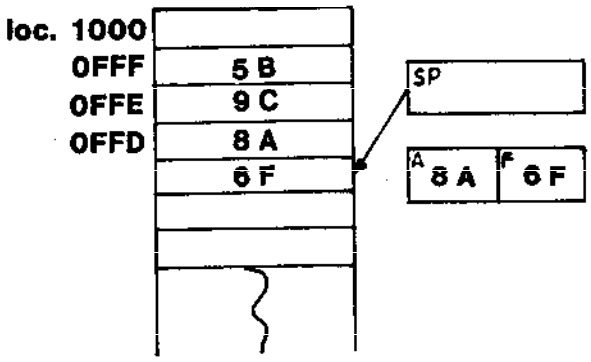
Stack pointer points to the current "top" of the stack.

After execution of PUSH BC , the stack looks like:



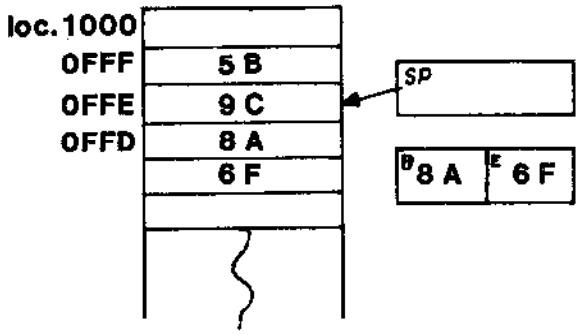
The Stack Pointer is decremented and the contents of register B stored in the location pointed to by the Stack Pointer. It is then decremented again and the contents of register C stored in the next location down.

After execution of PUSH AF, the stack looks like:



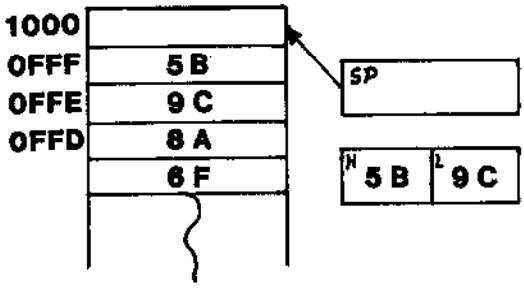
SP is decremented and the contents of register A stored in the location to which it points. SP is then decremented again and the contents of register F stored in the next location down.

After execution of POP DE:



The data in the location originally pointed to by SP has been transferred to register E, SP incremented and the data then pointed to has been transferred to register D.

After execution of POP HL :



The data in the location originally pointed to by SP has been transferred to register L, SP incremented and the data then pointed to has been transferred to register H.

In practice, the programmer need only have regard for the order in which data has been PUSHed onto the stack and, to a lesser extent, the depth of the stack itself. This last point is to prevent the stack area encroaching onto an area of memory used for some other purpose.

The stack is most useful for saving the contents of cpu registers, so freeing them for another use, and then restoring them to their original contents in order to proceed. NAS-SYS sets the Stack Pointer to 1000H.

MACHINE CODE PROGRAMMING
for the NASCOM 1 & 2

10.0 THE VIDEO DISPLAY

10.1 Writing to the Video Display

Several of the subroutines in NAS-SYS facilitate writing to the display. The following program uses subroutines .B2HEX, .SPACE, .CRLF, and .MRET.

.B2HEX displays the contents of register A on the screen in hexadecimal. It is called by the code DF 68. Register A is modified by this routine.

.SPACE moves the cursor on one place so displaying a space. It is called by the code DF 69. Register A is set to 20H by this routine.

.CRLF positions the cursor at the beginning of the next line, ie. it performs what on a printer would be Carriage Return & Line Feed. It is called by the code DF 6A. Register A is set to 0DH by this routine.

Note that NAS-SYS routines are regarded here as additions to the Z80 assembly code and are distinguished by their beginning with a full stop. Thus, .CRLF is regarded as a Z80 assembly code mnemonic corresponding to the machine code DF 6A. In fact, the code DF 6A is:-

DF	RST	18H	Call subrouitne at 18H.
6A	DEFB	6AH	Define a byte.

That is, it is a call to the subroutine beginning at location 0018H (in NAS-SYS). This subroutine, called SCAL, uses the byte in the location following the RST 18H instruction, in this case 18H, to jump to the location where the subroutine to produce a carriage return and line feed actually starts. For present purposes it is best simply to regard these commands as adddditions to the Z80 code.

PROGRAM 10.1
Program Requirement: To display the five times table on the screen.

LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0D00	06 01		LD	B,01	Initialise B to 1.
0D02	78		LD	A,B	Transfer to A.
0D03	DF 68	LOOP:	.B2HEX		Display A.
0D05	DF 69		.SPACE		Write a space.
0D07	78		LD	A,B	Restore A.
0D08	CD 00 0E		CALL	X5	Multiply subroutine.

```
0D0B   DF 68              .B2HEX              Display
                                              product.
0D0D   DF 6A              .CRLF               New line.
0D0F   04                 INC       B
0D10   78                 LD        A,B
0D11   FE 0E              CP        0E        Finished?
0D13   C2 03 0D           JP        NZ,LOOP   No.
0D16   DF 5B              .MRET               Yes,return to
                                              NAS-SYS.

::::
::::   ;Multiply number in A by 5.
::::
0E00   57          X5:    LD        D,A
0E01   CB 27              SLA
0E03   CB 27              SLA
0E05   82                 ADD       A,D
0E06   C9                 RET
```

Execute the program and see the five times table in hexadecimal.

10.2 Reading the Keyboard & Displaying a Character

The keyboard may be read by using the NAS-SYS subroutine, .BLINK, called by the code DF 7B. This subroutine scans the keyboard repeatedly until a key is pressed, and then places the ASCII code for the character into register A.

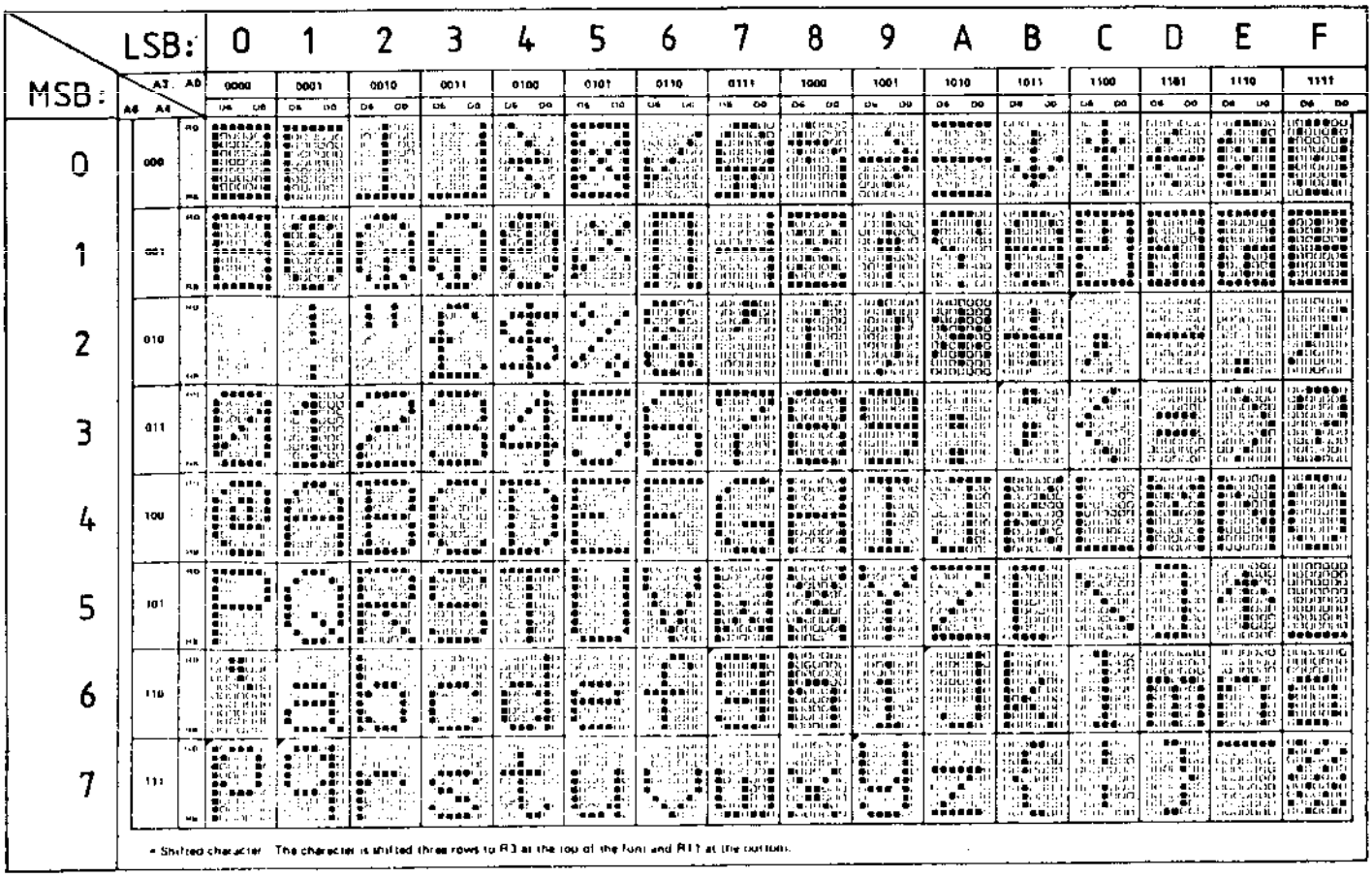
ASCII, or American Standard Code for Information Interchange, is an accepted standard for coding alphanumeric characters. Program 10.2 displays the ASCII code for the keyboard characters.

PROGRAM 10.2
Program Requirement: To read the keyboard and display the character and its ASCII code.

```
LOC'N  CONTENTS    LABEL    OP CODE   ARGUMENTS  COMMENTS
0D00   DF 6A       START:   .CRLF                Newline
0D02   DF 7B                .BLINK               Get character
0D04   F7                   .ROUT                Display it.
0D05   F5                   PUSH      AF         Save A since
0D06   DF 69                .SPACE               .SPACE uses A.
0D08   F1                   POP       AF         Restore A.
0D09   DF 68                .B2HEX               Display code.
0D0B   C3 00 0D             JP        START
```

Note that since register A is used by the routine .SPACE, its contents are saved on the stack before calling the routine. Register A is then popped from the stack immediately afterwards.

Execute the program then press any of the keys and compare the codes with those given in the table below. Observe that pressing 'G' displays 'G 47' , ie. 47H is the ASCII code for the character 'G'; pressing SHIFT G displays 'g 67' where 67H is the ASCII code for character 'g' .These codes and others should be found in the table. Note that the commercial 'at' key produces no effect unless shifted. This key has a special function, it is the CONTROL key.



Codes for the NASCOM character set

The effect of the CONTROL key may be observed as follows: press CONTROL ; , ie press @; together, and observe that instead of the character ';' (ASCII 3BH) being displayed, '{' (ASCII code 7BH) is displayed. Try again with @, and @/ . The effect of the CONTROL (@) key is to add 40H to the code for the keyed character, ie. the displayed character is four rows down the column in the table. This fact may be used to display some of the graphics characters; press @G to diplay the 'bell' character, (ASCII 07H), since four rows down the table from 'G' (wrapping around from bottom to top) brings the code to 07H.

However, not all the graphics characters can be displayed in this way, as many of the codes (including those from 11H to 18H) are interpreted by .ROUT as screen editing commands. The effect of these is shown in the following program which places all possible codes in turn into register A before calling .ROUT .

PROGRAM 10.3
Program Requirement: To display all possible ASCII codes and their corresponding characters.

Because .CRLF and .B2HEX modify the contents of register A, the incrementing count is kept in register C and transferred to register A when required. The subroutine .BLINK is included simply to allow the codes and characters to be displayed at a speed determined by the user. The space bar or any character key must be pressed before the program proceeds to display the next code.

```
LOC'N  CONTENTS   LABEL   OP CODE  ARGUMENTS  COMMENTS
0D00   0E 00              LD       C,0H
0D02   DF 6A      NEXT:   .CRLF
0D04   79                 LD       A,C
0D05   DF 68              .B2HEX              Display code
0D07   DF 69              .SPACE
0D09   79                 LD       A,C
0D0A   F7                 .ROUT               Display char
0D0B   0C                 INC      C
0D0C   DF 7B              .BLINK              Wait
0D0E   C3 02 0D           JP       NEXT
```

Execute the program and observe that while the character corresponding to the ASCII code in register A is normally displayed, some codes produce screen editing commands.
Thus, code 00H produces no effect,
08H produces a backspace,
0AH produces no effect,
0CH clears the screen,
0DH produces a carriage return & line feed,
11H moves the cursor left,
12H moves the cursor right,
13H moves the cursor up,
14H moves the cursor down,
15H deletes the character at the cursor (and moves the rest of the line left)
16H moves the line to the right,
17H moves the cursor to the start of the current line,
18H produces a carriage return (unless the cursor is at the start of a line),
1BH deletes the current line.

Observe that code 80+x always produces the character

having code x, even if code x is a screen editing command. Thus, 8DH is displayed as the character left-arrow not as a carriage return & line feed (= code 0DH).

10.3 Software Delays, .RDEL and .TDEL

Subroutine .RDEL produces a delay proportional to the number in register A. If A is set to zero before calling .RDEL, the delay is about 5.4ms, (2.7ms on NASCOM 2). It may be called many times to produce longer delays. The following instructions produce a delay of about 1 second, by calling .RDEL 185d(=B9H) times.

```
06 B9              LD      B,B9
FF         DLY:    .RDEL
10 FD              DJNZ    DLY
```

Note that the initial content of register A is insignificant in this program since register A is set to zero by the first call to .RDEL . The instruction DJNZ DLY decrements register B and, if it not then zero, causes a relative jump to DLY.

These instructions may be written as a subroutine, called ONESEC, and then the instruction CALL ONESEC will produce a delay of about one second.

Subroutine .TDEL (called by code DF 5D) produces a delay of 2.7s (1.35s on NASCOM 2) by calling .RDEL 512 times. It also sets registers A & B to zero.

The previous program slowed down the display of codes by waiting for a key depression; the following program replaces the keyboard scan by ONESEC .

PROGRAM 10.4
Program Requirement: As Program 10.3 but utilising a software delay.

```
LOC'N  CONTENTS    LABEL   OP CODE  ARGUMENTS  COMMENTS
0D00   0E 00               LD       C,0
0D02   DF 6A       NEXT:   .CRLF               Newline.
0D04   79                  LD       A,C        Next code.
0D05   DF 68               .B2HEX              Display code.
0D07   DF 69               .SPACE
0D09   79                  LD       A,C
0D0A   F7                  .ROUT
0D0B   0C                  INC      C
0D0C   CD 00 0E            CALL     ONESEC
0D0F   C3 02 0D            JP       NEXT
::::
::::
```

```
::::
::::    ;Subroutine to provide one second delay.
::::    ;Sets register A to zero.
::::
0E00    06 B9       ONESEC: LD      B,B9
0E02    FF          DLY:    .RDEL
0E03    10 FD               DJNZ    DLY
0E05    C9                  RET
```

Execute the program and observe the one second delay between the codes.

10.4 Writing Messages on the Screen

It is frequently useful to write a message on the screen in the course of a program. Two ways of doing this are illustrated in the following examples.

10.4.1 Programming the message

PROGRAM 10.5
Program Requirement: To read a character from the keyboard and display on the screen "You typed (keyed character)".

```
LOC'N  CONTENTS    LABEL   OP CODE  ARGUMENTS  COMMENTS
0D00   DF 7B       READ:   .BLINK              Read keyboard
0D02   08                  EX       AF,AF'     Save A in A'
0D03   21 00 0E            LD       HL,MSG     MSG is start of
                                                message.
0D06   7E          WRITE:  LD       A,(HL)     Get first
                                                character.
0D07   FE 24               CP       24         Is it $?
0D09   CA 11 0D            JP       Z,CHAR     Yes,jump.
0D0C   F7                  .ROUT                No,display
0D0D   23                  INC      HL         Point to next
                                                character.
0D0E   C3 06 0D            JP       WRITE      Continue
0D11   08          CHAR:   EX       AF,AF'     Message done,
                                                restore keyed
                                                character to A.
0D12   F7                  .ROUT                Display
                                                character.
0D13   DF 6A               .CRLF
0D15   DF 6A               .CRLF
0D17   C3 00 0D            JP       READ
::::
::::
::::
```

```
::::
0E00    59 6F 75 MSG:       Y o u
0E03    20 74 79            (sp) t y
0E06    70 65 64            p e d
0E09    20 24               (sp) $
```

The code from location 0D06 to 0D15 inclusive, together with the message itself (locations 0E00 to 0E0A), forms a general routine to write a message. Register pair HL is initialised to the address of the start of the message code and then succeeding bytes are transferred into register A and displayed using subroutine .ROUT. In order to detect the end of the message, a 'delimiter' character, in this case '$' (code 24H) is used. On receipt of this character, the program ends the message printing routine.

An alternative way of detecting the end of a message would be to count the number of bytes in it and then display that number of characters. This is most conveniently done using the LDIR instruction. However, the character count must be changed whenever the message is changed. The use of a delimiter character is thus more convenient.

10.4.2 Using the Print String Subroutine, .PRS

The NAS-SYS subroutine .PRS provides an alternative method for writing messages. It is only necessary to program the instruction .PRS (code EF) followed by the ASCII code for the message delimited by 00H. The following program uses this method.

PROGRAM 10.6
Program Requirement: As for Program 10.5 but using subroutine .PRS.

```
LOC'N  CONTENTS    LABEL   OP CODE  ARGUMENTS  COMMENTS
0D00   DF 7B       READ:   .BLINK
0D02   08                  EX       AF,AF'
0D03   EF                  .PRS
0D04   59 6F 75 20         Y o u (sp)
0D08   74 79 70 65         t y p e
0D0C   64 20 00            d (sp) 00H
0D0F   08                  EX       AF,AF'
0D10   F7                  .ROUT
0D11   DF 6A               .CRLF
0D13   DF 6A               .CRLF
0D15   C3 00 0D            JP       READ
```

Execute the program and observe that it has the same effect as the previous one.

NAS-SYS provides an additional facility for entering messages by allowing the direct entry of ASCII characters. Thus, when entering the program above into memory, locations 0D04 to 0D0D may be entered as :

,Y,o,u,(sp),t,y,p,e,d,(sp)

The corresponding ASCII codes, as shown in the program, will automatically be entered.

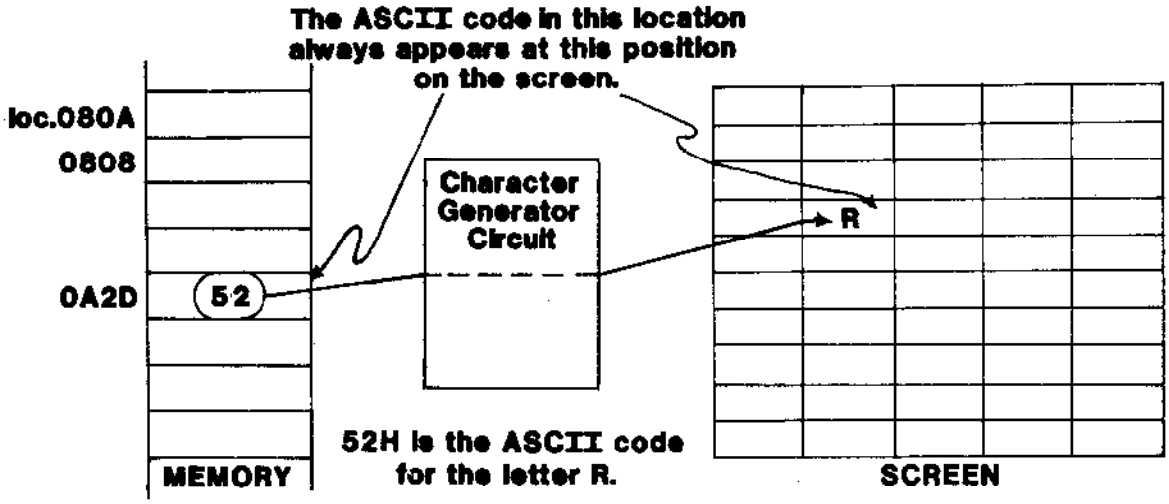
Screen editing commands may be written using .PRS. The following program is given simply as an illustration.

PROGRAM 10.7

```
LOC'N  CONTENTS    LABEL    OP CODE   ARGUMENTS  COMMENTS
0D00   EF          START:   .PRS
0D01   2A 08 08             * BS BS
0D04   12 2A 00             CUR * 00
0D07   06 18                LD        B,18H
0D09   FF          DEL:     .RDEL
0D0A   10 FD                DJNZ      DEL
0D0C   C3 00 0D             JP        START
```

10.5 Organisation of the Display

The NASCOM display is said to be 'memory mapped'. The screen is divided into 16 lines, each capable of displaying 48 characters. Each of the 768(=16x48) printing positions corresponds to a location in the video RAM. The screen may be regarded as a window through which the contents of the video RAM may be viewed. The character generation circuit produces on the screen the character whose ASCII code is in the corresponding location in video RAM, as shown below.



The Memory-Mapped Screen

It is thus possible to write a character on the screen in any position by loading the memory location corresponding

to that position with the ASCII code for the character. A map of the screen showing the actual memory addresses is shown below.

0BCA 0BCB 0BCC ...	..0BF8 0BF9
080A 080B 080C ...	..0838 0839
084A 084B 084C ...	..0878 0879
088A 088B 088C ...	..08B8 08B9
08CA 08CB 08CC ...	..08F8 08F9
....	
....	
....	
....	
0B8A 0B8B 0B8C ...	..0BB8 0BB9

Map of the Video RAM Addresses

Note 1: The top row (locations 0BCA to 0BF9) is not scrolled.
Note 2: Some locations, those in the 'margins', are not displayed.

A simple, though inelegant, method of writing to the display is to 'poke' the appropriate character code into the video RAM, as shown below.

PROGRAM 10.8
Program Requirement: To display the message 'Nascom' in the middle of the top line of the screen.

```
LOC'N  CONTENTS   LABEL   OP CODE  ARGUMENTS   COMMENTS
0D00   21 DF 0B           LD       HL,0BDF     0BDF is the
                                                location of the
                                                middle of the
                                                top line.
0D03   36 4E              LD       (HL),4E     4E='N'
0D05   23                 INC      HL          Point to next
                                                printing
                                                position.
0D06   36 41              LD       (HL),41     41='a'
0D08   23                 INC      HL
0D09   36 53              LD       (HL),53     53='s'
0D0B   23                 INC      HL
0D0C   36 43              LD       (HL),43     43='c'
0D0E   23                 INC      HL
0D0F   36 4F              LD       (HL),4F     4F='o'
0D11   23                 INC      HL
0D12   36 4D              LD       (HL),4D     4D='m'
0D14   DF 5B              .MRET
```

Register pair HI is initialised to 0BDF which is the

address in the video RAM corresponding to where the first character of the message is to be displayed. The ASCII code for 'N' is then loaded into this location, HL incremented and the ASCII code for 'a' loaded into the video RAM. This is repeated until the complete message is displayed.

By single-stepping through this program, the message will be seen to build up on the display. (Clear the screen first.)

It is particularly useful to write headings and titles on the top line of the screen since this line, unlike the others, is not scrolled up, thereby leaving the message on the display. However, NAS-SYS subroutines .ROUT and .PRS do not allow writing to this line so it is necessary to poke the video ram locations with the required code. A more elegant method of doing this is described in the following section.

10.6 Single-step Headings Program

A more efficient method of writing messages is illustrated by the program in Appendix b, which displays the headings for use when single stepping through programs. In that program, the ASCII codes for the message are stored in a block beginning at MSG and the program simply transfers them into the video RAM. This is done most effectively by making use of the block transfer instruction, LDIR. This loads the location whose address is in register pair DE with the contents of the location whose address is in register pair HL. Both DE and HL are then automatically incremented, register pair BC is decremented, and the process repeated until BC reaches zero.

Thus DE is loaded with 0BCA, the location corresponding to the top left-hand corner of the screen; HL is loaded with 0F9B, the location of the first byte of the message; BC is loaded with 0030, the number of bytes in the message.

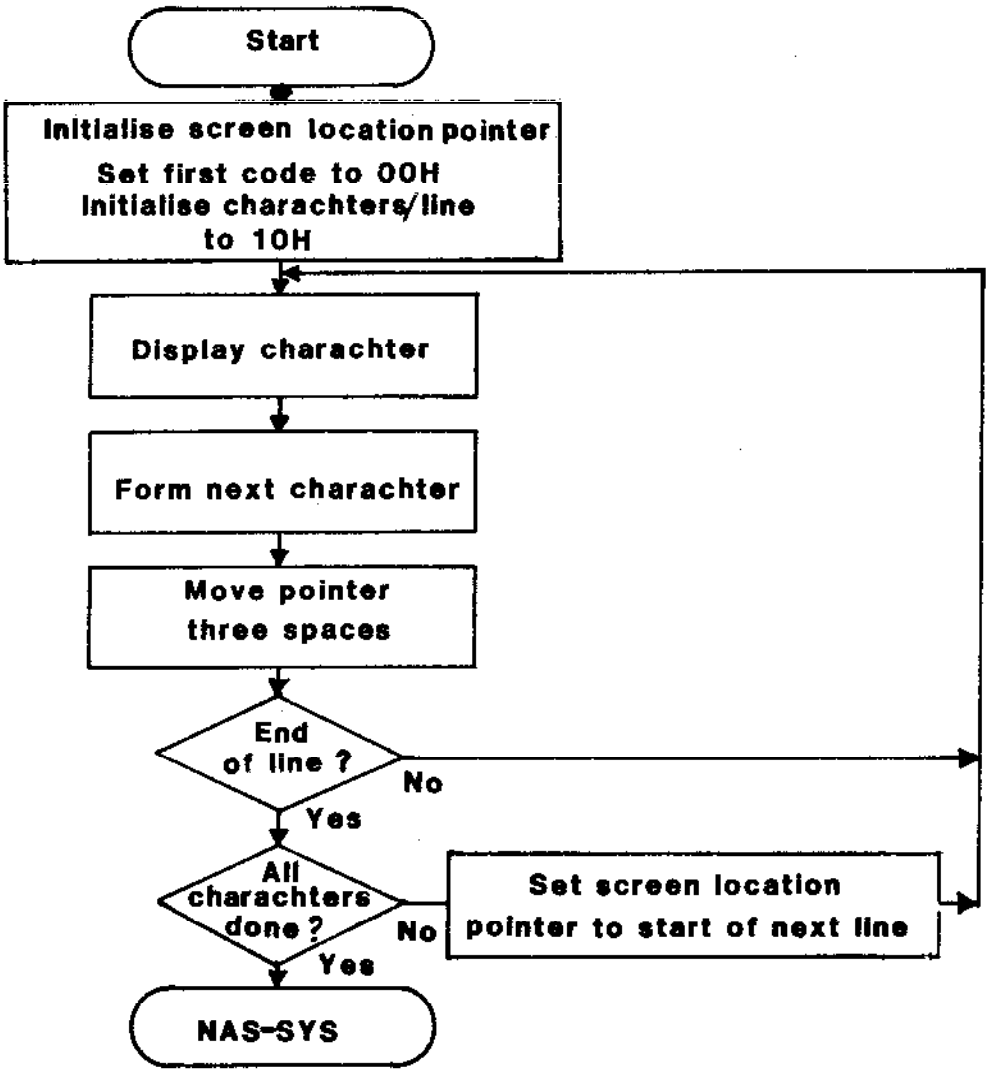
The initial contents of BC, DE, and HL are saved temporarily on the stack so that if another program is making use of any of these registers their contents will not be changed by execution of the headings program.

The LDIR instruction is very useful for copying a block of data from one area of memory to another. Its use provides a more convenient method of achieving the objective of Program 6.6.

10.7 Displaying the Complete Character Set

Program 10.9
Program Requirement: To display the complete character set on the screen.

Flowchart:



Flowchart for Program 10.9

PROGRAM 10.9

LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0D00	3E 0C		LD	A,0CH	0C is clear screen
0D02	F7		.ROUT		
0D03	11 10 00	START:	LD	DE,0010	Margin width.
0D06	21 0B 08		LD	HL,080B	Near top left of display.
0D09	3E 00		LD	A,00	First character code.
0D0B	06 10	LINE:	LD	B,10	16d chars/line.
0D0D	77	WRITE:	LD	(HL),A	Write char.
0D0E	3C		INC	A	Next char code.
0D0F	23		INC	HL	Move pointer..
0D10	23		INC	HL	..right
0D11	23		INC	HL	..3 spaces.
0D12	10 F9		DJNZ	WRITE	Jump to WRITE if line not ended.
0D14	FE 80		CP	80	
0D16	CA 1D 0D		JP	Z,END	Jump to NAS-SYS if all characters done.
0D19	19		ADD	HL,DE	Skip over margin.
0D1A	C3 0B 0D		JP	LINE	Next line.
0D1D	3E 4A	END:	LD	A,4A	
0D1F	32 29 0C		LD	(0C29),A	
0D22	3E 0B		LD	A,0BH	
0D24	32 2A 0C		LD	(0C2A),A	
0D27	DF 5B		.MRET		

Execute the program and see the complete character set displayed.

Notes:

1. There are sixteen non-displayed memory locations in the video RAM between the end of one line on the display and the beginning of the next. Thus, in order to move the pointer from one line to the next 10H must be added. This quantity is stored in register pair DE and added by the instruction in location 0D11.

2. Register pair HL points to the current printing position. It is initially loaded with 080BH, corresponding to the printing position near the top left-hand corner of the display.

3. Register A holds the ASCII code of the character to be displayed. It is initially loaded with 00H and incremented as far as 80H. At this point the complete character set has been displayed and this is detected by the

instructions in locations 0D12 and 0D14.

4. Register B is loaded with the number of characters per line to be displayed. This is sixteen (10H). Each character is followed by three spaces so using up the full 48d character width of a line. Following the display of each character and three spaces, the DJNZ WRITE instruction at location 0D12 decrements register B and, if it is zero, causes a jump to WRITE.

5. At END, the cursor is repositioned towards the bottom of the screen so that the 'NAS-SYS 1' title resulting from .MRET does not overwrite the displayed character set. The cursor position, ie. its address in video ram, is stored in locations 0C29 and 0C2A. These locations are loaded with 4A and 0B respectively, so positioning the cursor at location 0B4A in video ram. This is at the start of the next line from the bottom of the screen.

11.0 INPUT AND OUTPUT

There are three methods of transferring data between the cpu and the outside world. These are:

1. Programmed I/O, in which the data transfer always takes place at a specific part of the program.

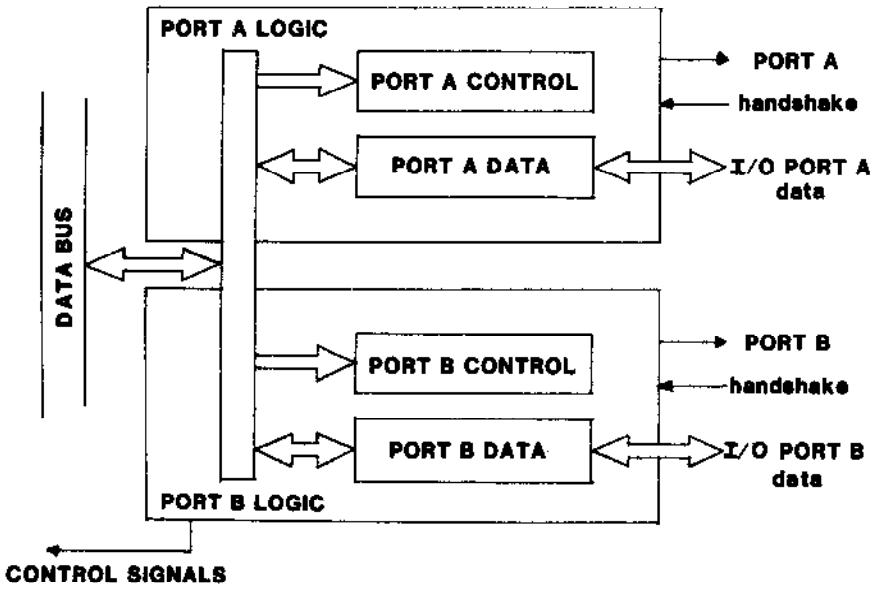
2. Interrupt I/O, in which the external device determines when the data transfer takes place. It does this by stopping the current program and causing the cpu to execute an 'interrupt service' routine in which the data transfer takes place.

3. DMA or Direct Memory Transfer, in which the external device assumes control of the memory and data buses and data transfers take place directly between the memory and the device.

Both methods 1 and 2 are made simple by the use of a peripheral input-output device, PIO. Method 3 is not so often used and only the first two methods are described in this chapter.

11.1 Peripheral Input-Output Device

The PIO is connected to the data bus of the microcomputer and is controlled by signals from the cpu. Its internal structure is shown below:



Internal Structure of the PIO

Each port may be an input or output (or both) depending on the pattern of bits in the control register.

In the NASCOM 1, port A control register is port 6,
port A data register is port 4,
port B control register is port 7,
port B data register is port 5.

11.2 Transferring Data between the CPU and the PIO Registers

Instructions to perform transfers between a port and register A:

IN A,(n)	where n is the port number (4,5,6,or 7). Data from port n is transferred to register A.
OUT (n),A	Data is transferred from register A to port n.

Instructions to perform transfers between a port and any cpu general purpose register:

IN reg,(C)	Data from the port whose number is kept in register C is transferred to the specified cpu register.
OUT (C),reg	Data is transferred from the specified cpu register to the port whose number is kept in register C.

11.3 Programmed Input and Output

Programmed I/O is the simplest method of data transfer. It is illustrated in the following programs which use the leds and switches attachment, the construction of which is described in Appendix c .

Program 11.1
Program Requirement: To read the switches (port 5) into the leds (port 4).

LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0D00	3E 07		LD	A,07	Disable interrupts
0D02	D3 07		OUT	(7),A	.on port 5
0D04	D3 06		OUT	(6),A	.& port 4
0D06	3E 0F		LD	A,0F	Select o/p mode
0D08	D3 06		OUT	(6),A	for port 4
0D0A	3E 4F		LD	A,4F	Select i/p mode
0D0C	D3 07		OUT	(7),A	for port 5
0D0E	DB 05	LOOP:	IN	A,(5)	Input 5
0D10	D3 04		OUT	(4),A	Output 4
0D12	C3 0E 0D		JP	LOOP	

Execute the program and observe that the leds may be switched on or off by the switches. Then single step through it and note that the state of the switches is read into register A by the IN A,(5) instruction, and output to the leds by the OUT (4),A instruction.

Instructions in locations 0D00 to 0D0D inclusive initialise the PIO and are executed only once, while the last three instructions of the program actually make the data transfers.

In order to disable the interrupt facility the control register of both ports must receive the code 07H. The first three instructions effect this.

Since the leds are connected to data port 4, this port must be an output port. (Data directions are defined from the point of view of the cpu.) The third and fourth instructions send code 0FH to the control register so defining all eight bits of port 4 as outputs.

The switches are connected to data port 5 so this must be an input port. The fifth and sixth instructions send code 4FH to the corresponding control register so defining all eight bits of port 5 inputs.

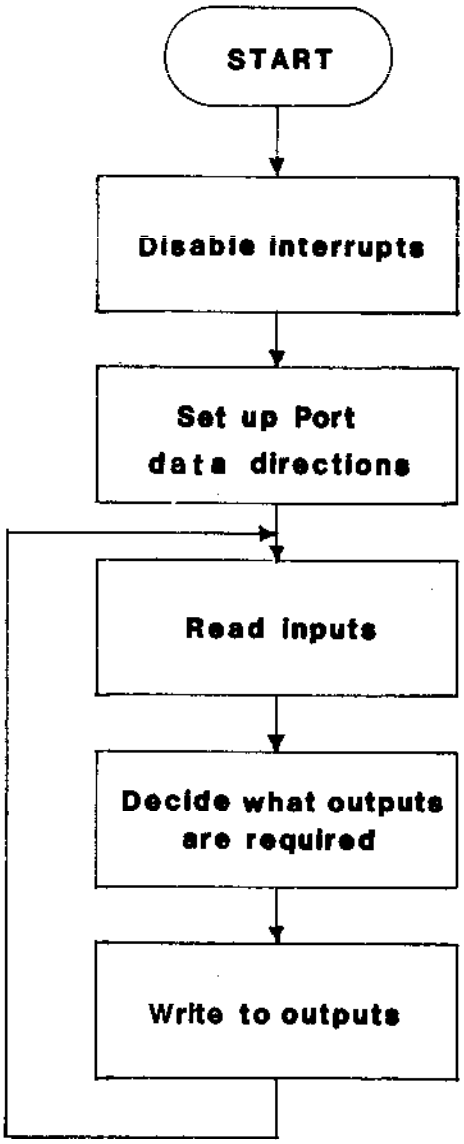
The PIO initialisation instructions for this form of data transfer (said to be without interrupts, without handshaking) may be summarised thus:

> At the beginning of the program load the port control register (port 6 or 7) with:
> 07 to disable interrupt facility on PIO,
> 0F for output port OR 4F for input port.

This procedure can be seen at the beginning of the programs which follow.

11.3.1 General Program Structure for Simple I/O

Program 11.1 shows how to make a (rather inefficient!) direct connection between the switches and the leds. However, it illustrates the method which is used in many microprocessor applications. The method is shown in the following flowchart:

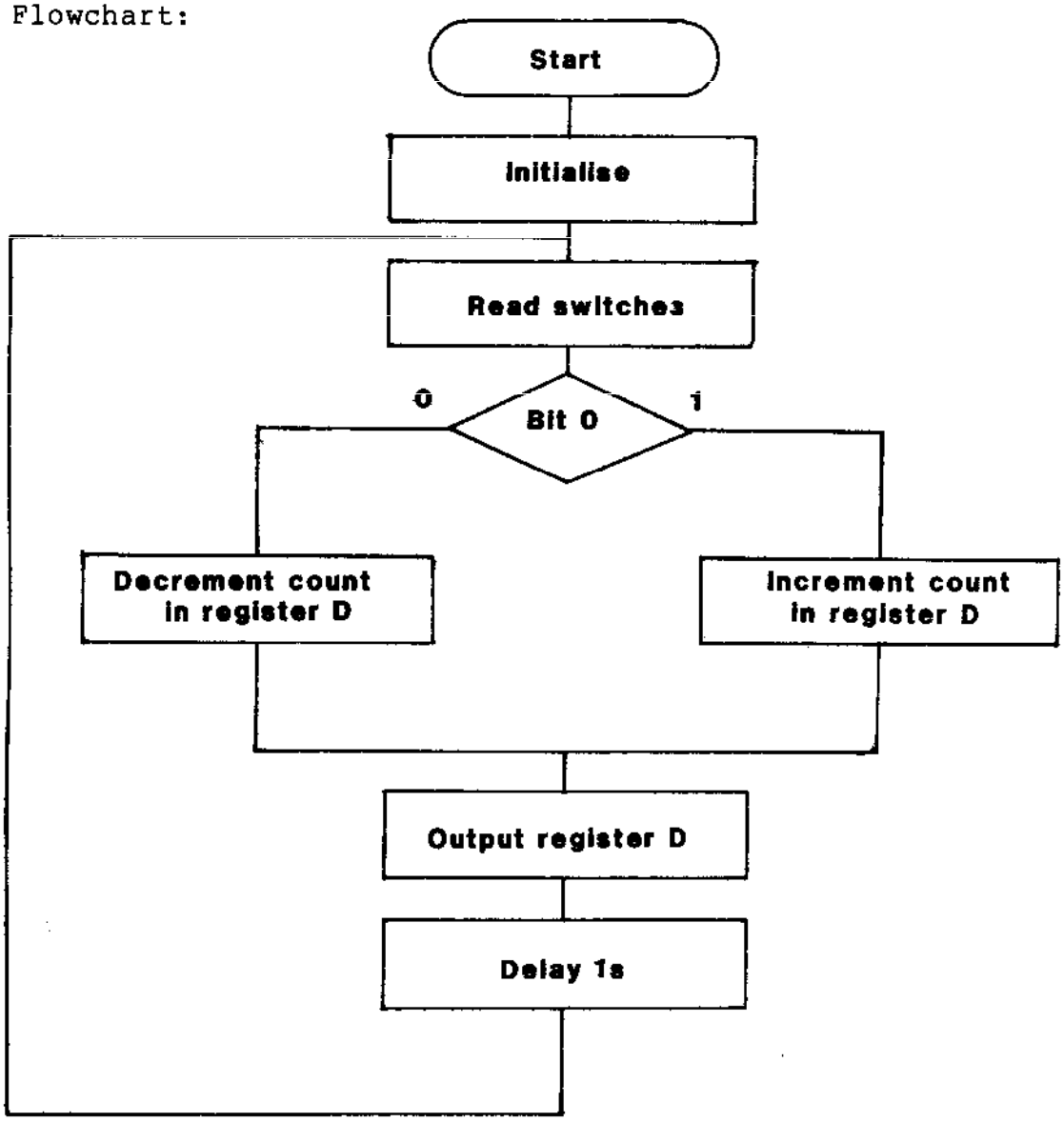


General Flowchart of Programs Employing Simple I/O

A further example of the general method is given in the following program.

Program 11.2
Program Requirement: To produce a pure binary up-count on the leds if switch 0 is 1, and a down-count if the switch is 0.

Flowchart:



The program keeps the current count in register D so, in order to output from this register, the OUT (C),D instruction is used, register C having been set to 04 at the beginning of the program. The AND 01 instruction logically ANDs the state of the switches (which have been read into register A) with 01 thereby setting all bits other than bit 0 to zero. The result will therefore be 00 or 01 depending on the position of switch 0 and this sets up the Zero flag for use by the JP NZ,UP instruction.

PROGRAM 11.2

LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0D00	3E 07		LD	A,07	Disable interrupts.
0D02	D3 07		OUT	(7),A	.on port 5
0D04	D3 06		OUT	(6),A	& port 4
0D06	3E 0F		LD	A,0F	Select o/p mode
0D08	D3 06		OUT	(6),A	..port 4.
0D0A	3E 4F		LD	A,4F	Select i/p mode
0D0C	D3 07		OUT	(7),A	..port 5.
0D0E	0E 04		LD	C,04	For I/O.
0D10	DB 05	LOOP:	IN	A,(5)	Read 5
0D12	E6 01		AND	01	Mask off all except bit 0.
0D14	C2 1B 0D		JP	NZ,UP	Jump if Sw=1.
0D17	15		DEC	D	Here to count down.
0D18	C3 1C 0D		JP	CONT	
0D1B	14	UP:	INC	D	Here to count up.
0D1C	ED 51	CONT:	OUT	(C),D	Output register D.
0D1E	CD 00 0E		CALL	ONESEC	
0D21	C3 10 0D		JP	LOOP	
::::					
::::					
::::					
::::					
0E00	06 B9	ONESEC:	LD	B,B9	
0E02	FF	DLY:	.RDEL		
0E03	10 FD		DJNZ	DLY	
0E05	C9		RET		

Execute the program and observe a binary count on the leds, up if switch 0 is 1, down if switch 0 is 0.

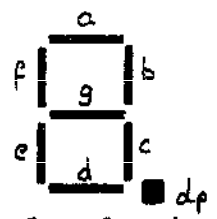
11.3.2 Driving a Seven-Segment Display

The seven-segment display described in Appendix c comprises eight leds, each of which may be turned on or off under program control. A program to drive the display is given below. This program will be modified subsequently and new programming techniques introduced.

MACHINE CODE PROGRAMMING
for the NASCOM 1 & 2

Program 11.3A
Program Requirement: To read a digit 0 to 9 from the keyboard and display on the seven-segment display.

The Display:



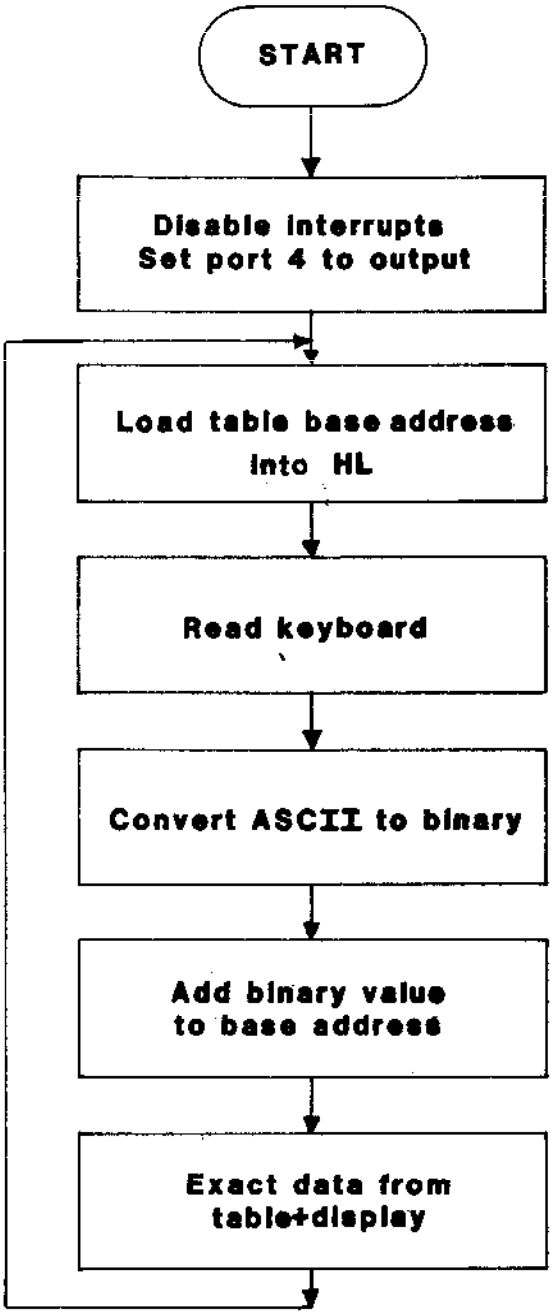
dp g f e d c b a
7 6 5 4 3 2 1 0
PIO Port 4

In order to display the character '1', segments b and c must be illuminated, ie. bits 1 and 2 of the output port must be set to 1, all others being 0. Thus, the output quantity must be 00000110 or 06H. The table below gives the required outputs for all the ten digits.

Digit	Segment								Hex
	dp	g	f	e	d	c	b	a	
0	0	0	0	0	0	0	0	0	00
1	0	0	0	0	0	1	1	0	06
2	0	1	0	1	1	0	1	1	5B
3	0	1	0	0	1	1	1	1	4F
4	0	1	1	0	0	1	1	0	66
5	0	1	1	0	1	1	0	1	6D
6	0	1	1	1	1	1	0	1	7D
7	0	0	0	0	0	1	1	1	07
8	0	1	1	1	1	1	1	1	7F
9	0	1	1	0	0	1	1	1	67

The program makes use of a look-up table; the output codes for the digits 0 to 9 are held, in order, in the table which starts in location TABLE. When the ASCII code for a digit is available it is converted to the binary number equal to the digit. Thus, if '5' is entered from the keyboard, the ASCII code for '5', 35H, is converted to numeric 5; this is done simply by subtracting 30H from the ASCII code. The number 5 is then added to TABLE so forming the address of the location holding the appropriate output code. This code is then output.

Flowchart:



Flowchart for Program 11.3A

PROGRAM 11.3A

```
LOC'N   CONTENTS    LABEL   OP CODE   ARGUMENTS   COMMENTS
0D00    3E 07               LD        A,07H       Disable
                                                  interrupt..
0D02    D3 06               OUT       (6),A       .on port 4
0D04    3E 0F               LD        A,0FH       Port 4
0D06    D3 06               OUT       (6),A       .is o/p.
0D08    21 00 0E    LOOP:   LD        HL,TABLE    Table base
0D0B    DF 7B               .BLINK                Get char.
0D0D    D6 30               SUB       30H
0D0F    85                  ADD       A,L         Form        o/p
                                                  address..
0D10    6F                  LD        L,A         ..in HL.
0D11    7E                  LD        A,(HL)      Get       output
                                                  code.
0D12    D3 04               OUT       (4),A
0D14    C3 08 0D            JP        LOOP        Again.
::::
::::
::::
::::
0E00    3F 06 5B                                  0,1,2
0E03    4F 66 6D                                  3,4,5
0E06    7D 07 7F                                  6,7,8
0E09    67                                        9
```

Execute the program and observe that the digits 0 to 9 may be written to the seven-segment display. Observe too that other keyed characters produce outputs. If, for example, character 'A' is keyed, its ASCII code of 41H is converted to 11H so that the output byte will be whatever happens to be in location 0E11. Good practice demands that the computer inputs be checked for validity and, if not valid, a warning or error message be given to the user. The next section shows how an illegal character may be made to produce the character 'E' on the display.

11.3.2.1 Modification to detect an illegal character.

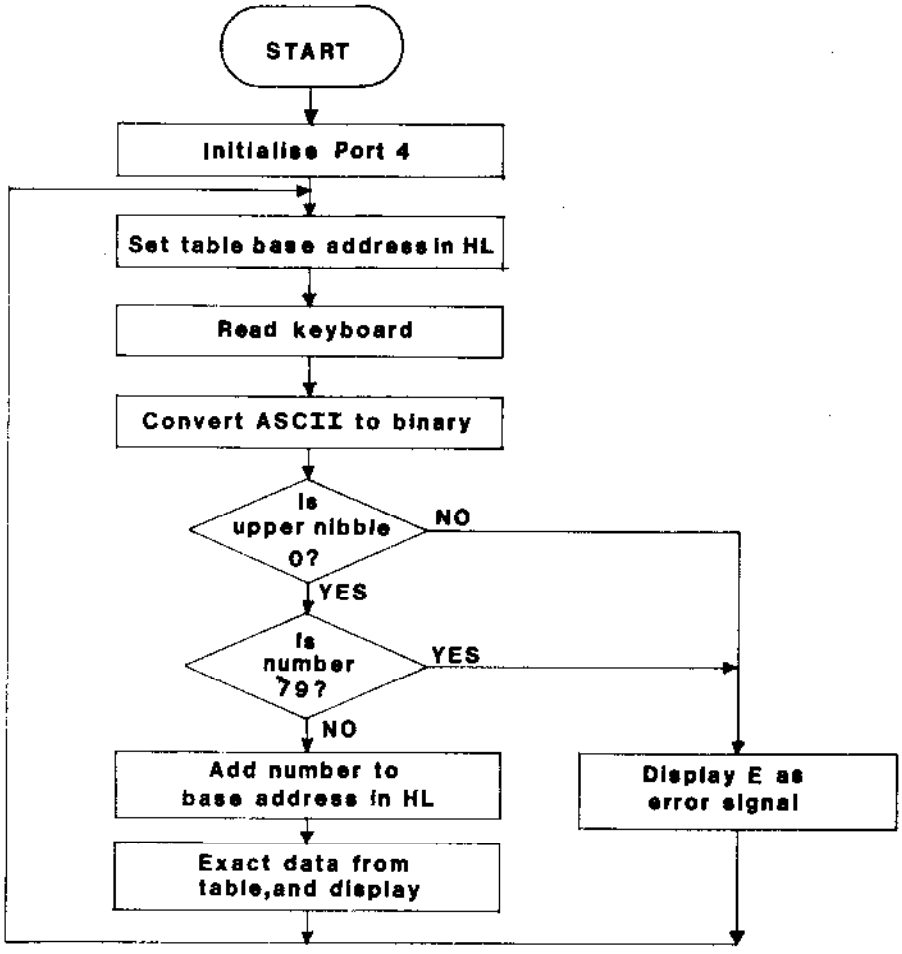
The possible ASCII codes for the keyboard lie within the range 00H to 7FH, but only codes 30H to 39H are legal codes, representing the digits 0 to 9. By subtracting 30H the possible codes lie within the range -30H to 4FH, legal codes being 00H to 09H. The legal codes all have their upper nibble equal to 0 and this forms the basis of the first test. Only codes 00H to 0FH pass this test. The remaining codes then simply have to be checked to see if they are greater than 9 or not. These tests are diagrammed below.

Possible ASCII codes:	00...2F	30...39	3A...3F	40...7F
Subtract 30H:	-30...-1	00...09	0A...0F	10...4F
Upper nibble =0?	NO	YES	YES	NO
Not greater than 9?		YES	NO	

Only the legal codes for 0 to 9 pass both tests. These tests are incorporated in the following program.

Program 11.3B
Program Requirement: As for Program 11.3A but with detection of illegal entry.

Flowchart:



Flowchart for Program 11.3B

PROGRAM 11.3B

LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0D00	3E 07		LD	A,07H	Disable ints on....
0D02	D3 06		OUT	(6),A	.port 4.
0D04	3E 0F		LD	A,0FH	O/p mode..
0D06	D3 06		OUT	(6),A	.port 4.
0D08	DF 7B	LOOP:	.BLINK		Read keyboard.
0D0A	21 00 0E		LD	HL,TABLE	Base address.
0D0D	D6 30		SUB	30H	To binary
0D0F	47		LD	B,A	Save it
0D10	E6 F0		AND	F0	Is upper nibble =0?
0D12	C2 23 0D		JP	NZ,ERROR	If yes, jump.
0D15	78		LD	A,B	Replace it
0D16	FE 0A		CP	0AH	>9?
0D18	F2 24 0D		JP	P,ERROR	Yes,jump.
0D1B	85		ADD	A,L	Add it to
0D1C	6F		LD	L,A	..base.
0D1D	7E		LD	A,(HL)	Get o/p..
0D1E	D3 04		OUT	(4),A	.& display
0D20	C3 08 0D		JP	LOOP	Again
0D23	3E 79	ERROR:	LD	A,79H	Code for E
0D25	D3 04		OUT	(4),A	.& display
OD27	C3 08 0D		JP	LOOP	Again

11.3.2.2 Adding a Flashing E

Error messages to the user of a computer program often need to be eye-catching. In the case of the segment display, the 'E' may be made to flash on and off. The program segment below replaces the ERROR routine in Program 11.3B, otherwise the program is unchanged. The modified routine blanks the display, waits a quarter-second, displays 'E', waits for a quarter-second, and repeats for a total of eight times before returning to read the keyboard again.

PROGRAM 11.3C
Program Requirement: As for Program 11.3B but with a flashing 'E' to indicate an illegal entry.

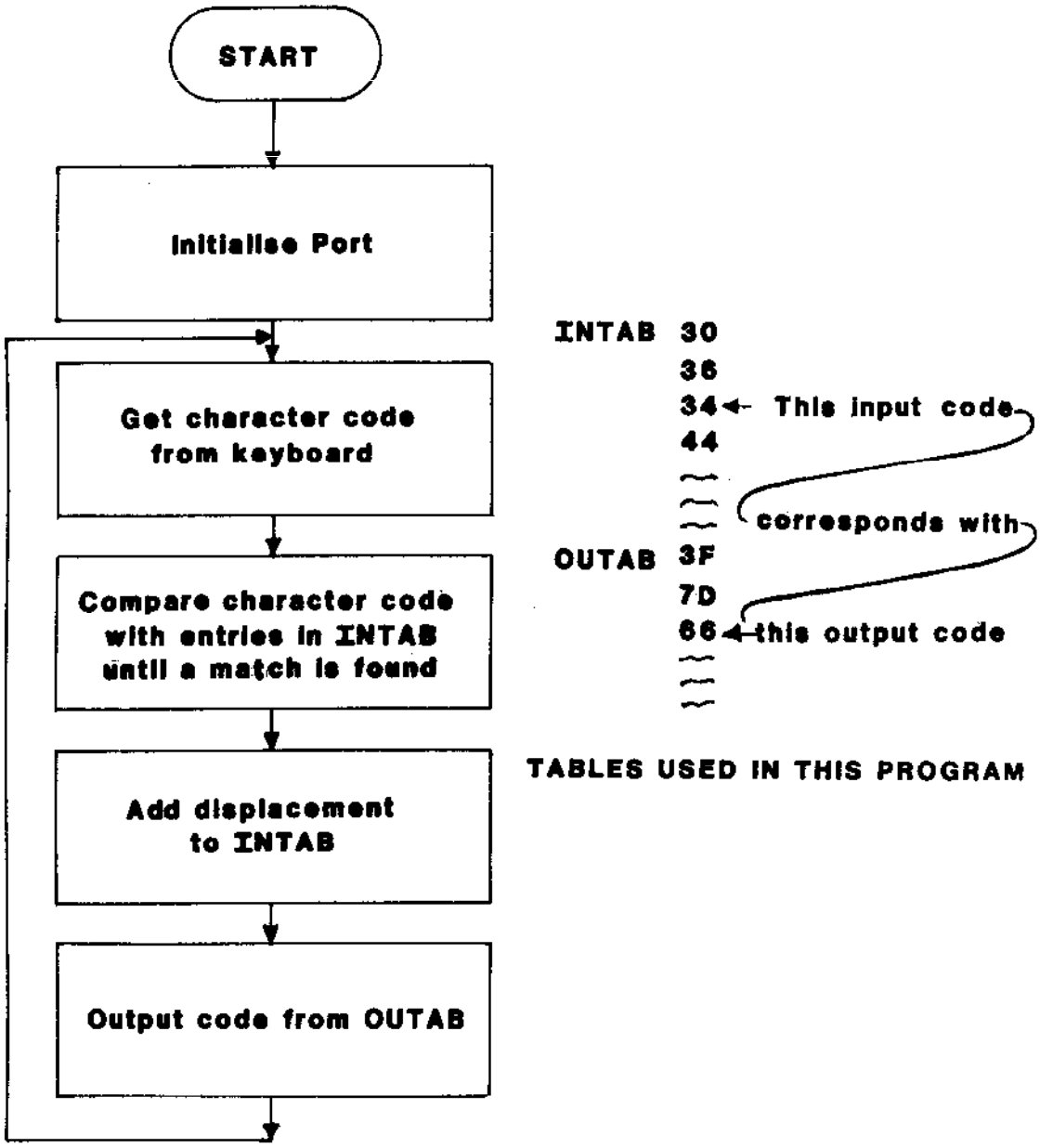
```
LOC'N  CONTENTS    LABEL    OP CODE   ARGUMENTS   COMMENTS
0D00   as for
::::   Program 11.3B
0D23   0E 08       ERROR:   LD        C,08H       Loop counter.
0D25   AF          RPEAT:   XOR       A           Blank..
0D26   D3 04                OUT       (4),A       ..display.
0D28   06 20                LD        B,20H       Delay..
0D2A   FF          DLY:     .RDEL                 ..
0D2B   10 FD                DJNZ      DLY         ..loop.
0D2D   3E 79                LD        A,79H       Load 'E'
0D2F   D3 04                OUT       (4),A       & display.
0D31   06 20                LD        B,20H       Delay
0D33   FF          DLY1:    .RDEL
0D34   10 FD                DJNZ      DLY1
0D36   0D                   DEC       C
0D37   C2 25 0D             JP        NZ,RPEAT
0D3A   C3 08 0D             JP        LOOP
```

11.3.2.3 An Alternative Solution -Table Searching

The previous programs to drive the seven-segment display make use of a single look-up table. This is convenient since the character code can be converted to a number in the range 0 to 9 and added to the table base address. Since the output table is arranged in order the correct output code is easily found. If this were not the case, two tables may be used; an input table consisting of all the valid codes (in any order) and an output table consisting of the corresesponding output codes in the same order as the input table. The code from the keyboard may then be compared in succession with all the entries in the input table and, when a match is found, the corresponding entry in the output table provides the appropriate code. The CPIR instruction greatly facilitates such table searches and its use is shown in the following program.

Program 11.3D
Program Requirement: To display the hexadecimal digits on the seven-segment display, showing 'E' for an illegal entry.

Flowchart:



Flowchart for Program 11.3D

```
LOC'N  CONTENTS    LABEL    OP CODE  ARGUMENTS   COMMENTS
0D00   3E 07               LD       A,07H       Disable
0D02   D3 06               OUT      (6),A       interrupt.
0D04   3E 0F               LD       A,0FH       Set port 4
0D06   D3 06               OUT      (6),A       .as output
0D08   DF 7B       READ:   .BLINK
0D0A   21 00 0E            LD       HL,INTAB    Compare..
0D0D   01 11 00            LD       BC,0011H    INTAB data
0D10   ED B1               CPIR                 til match.
0D12   11 10 00            LD       DE,0010H    Add disp'ment
0D15   19                  ADD      HL,DE       to HL.
0D16   7E                  LD       A,(HL)      Get o/p..
0D17   D3 04               OUT      (4),A       .& display
0D19   C3 08 0D            JP       READ
::::
::::
::::
::::
0E00   30 36 34    INTAB:                       0,6,4
0E03   33 35 31                                 3,5,1
0E06   37 38 32                                 7,8,2
0E09   39 44 41                                 9,D,A
0E0C   43 42 45                                 C,B,E
0E0F   46 00                                    F,x
0E11   3F 7D 66    OUTAB:                       Corresp
0E14   4F 6D 06                                 output
0E17   07 7F 5B                                 codes.
0E1A   67 5E 77
0E1D   58 7C 7B
0E20   71 79
```

The INTAB table contains all the valid input codes, in any order, while OUTAB contains the corresponding output codes, in the same order as INTAB.

The instructions in 0D0A, 0D0D, and 0D10 perform the search through INTAB. The content of the memory location pointed to by the register pair HL is compared with the content of register A. HL is incremented by 1, and the contents of register pair BC (Byte Count) is decremented by 1. If the decremented value of register pair BC is not zero and the content of the memory location does not match that of register A, then the instruction is repeated. Repetition continues until either register pair BC reaches zero or a match is found.

When a match is found, a displacement (in this case one less than the length of INTAB) is added to HL so that it points to the required output code. Should a match not be found, the CPIR will cease when register pair BC reaches zero, and the code, 79H, for the character 'E' will be output.

11.3.3 Position Encoder

The circuit shown in Appendix c allows the positions of two knobs to be read into the NASCOM. With the knob turned to one extremity the program will return a value of zero or near zero and, at the other extremity, a value of FF (or near) is returned.

Program 11.4
Program Requirement: To read the position of the knob controlling monostable 0 and to display on the screen a value indicating its position.

The knob position is read by the subroutine POTPOS0. This outputs a trigger pulse to monostable 0 and then reads the state of the monostable output pulse. All the time that this pulse is high, register C is incremented from 00. At the end of the monostable pulse the subroutine returns to the main program with the required value in register C.

ie.
1. Clear register C.
2. Trigger the monostable.
3. Increment register C.
4. Input the state of the monostable.
5. If the input is 1, go to step 3,
   else return to main program.

Since only one bit of output and one bit of input are required, it would be extravagant to use one PIO port for output and another for input as in previous programs. Instead, a single port is used in "mode 3"* This allows each of the eight bits of the port to be programmed as an input or output independent of the other pins. Mode 3 is selected by sending code CFH to the port control register, followed by another byte containing 1s where the corresponding bit is to be an input, and 0s where the port bit is to an output. Thus, in the initialisation sequence, code F0H is sent to the control register.

11110000=F0H

This programs bits 0 to 3 as outputs, and bits 4 to 7 as inputs. (The subroutine outputs the trigger pulse on bit 0 and reads the monostable on bit 7. Although only two bits are required, all eight bits must have a defined data direction.)

* When all the port pins are outputs as in previous programs, the port is in "mode 0". When all pins are inputs, the port is in "mode 1".

PROGRAM 11.4

```
LOC'N  CONTENTS    LABEL     OP CODE   ARGUMENTS   COMMENTS
0D00   3E 07               LD        A,07H       Disable
0D02   D3 06               OUT       (6),A       interrupts
0D04   3E CF               LD        A,CFH       Mode 3
0D06   D3 06               OUT       (6),A       .on port 4
0D08   3E F0               LD        A,F0H       Upper nibble is
                                                 input.
0D0A   D3 06               OUT       (6),A       Lower nibble is
                                                 output.
0D0C   CD 00 0E   INPUT:   CALL      POTPOS0
0D0F   79                  LD        A,C
0D10   DF 68               .B2HEX
0D12   DF 6A               .CRLF
0D14   06 B9               LD        B,B9H       One..
0D16   FF         DEL:    .RDEL                 ..second
0D17   10 FD               DJNZ      DEL         ..delay.
0D19   C3 0C 0D            JP        INPUT       Again
::::
::::           ;Subroutine to read pot 0.
::::           ;Return value is in register C.
::::
0E00   3E 01       POTPOS0:LD        A,01H       Produce
0E02   D3 04               OUT       (4),A       negative
0E04   AF                  XOR       A           trigger
0E05   D3 04               OUT       (4),A       pulse
0E07   3C                  INC       A           on
0E08   D3 04               OUT       (4),A       bit 0.
0E0A   0E 00               LD        C,00H
0E0C   DB 04       LOOP:   IN        A,(4)
0E0E   A7                  AND       A
0E0F   F0                  RET       P           Pulse done
0E10   0C                  INC       C
0E11   00                  NOP
0E12   00                  NOP
0E13   C3 0C 0E            JP        LOOP
```

The initialisation sequence, from the start down to INPUT, sets port 4 to mode 3 with bits 0 to 3 as outputs, bits 4 to 7 as inputs. On return from subroutine POTPOS0, register C contains the required value which is displayed and, after one second delay, the program repeats.

Subroutine POTPOS0 begins by outputting the sequence 1,0,1 (ie. a negative trigger pulse) on bit 0 to monostable 0. Register C is then cleared and LOOP increments register C until the monostable output becomes 0. Since the input from the monstable is in bit 7, the end of the pulse is detected by the input byte being positive, so that return to the main program may be by the RET P (RETurn if Positive) instruction. Note that since the IN A,(4) istruction does not affect the state of the flags, the AND A instruction is necessary. This does not alter the contents of register A but it does alter

the flags.

The NOP (No OPeration) instructions have been included to increase the time taken to increment register C. More or less NOPs may be inserted so that, when the knob is turned to give maximum pulse length, the maximum count in register C is not more than FFH.

Program 11.4 could be modified to read monostable 1 instead but it is more convenient to have a single subroutine which reads both monostables. Such a routine is incorporated into Program 11.5 On return from this subroutine, registers D and E contain the values from monostables 0 and 1 respectively. The main program simply displays the two readings side by side on the screen.

POTRD produces a negative trigger pulse to both monostables and then increments registers D and E from zero until the respective monostable pulse ends. This is detected by using the bit test instructions, BIT b,R, which set the Zero flag if the bit tested is 0. To detect when both monstables have timed out, the input byte is ANDed with C0H and a return is made if the result is zero.

Program 11.5
Program Requirement: To show the use of a subroutine which reads both monostable 0 and 1.

```
LOC'N  CONTENTS    LABEL    OP CODE   ARGUMENTS   COMMENTS
0D00   3E 07               LD        A,07H       Disable
0D02   D3 06               OUT       (6),A       ints
0D04   3E CF               LD        A,CFH       & mode 3
0D06   D3 06               OUT       (6),A       .on port 4
0D08   3E F0               LD        A,F0H       Upper nibble is
                                                 input.
0D0A   D3 06               OUT       (6),A       Lower nibble is
                                                 output.
0D0C   CD 00 0E   INPUT:   CALL      POTRD
0D0F   7A                  LD        A,D
0D10   DF 68               .B2HEX                Display
0D12   DF 69               .SPACE                both
0D14   7B                  LD        A,E         values on
0D15   DF 68               .B2HEX                one line.
0D17   DF 6A               .CRLF
0D19   06 B9               LD        B,B9H       Delay
0D1B   FF          DEL:    .RDEL                 one
0D1C   10 FD               DJNZ      DEL         second.
0D1E   C3 0C 0D            JP        INPUT       Again.
```

```
                        ;Read both monostables.
                        ;Return monostable 0 in register D,
                        ;and monostable 1 in register E.
                        ;
0E00    16 00   POTRD:  LD      D,00H       Clear D
0E02    5A              LD      E,D         and E.
0E03    3E 03           LD      A,03H       Output
0E05    D3 04           OUT     (4),A       negative
0E07    AF              XOR     A           trigger
0E08    D3 04           OUT     (4),A       pulse
0E0A    3E 03           LD      A,03H       to both
0E0C    D3 04           OUT     (4),A       m/s.
0E0E    DB 04   LOOP:   IN      A,(4)       Read.
0E10    CB 7F           BIT     7,A         Pulse 0
0E12    CA 16 0E        JP      Z,DONE      done?
0E15    14              INC     D           No.
0E16    CB 77   DDONE:  BIT     6,A         Pulse 1
0E18    CA 1C 0E        JP      Z,EDONE     done?
0E1B    1C              INC     E           No.
0E1C    E6 C0   EDONE:  AND     C0          Both done?
0E1E    C8              RET     Z           Yes.
0E1F    C3 0E 0E        JP      LOOP        No.
```

Obvious applications of the dual position converter include using the two knobs to control the 'bats' in a game of 'tele-tennis' and to generate XY coordinates for graphics displays.

11.4 Digital-to-Analogue Converter

The circuit described in Appendix c can be used both to output an analogue voltage from the NASCOM 1 and to input an analogue voltage. This section describes its use as a digital-to-analogue converter, DAC, ie. a device to output an analogue voltage.

Program 11.6
Program Requirement: To input a digit 0 to 8 inclusive and output a voltage proportional to the input.

The analogue output voltage is in the range 0 to 4V approximately, corresponding to a binary input of 00H to FFH. In order to use the full range of the analogue voltage output, the input digit (0 to 8) is multiplied by 32d to obtain the required binary output. This is simply achieved by shifting the number left five times. This, however, results in an input of 8 attempting to produce a binary output of 256d which is overflow. This is detected and an output of 255d (=FFH) generated instead.

The analogue output voltage is most easily observed with

a voltmeter. However, the circuit in Appendix c allows the brightness of a torch-bulb or the no-load speed of a model-makers electric motor to be controlled. (It should be noted that there are purely digital methods for controlling such devices. These essentially switch the power to the device on and off rapidly, using however the same circuit as given here.)

PROGRAM 11.6

LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0D00	3E 07		LD	A,07H	Disable..
0D02	D3 07		OUT	(7),A	..ints
0D04	3E 0F		LD	A,0FH	& mode 0
0D06	D3 07		OUT	(7),A	on port 5.
0D08	DF 7B	IP:	.BLINK		
0D0A	D6 30		SUB	30H	
0D0C	CB 27		SLA	A	
0D0E	CB 27		SLA	A	
0D10	CB 27		SLA	A	
0D12	CB 27		SLA	A	
0D14	CB 27		SLA	A	
0D16	D2 1B 0D		JP	NC,OP	
0D19	3E FF		LD	A,FFH	
0D1B	D3 05	OP:	OUT	(5),A	
0D1D	C3 08 0D		JP	IP	

11.5 Analogue-to-Digital Converter

This section describes how the circuit described in Appendix c can be used to input an analogue voltage, such as may be obtained from the 10k potentiometer.

There are two or three ways of using the circuit as an ADC. However, they all depend on generating an output from the port to the DAC so that the corresponding analogue voltage from it is just a little larger than the voltage at the analogue voltage input. The comparator output will then be logical 1 and this signifies that the conversion is complete. The most straightforward method is to output a binary count up from zero until the comparator output goes to 1. The binary number at that point is proportional to the input voltage. This however is relatively slow; a much quicker method is that of "successive approximation".

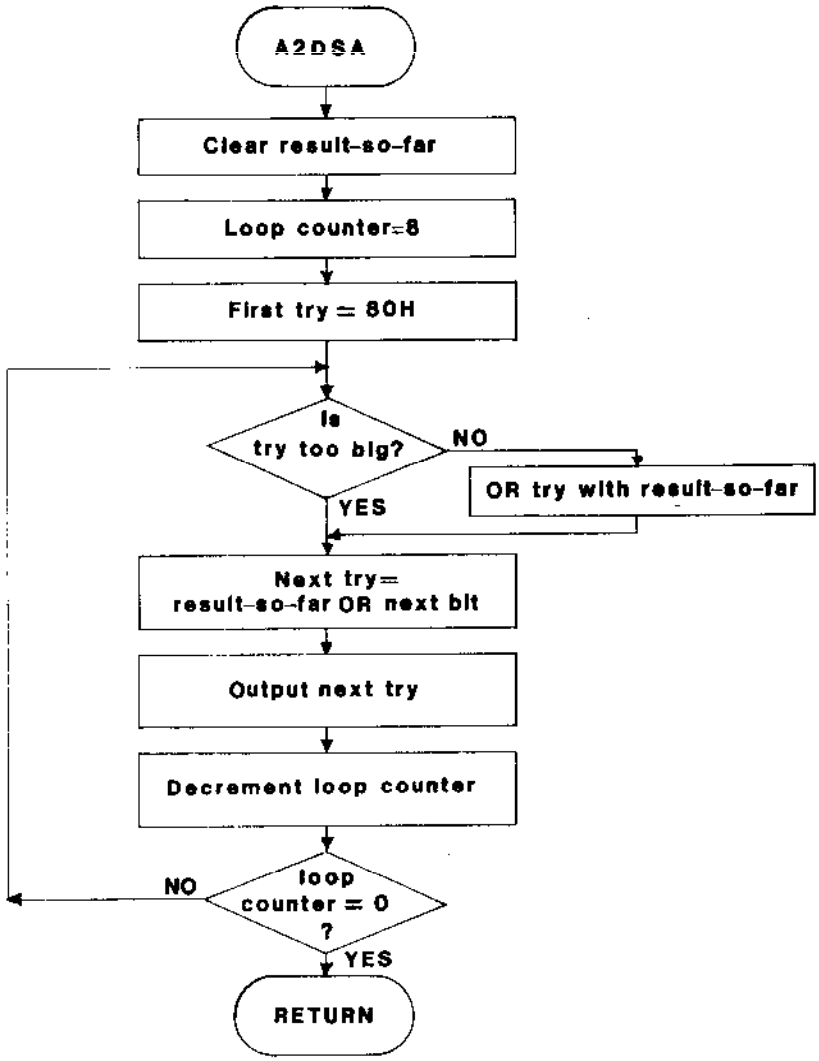
Suppose, for simplicity, that only a 3 bit conversion is required, and that binary output 100 produces 2V from the DAC. Output 010 would then produce 1V, and 001 would produce 0.5V. Further, assume that the analogue input is 2.6V.

The first approximation would be 100 producing 2V from the DAC and hence a comparator output of logical 0 (="too small"). Since the approximation is too small, the most significant bit of the required result is 1. The next try would therefore be 110, producing 3V (ie.2V + 1V) from the DAC and a comparator output of logical 1 (="too large").

This second bit should therefore be 0 and the approximation-so-far is 100. The final try would be 101 producing 2.5V from the DAC and logical 0 from the comparator. The final approximation would therefore be 101.

Note that for a 3 bit conversion only 3 succssive approximations are required. For an 8 bit converter, 8 steps are required. The conversion routine is written as a subroutine, returning the required value in register C. The initialisation of the PIO is at the beginning of the main routine.

Flowchart:



Flowchart for Program 11.7

PROGRAM 11.7

```
LOC'N  CONTENTS    LABEL    OP CODE   ARGUMENTS   COMMENTS
0D00   3E 4F               LD        A,4FH       Mode 1
0D02   D3 06               OUT       (6),A       on port 4.
0D04   3E 0F               LD        A,0FH       Mode 0
0D06   D3 07               OUT       (7),A       on port 5.
0D08   CD 00 0E   BEGIN:   CALL      A2DSA
0D0B   79                  LD        A,C         Display..
0D0C   DF 68               .B2HEX                ..result.
0D0E   DF 6A               .CRLF
0D10   06 B9               LD        B,B9H       One..
0D12   FF         DEL:    .RDEL                 ..second
0D13   10 FD               DJNZ      DEL         ..delay.
0D15   C3 08 0D            JP        BEGIN       Again.
::::               ;
::::               ;Analogue-to-digital subroutine.
::::               ;Result in register C.
::::               ;
0E00   0E 00       A2DSA:  LD        C,00H       Clear result
                                                 register.
0E02   16 08               LD        D,08H       Loop count
0E04   06 80               LD        B,80H       First try.
0E06   78                  LD        A,B         Output..
0E07   D3 05               OUT       (5),A       ..1st try.
0E09   DB 04       COMPIN: IN        A,(4)       Too..
0E0B   A7                  AND       A           ..big?
0E0C   F2 12 0E            JP        P,NEXT      Yes, jump.
0E0F   78                  LD        A,B         No.
0E10   B1                  OR        C
0E11   4F                  LD        C,A         Result-so-far.
0E12   CB 08       NEXT:   RRC       B           Next bit.
0E14   78                  LD        A,B         OR it..
0E15   B1                  OR        C           with r-s-f
0E16   D3 05               OUT       (5),A       Next try.
0E18   15                  DEC       D           Loop done?
0E19   C2 09 0E            JP        NZ,COMPIN   No, jump.
0E1C   C9                  RET                   Yes,return
```

11.6 Waiting for a Device Flag.

In the programs so far, the devices (ie. the leds, switches, and converters) are always ready to take part in a data transfer. This may not always be the case; for example, a paper tape punch takes a relatively long time to punch one byte and will not be ready to receive a further byte until it has completed the punching of the previous one. Such 'slow' devices have to generate a signal to indicate that they are ready for a data transfer. In the following program switches 0,1,& 2 are used to input data to the program but the transfer of data will only take place when the 'device flag' is active. The device flag in this program actually comes from Switch 3 which is driven by hand: usually, the device

itself will activate the flag automatically. Thus, a paper-tape reader will automatically activate its flag to signal that it has moved the paper-tape onto the next character and is ready to have it read.

Program 11.8
Program Requirement: To read Switches 0,1,& 2 when the device flag (=Switch 3) is active. The data input of a number 0 to 7 is then displayed on the screen.

A byte of data is read from port 5 into register A and then bit 3 (from Switch 3) tested. If the bit is 0, indicating that the device is not ready, the program reads the port again. This short loop will be executed very many times until Switch 3 goes to 1. (Flick the switch only momentarily to 1, otherwise several data transfers will take place.)

When the data is read it is converted to its ASCII code first by setting all bits other than the data bits to 0 and then ORing the result with 30H. The resulting ASCII code is displayed using subroutine CRT, and the sequence repeated.

PROGRAM 11.8

LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0D00	3E 07		LD	A,07H	Disable..
0D02	D3 07		OUT	(7),A	..int
0D04	3E 4F		LD	A,4FH	& mode 1..
0D06	D3 07		OUT	(7),A	on port 5.
0D08	DB 05	SWRDY:	IN	A,(5)	Input.
0D0A	CB 5F		BIT	3,A	Flag set?
0D0C	CA 08 0D		JP	Z,SWRDY	No, jump.
0D0F	E6 07		AND	07H	Yes.
0D11	F6 30		OR	30H	Convert.
0D13	F7		.ROUT		Display.
0D14	DF 6A		.CRLF		
0D16	06 30		LD	B,30H	Quarter..
0D18	FF	DEL:	.RDEL		..second..
0D19	10 FD		DJNZ	DEL	..delay.
0D1B	C3 08 0D		JP	SWRDY	Again.

The same principle applies when outputting data to a slow device, eg. a paper-tape punch. The punch must signal that it has finished punching the previous data and is ready to receive the next byte to be punched. This is illustrated in the following program, in which the 'punch' is the leds and the 'punch flag' is Switch 0.

Program 11.9
Program Requirement: To display a four bit binary count on the screen and transfer the data to the leds when the leds flag is set.

Since only four bits of output (the data) and one bit of input (the leds flag) are required, it is possible to use just one I/O port in mode 3. Although this requires only a simple reconnection of Switch 0 to one of the unused bits of port 4, the following program requires no rewiring since it uses two ports, one for input and the other for output. Many data transfers are of eight bits anyway and so demand the use of two ports.

Switch 0 is used as the leds flag. This provides an alternative method of testing the flag to the BIT test instruction used in the previous program. Here the switch data is rotated right instead so that bit 0 enters the Carry flag. As before, flag switch 0 must be flicked only momentarily to 1 to avoid multiple transfers.

PROGRAM 11.9

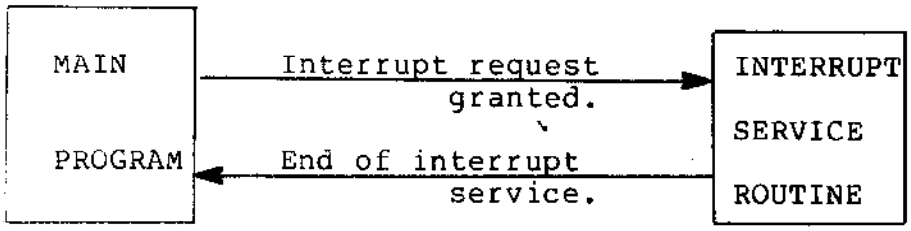
```
LOC'N  CONTENTS    LABEL     OP CODE  ARGUMENTS    COMMENTS
0D00   3E 07               LD       A,07H        Disable..
0D02   D3 06               OUT      (6),A        ..both..
0D04   D3 07               OUT      (7),A        ..ints.
0D06   3E 0F               LD       A,0FH        Mode 0..
0D08   D3 06               OUT      (6),A        .on port 4
0D0A   3E 4F               LD       A,4FH        Mode 1..
0D0C   D3 07               OUT      (7),A        .on port 5
0D0E   0E 00               LD       C,00H        Clear C
0D10   79          DISPLAY:LD       A,C          Display..
0D11   E6 0F               AND      0FH          ..binary
0D13   DF 68               .B2HEX                ..count.
0D15   DF 6A               .CRLF                 in reg C.
0D17   DB 05       LEDSRDY:IN       A,(5)        Input
0D19   1F                  RRA                   Flag set?
0D1A   D2 17 0D            JP       NC,LEDSRDY   No, jump.
0D1D   79                  LD       A,C          Yes,..
0D1E   D3 04               OUT      (4),A        output.
0D20   06 30               LD       B,30H        Quarter..
0D22   FF          DEL:    .RDEL                 ..second
0D23   10 FD               DJNZ     DEL          ..delay.
0D25   0C                  INC      C
0D26   C3 10 0D            JP       DISPLAY      Again.
```

The data transfer method used in Programs 11.8 and 11.9 use the cpu to constantly query the external device "Are you ready for another data transfer?". It does this by reading the device flag over and over again until it indicates that the device is ready. This technique is called 'polling'; it is the simplest method of communicating with a slow device and is adequate for most microprocessor applications.

However, the cpu does spend much of its time waiting for the ready signal and this represents a waste of processing capability. An alternative method, that of interrupt driven I/O, is described below and it will be seen that the cpu can be allowed to continue with other processing tasks (like performing calculations on the data previously input) while the external device is not yet ready.

11.7 Interrupt Driven I/O

The two following programs show how the cpu may be programmed to carry out one task continually until interrupted when a second task, that of input or output, is performed. On completion of the second task, the cpu continues from where it left off the first task. The first task is carried out by the main program and the second is carried out by the interrupt service routine. The scheme is illustrated below.



General Scheme for a Single Interrupt

The interrupt request is generated manually by pressing the button on the pulser circuit described in Appendix c.

Program 11.10
Program Requirement: To display a binary count on the screen. When an interrupt request is made the value of the switches will be displayed.

The switches are connected to PIO port 5. This is port B of the PIO and, in order to make an interrupt request to this port, the output of the pulser must be connected to the pin labelled $\overline{\text{BSTB}}$ on the PIO.

PROGRAM 11.10

```
LOC'N  CONTENTS    LABEL    OP CODE   ARGUMENTS   COMMENTS
0D00   ED 5E       INIT:    IM        2           Cpu int
0D02   FB                   EI                    enable.
0D03   3E 0E                LD        A,0EH       Interrupt
0D05   ED 47                LD        I,A         vector
0D07   3E 00                LD        A,00H       is
0D09   D3 07                OUT       (7),A       0E00.
0D0B   3E 4F                LD        A,4FH       Port 5
0D0D   D3 07                OUT       (7),A       is i/p.
0D0F   3E 87                LD        A,87H       Enable
0D11   D3 07                OUT       (7),A       PIO int.
0D13   0C          MAIN:    INC       C           Display
0D14   79                   LD        A,C         binary
0D15   DF 68                .B2HEX                count.
0D17   DF 6A                .CRLF                 on screen.
0D19   06 B9                LD        B,B9H       One
0D1B   FF          DEL:     .RDEL                 second
0D1C   10 FD                DJNZ      DEL         delay.
0D22   C3 13 0D             JP        MAIN        Again.
::::                ;
0E00   00 0F        ;Interrupt service routine
::::                ;begins at location 0F00.
::::                ;
                    ;Interrupt service routine.
                    ;Input switches to screen.
0F00   EF           SERVIN: .PRS                  Display
0F01   49 6E 74             I n t                 message.
0F04   65 72 72             e r r
0F07   75 70 74             u p t
0F0A   21 00                !  00
0F0C   DB 05                IN        A,(5)       Input
0F0E   E6 0F                AND       0FH         Convert
0F10   F6 30                OR        30H
0F12   F7                   .ROUT                 Display
0F13   DF 6A                .CRLF
0F15   FB                   EI                    Enable ints
0F16   ED 4D                RETI
```

Execute the program and observe that a binary count is displayed on the screen, incrementing at one second intervals. This is the program MAIN. Whenever the interrupt button is pressed, observe that the message 'Interrupt! x' appears, where x depends on the switch settings. TAKE CARE that the program is entered corectly before attempting to execute it because an error may result in the interrupt being granted but not serviced corectly. (One symptom of this is that the interrupt will be serviced once only. This may possibly be corrected by executing the following code from F80:

```
0F80        CD 8B 0F
0F83        3E 03
0F85        D3 06
0F87        D3 07
0F89        DF 5B
0F8B        ED 4D
```

If this fails power-off and re-enter the program correctly.)

To use the interrupt facility of the PIO, the Z80 must itself be placed in its own interrupt mode 2, and its interrupt facility enabled. This is achieved by the instructions IM 2 and EI. It is then necesary to inform the cpu where, in memory, it can find the address of the start of the interrupt routine. In this example, this address is held in memory locations 0E00 and 0E01; the required 'vector' is 0E00. The upper byte, 0E, is loaded into register I of the cpu and the lower byte, 00, is sent to the PIO control port. (The vector must be an even number.) The remaining instructions of the INITialisation part of the program set port 5 to be an input port and enable the interrupt facility on the port itself.

In response to an interrupt request, the PIO sends the low byte of the vector, 00, back to the cpu which uses it together with the contents of register I, 0E, to form the required vector, 0E00. From that memory location (and the one immediately following) the cpu obtains the starting address of the service routine, 0F00.

The service routine, SERVIN, simply displays a short message, inputs the value of the switches, converts it to ASCII code, and displays the corresponding character. Since the granting of an interrupt request automatically disables the interrupt facility of the cpu, it is necessary to re-enable it with EI before returning to the main program with RETI.

Program 11.11
Program Requirement: To display a binary count on the screen. When an interrupt request is made, the current count is displayed on the screen and the lower nibble sent to the leds.

The output to the leds is from port 4 (port A) of the PIO so the interrupt request will be made from this port. The pulser output must therefore be connected to the pin labelled $\overline{\text{ASTB}}$ on the PIO.

This program is very similar to the previous one; some bytes in the INIT part of the program have been changed to refer to port 4 instead of port 5 and a few instructions in the service routine have been changed to cause output to PIO port 4.

Again, observe the binary count on the screen and,

following an interrupt request from the pulser, the low byte of the current count on the leds. Note that, while the output to the leds is almost always the same as that displayed on the screen, it is possible for it to be one count ahead. This arises if the interrupt request is granted at the end of the INC C instruction at location MAIN.

PROGRAM 11.11

```
LOC'N  CONTENTS    LABEL    OP CODE   ARGUMENTS   COMMENTS
0D00   ED 5E       INIT:    IM        2           Mode 2
0D02   FB                   EI                    cpu int.
0D03   3E 0E                LD        A,0EH       Vector
0D05   ED 47                LD        I,A         is
0D07   3E 00                LD        A,00H       0E00.
0D09   D3 06                OUT       (6),A
0D0B   3E 0F                LD        A,0FH       Port 4
0D0D   D3 06                OUT       (6),A       is o/p.
0D0F   3E 87                LD        A,87H       Enable
0D11   D3 06                OUT       (6),A       PIO int.
0D13   0C          MAIN:    INC       C           Display
0D14   79                   LD        A,C         up
0D15   DF 68                .B2HEX                count on
0D17   DF 6A                .CRLF                 screen.
0D19   06 B9                LD        B,B9H       One
0D1B   FF          DEL:     .RDEL                 second
0D1C   10 FD                DJNZ      DEL         delay.
0D1E   C3 13 0D             JP        MAIN        Again.
::::
0E00   00 0F                ;Interrupt service routine
::::                        ;begins at location 0F00.
::::                        ;
::::                        ;Interrupt service routine.
::::                        ;Output register C to leds.
::::                        ;
0F00   EF          SERVOUT: .PRS                  Display
0F01   49 6E 74             I n t                 message.
0F04   65 72 72             e r r
0F07   75 70 74             u p t
0F0A   20 00                (sp) 00
0F0C   79                   LD        A,C
0F0D   D3 04                OUT       (4),A       O/p reg C.
0F0F   DF 68                .B2HEX                Display reg  C.
0F11   DF 6A                .CRLF
0F13   FB                   EI
0F14   ED 4D                RETI
```

11.7.1 Bit Mode Interrupt

The PIO mode 3 allows the data direction of each bit of the port to be programmed independently of the others; this was described in Chapter 11.3.3. When the interrupt facility is also enabled, each bit may be programmed to generate an interrupt request.

Program 11.12
Program Requirement: To display a binary count on the screen. When any of the switches is at logic 1 an interrupt is generated and the service routine indicates which switch caused the interrupt.

The main program initialises PIO port 5 to which the switches are connected and then enters the COUNT loop which simply displays a two byte count on the screen. The initialisation commences by setting the cpu to its interrupt mode 2 and enabling the interrupt facility. The interrupt vector is 0C50 and the upper byte, 0C, is transferred to cpu register I. Five bytes now have to be transferred to control port 7 and it is most convenient to transfer them using the OTIR instruction rather than loading each byte into register A and transferring it with the OUT instruction. The control bytes are stored in a block at locations 0C52 to 0C56 inclusive; register pair HL is loaded with the starting address of the block, register B with the number of bytes in the block, and register C with the port number to which the bytes are to be transferred. Instruction OTIR then transfers all five bytes.

The first control byte, CF, simply selects mode 3 on the PIO. The next byte,0F(=00001111), sets bits 0 to 3 as inputs and bits 4 to 7 as outputs. (Refer to 11.3.3.) The third byte, 50, is the low byte of the interrupt vector.

The fourth byte, B7(=10110111), is the interrupt control byte. Bit 7 set to 1 enables the port interrupt faclity. (0 in this bit would disable the faclity.) Bit 6 set to 0 programs any one of the selected bits (see next paragraph) to generate an interrupt request.(1 here would indicate that **all** the selected bits would have to be active to generate an interrupt request.) Bit 5 set to 1 defines an active input as one that is logic 1. (0 would make logic 0 the active level.)

The final byte, F0(=11110000), has 0s in it wherever a bit is selected as one that can cause an interrupt request. Thus, bits 0 to 3 (which are connected to the switches) are the bits selected to cause interrupts. The initialisation is now complete.

The interrupt service routine inputs the data on the port, masks off the upper nibble, and saves it in register D. The message 'ALARM STATE:' is displayed folowed by the value of the switches. Bits 0 to 3 in register D are then tested

in turn: if the bit is 1 the message 'Alarm x active' is displayed, where x is the number of the switch causing the interrupt. The screen is then scrolled up six lines, the interrupt faclity re-enabled, and a return made to the main program.

PROGRAM 11.12

```
LOC'N  CONTENTS    LABEL   OP CODE  ARGUMENTS  COMMENTS
0C50   00 0E       ;Vector.
                   ;PIO control bytes.
0C52   CF          ;Mode 3
0C53   0F          ;I/O select:  low nibble is output.
0C54   50          ;Low byte of interrupt vector.
0C55   B7          ;Interrupt control:
                   ;   Enable interrupt
                   ;       Any input to generate interrupt
                   ;   Inputs active high
                   ;   Mask follows.
0C56   F0          ;Bits 0 to 3 to generate interrupt.
::::
::::
0D00   ED 5E       INIT:   IM       2          Mode 2
0D02   FB                  EI                  cpu int.
0D03   3E 0C               LD       A,0CH      Upper byte
0D05   ED 47               LD       I,A        of vector.
0D07   21 52 0C            LD       HL,0C52H   Transfer
0D0A   06 05               LD       B,05H      control
0D0C   0E 07               LD       C,07H      bytes to
0D0E   ED B3               OTIR                port 7.
0D10   0C          COUNT:  INC      C          Display
0D11   79                  LD       A,C        up
0D12   DF 68               .B2HEX              count on
0D14   DF 6A               .CRLF               screen.
0D16   06 B9               LD       B,B9H      One
0D18   FF          DEL:    .RDEL               second
0D19   10 FD               DJNZ     DEL        delay.
0D1B   C3 10 0D            JP       COUNT      Again.
::::                       ;
::::                       ;
::::                       ;Interrupt service routine.
::::                       ;
0E00   DB 05               IN       A,(5)      Read.
0E02   E6 0F               AND      0FH        Mask off.
0E04   57                  LD       D,A        Save in D.
0E05   EF                  .PRS                Display
0E06   41 4C 41            A L A               message.
0E09   52 4D 20            R M (sp)
0E0C   53 54 41            S T A
0E0F   54 45 3A            T E :
0E12   20 00               (sp) 00
0E14   7A                  LD       A,D
0E15   DF 68               .B2HEX
```

```
0E17   DF 6A              .CRLF
0E19   DF 6A              .CRLF
0E1B   CB 42              BIT     0,D       Alarm 0?
0E1D   CA 32 0E           JP      Z,A1      No, jump.
0E20   EF                 .PRS              Yes,
0E21   41 6C 61           A l a             display
0E24   72 6D 20           r m (sp)          message.
0E27   30 20 61           0 (sp) a
0E2A   63 74 69           c t i
0E2D   76 65 00           v e 00
0E30   DF 6A              .CRLF
0E32   CB 4A        A1:   BIT     1,D       Alarm 1?
0E34   CA 49 0E           JP      Z,A2      No, jump
0E37   EF                 .PRS              Yes,
0E38   41 6C 61           A l a             display
0E3B   72 6D 20           r m (sp)          message.
0E3E   31 20 61           1 (sp) a
0E41   63 74 69           c t i
0E44   76 65 00           v e 00
0E47   DF 6A              .CRLF
0E49   CB 52        A2:   BIT     2,D       Alarm 2?
0E4B   CA 60 0E           JP      Z,A3      No, jump.
0E4E   EF                 .PRS              Yes,
0E4F   41 6C 61           A l a             display
0E52   72 6D 20           r m (sp)          message.
0E55   32 20 61           2 (sp) a
0E58   63 74 69           c t i
0E5B   76 65 00           v e 00
0E5E   DF 6A              .CRLF
0E60   CB 5A        A3:   BIT     3,D       Alarm 3?
0E62   CA 75 0E           JP      Z,SCROLL  No, jump.
0E65   EF                 .PRS              Yes,
0E66   41 6C 61           A l a             display
0E69   72 6D 20           r m (sp)          message.
0E6C   33 20 61           3 (sp) a
0E6F   63 74 69           c t i
0E72   76 65 00           v e 00
0E75   06 06        SCROLL: LD    B,06H     Six
0E77   DF 6A        LF:   .CRLF             line
0E79   10 FD              DJNZ    LF        feeds.
0E7B   FB                 EI
0E7C   ED 4D              RETI
```

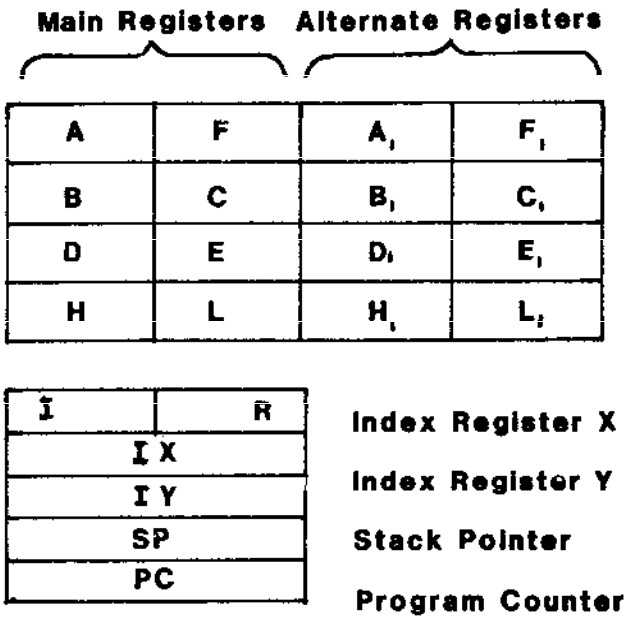
Ensure that the program is entered correctly, set all the switches to 0, and execute the program from location 0D00. Observe the count on the screen then flick one of the switches to 1 and back to 0. The appropriate message will appear on the screen and the count will continue. Note that, if a switch is left at 1, other switches have no effect. This is because the condition to generate an interrupt is already satisfied by the switch left at 1.

If the byte in location 0C55 is changed from B7 to F7, the condition to generate an interrupt will be that all the

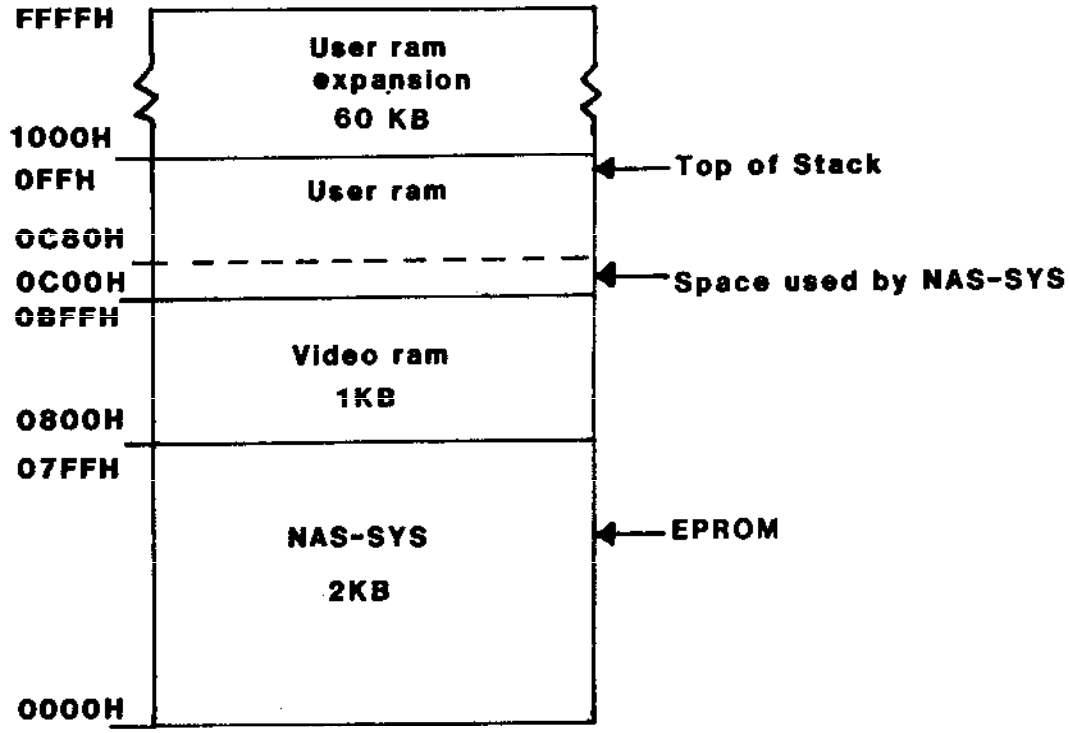
switches have to be 1. Setting bit 5 of this byte to 0 will change the active level of the switches from 1 to 0, ie. the switches should normally be at 1, going to 0 to generate an interrupt request.

MACHINE CODE PROGRAMMING
for the NASCOM 1 & 2

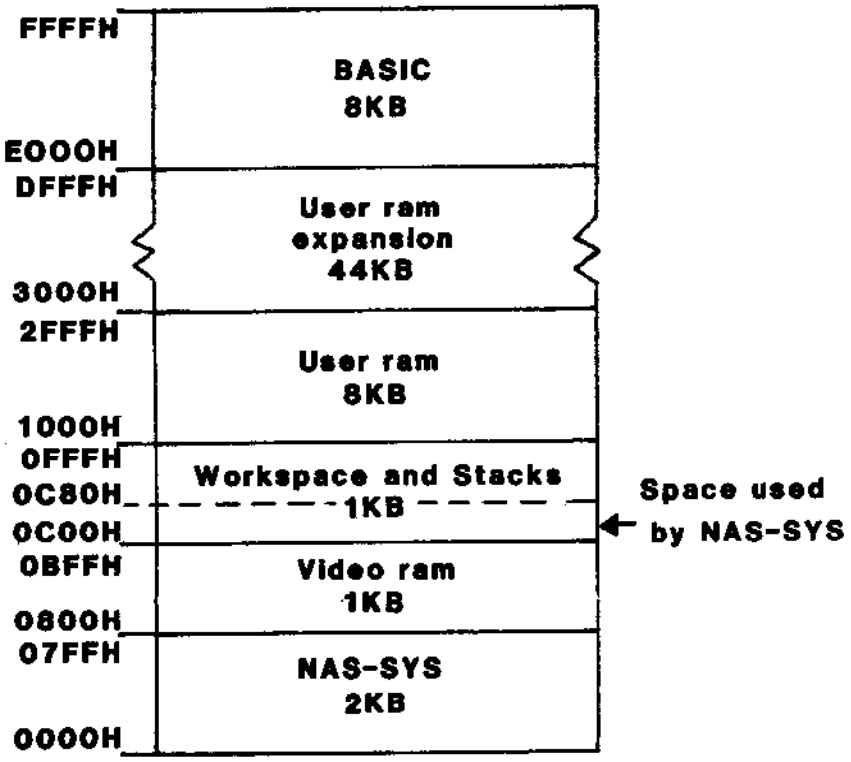
Appendix a
CPU Registers and Memory Maps



CPU Registers



Memory Map for NASCOM 1



Memory Map for NASCOM 2

MACHINE CODE PROGRAMMING
for the NASCOM 1 &2

Appendix b
Display of CPU Registers

SINGLE STEPPING

Single stepping through a program allows the programmer to observe the contents of the cpu registers after the execution of each instruction.

The register contents are displayed as follows:

SP PC AF HL DE BC I IX IY Flags

'Flags' is a more convenient way of showing the contents of the flag register, F. Those flags which are set are displayed. Note that PC indicates the address of the next instruction to be executed.

The following program displays the register names on the top line of the screen. It may be executed at any time; its starting address is 0F88. The highest address used is 0FCA, leaving sufficient space for the stack for virtually all programs. However, should it be necessary to relocate the program, bytes 0F8F and 0F90 must be changed to the new address of MSG, the start of the message.

CAPTIONS PROGRAM

LOC'N	CONTENTS	LABEL	OP CODE	ARGUMENTS	COMMENTS
0F88	C5		PUSH	BC	Save BC
0F89	D5		PUSH	DE	& DE
0F8A	E5		PUSH	HL	& HL.
0F8B	11 CA 0B		LD	DE,0BCA	Start of destn.
0F8E	21 9B 0F		LD	HL,MSG	Start of src.
0F91	01 30 00		LD	BC,0030	Length of msg.
0F94	ED B0		LDIR		Block transfer.
0F96	E1		POP	HL	Restore HL,
0F97	D1		POP	DE	& DE,
0F98	C1		POP	BC	& BC.
0F99	DF 5B		.MRET		Return to NAS-SYS.
0F9B	53 50 20 20	MSG:		S P (sp) (sp)	
0F9F	20 50 43 20			(sp) P C (sp)	
0FA3	20 20 41 20			(sp) (sp) A (sp)	
0FA7	46 20 20 48			F (sp) (sp) H	
0FAB	20 4C 20 20			(sp) L (sp) (sp)	
0FAF	44 20 45 20			D (sp) E (sp)	
0FB3	20 42 20 43			(sp) B (sp) C	
0FB7	20 20 49 20			(sp) (sp) I (sp)	
0FBB	20 20 49 58			(sp) (sp) I X	
0FBF	20 20 20 49			(sp) (sp) (sp) I	
0FC3	59 20 20 46			Y (sp) (sp) F	
0FC7	6C 61 67 73			l a g s	

Appendix c
Simple I/O Attachments

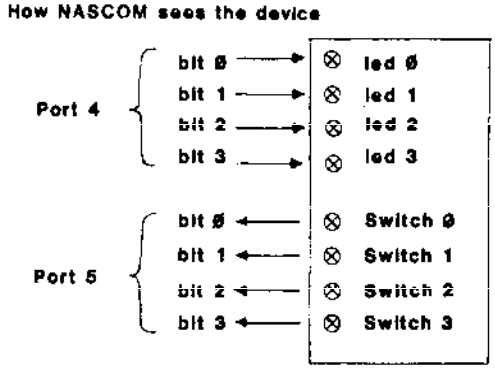
c.1 Connecting Devices to the PIO

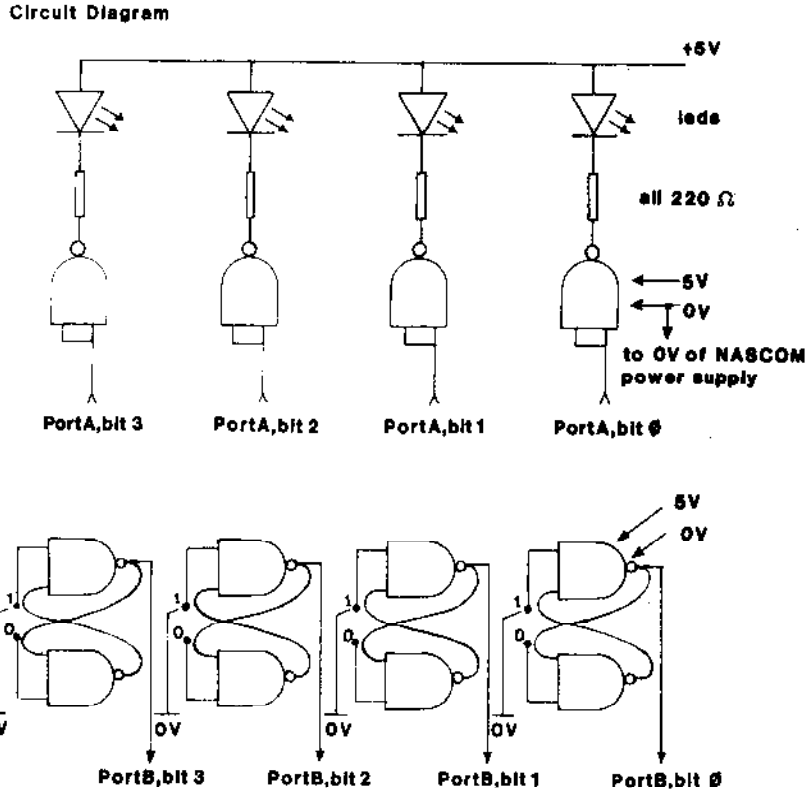
Connections to port A are taken via SKA on the NASCOM 1 board, while those to port B are taken via SKB. On NASCOM 2 both ports are connected via PL4. The connections are given below for reference. All the pins are fully TTL compatible, ie. they may be regarded as being driven from 74XX series logic inside the PIO.

		NASCOM 1	NASCOM 2
	A7	SKA pin 8	PL4 pin 24
	A6	SKA pin 7	PL4 pin 25
	A5	SKA pin 6	PL4 pin 23
	A4	SKA pin 5	PL4 pin 21
	A3	SKA pin 4	PL4 pin 19
PORT 4	A2	SKA pin 3	PL4 pin 17
	A1	SKA pin 2	PL4 pin 15
	A0	SKA pin 1	PL4 pin 13
	ASTB	SKA pin 11	PL4 pin 11
	ARDY	SKA pin 10	PL4 pin 7
	B7	SKB pin 8	PL4 pin 5
	B6	SKB pin 7	PL4 pin 3
	B5	SKB pin 6	PL4 pin 1
	B4	SKB pin 5	PL4 pin 2
	B3	SKB pin 4	PL4 pin 4
PORT 5	B2	SKB pin 3	PL4 pin 6
	B1	SKB pin 2	PL4 pin 8
	B0	SKB pin 1	PL4 pin 10
	BSTB	SKB pin 11	PL4 pin 9
	BRDY	SKB pin 10	PL4 pin 12
	+5V	SKA pin 16,SKB pin 16	PL4 pins 20,22
	GND	SKA pin 9,SKB pin 9	PL4 pins 16,18

c.2 Leds & Switches Attachment

This simply allows bits 0 to 3 of port 4 to drive leds when the port is programmed to be an output port, and bits 0 to 3 of port 5 to be driven by switches when programmed to be an input port.





If, for example, port 4 bit 3 is set to 1 by an OUT instruction, the output of the gate connected to this bit will be logic 0 since the 7400 gate is wired as an inverter. About 12mA will then flow through the light-emitting diode (led) causing it to emit light. If the port output is 0, the gate output will be logic 1 and virtually no current will flow through the led.

The switches are 'debounced' by the two cross-connected gates which together form a simple latch. Although the contacts of the switches will bounce on and off for several milliseconds after being switched, the latch outputs will be a clean transition from one logic level to the other.

The leds may be any of the inexpensive red ones which are readily available. All the gates are TTL type 7400, and the switches are any single-pole, two-way type.

To check the operation of the circuit, one of the outputs from the latches should be connected in turn to each of the gates driving the leds. The switch should be able to turn the leds on and off. This should be repeated using the output of each latch in turn.

Parts Required:

4 off red leds
3 off 7400 TTL Quad 2-input NAND
4 off single-pole, two way switches
4 off 220 ohm resistors, any power rating.

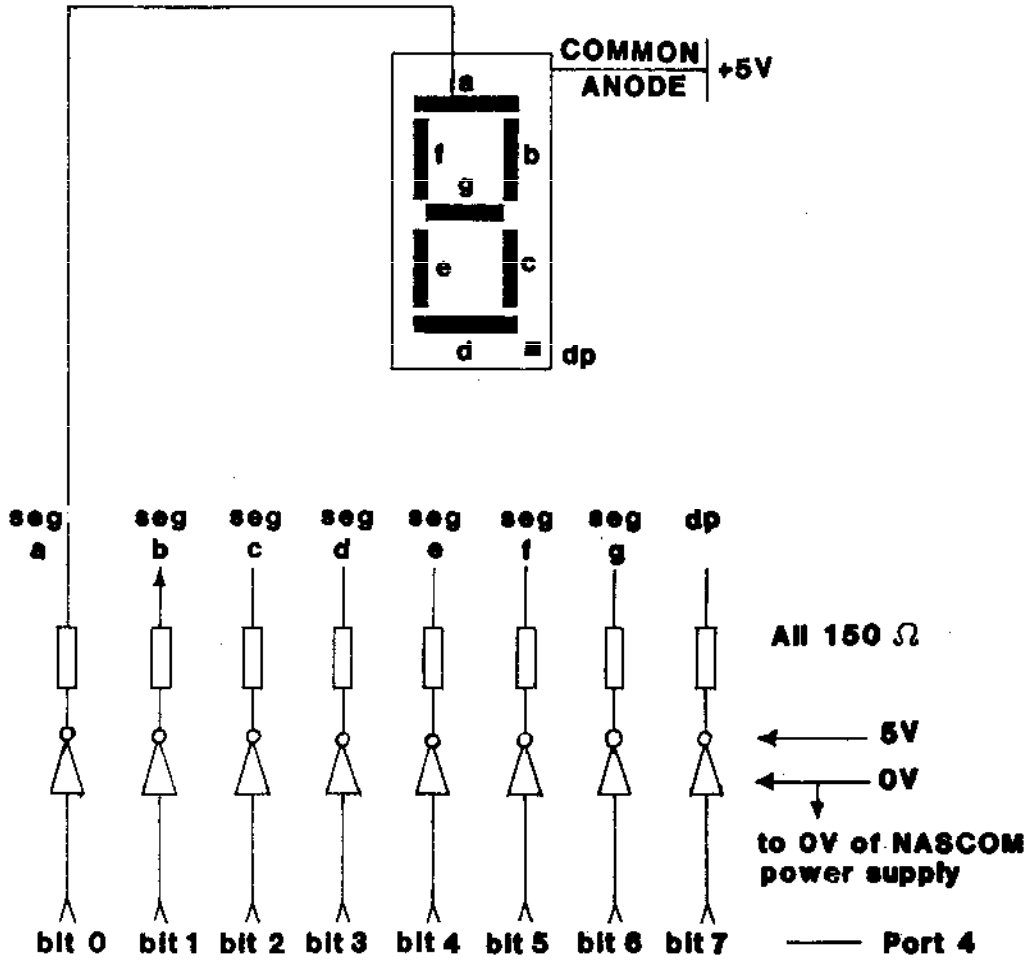
c.3 Seven-segment Display

This circuit, shown on page c-4, is essentially the same as the leds part of the Leds & Switches circuit. The seven-segment display should be of the common anode type, ie. all the anodes of the eight leds are connected together and taken directly to the +5V supply.

With the circuit not connected to the NASCOM, all the seven segments and the decimal point should light when power is applied. When any input is connected to 0V the corresponding segment should extinguish.

Parts Required:

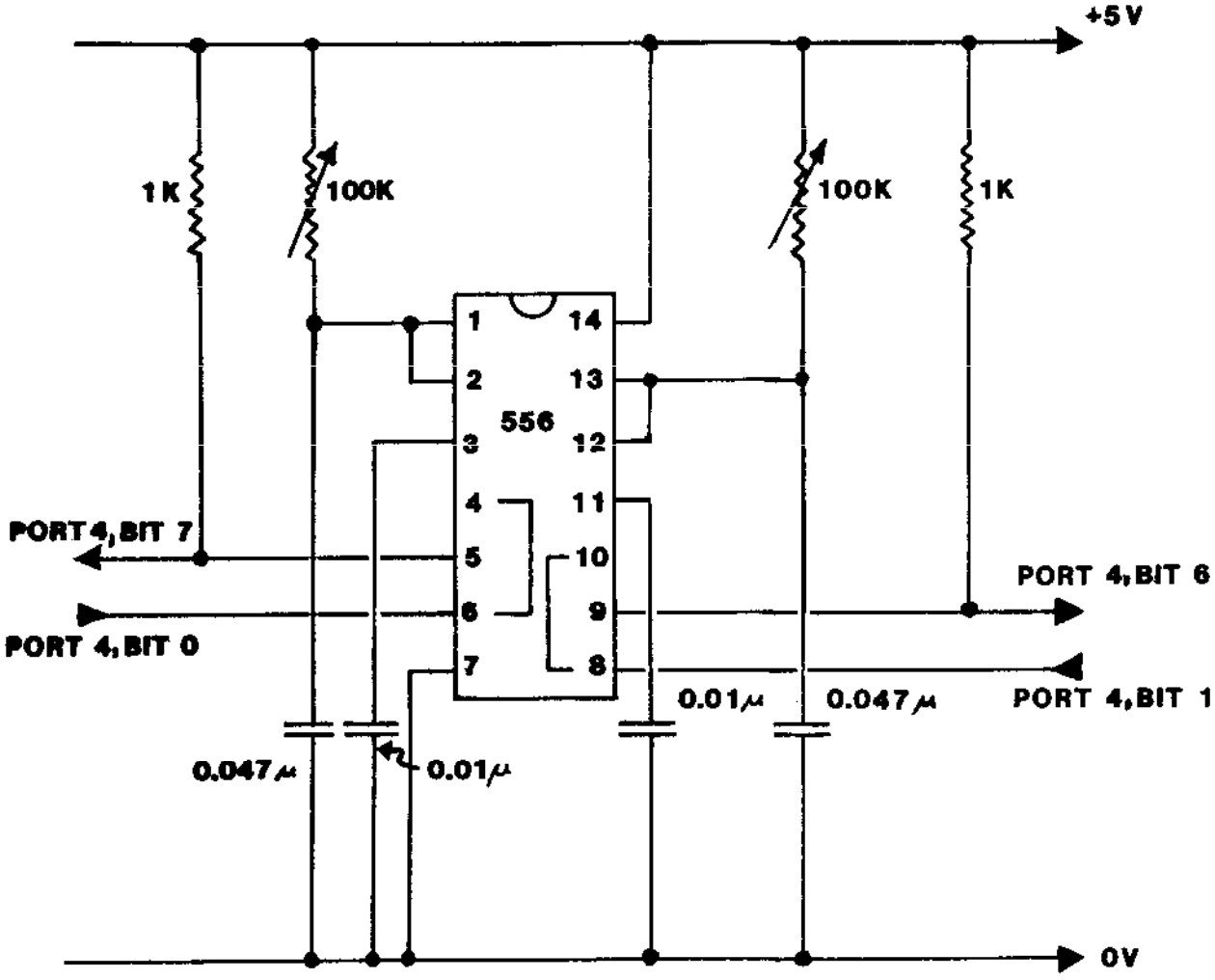
1 off seven-segment display,
common anode type.
2 off 7404 (TTL Hex Inverter)
8 off 15ohm resistors, any power rating.



Circuit Diagram of Seven-segment Display

c.4 Dual Position Encoder

The 556 device used in this circuit contains the equivalent of two of the popular 555 integrated circuit timers. Both halves of the device are connected as a monostable. Thus, when a narrow trigger pulse is applied to the input, an output pulse of width proportional to the value of the 100k variable resistor is produced. The trigger pulse is generated by the driving program and then a counter is incremented until the end of the monostable pulse is detected.



Circuit Diagram of the Dual Position Encoder

Parts Required:

1 off 556 i.c. timer
2 off 100k linear variable resistor/potentiometer
2 off 0.047 microfarad capacitor
2 off 0.01 microfarad capacitor

These values are suitable for the NASCOM 1. For NASCOM 2, either both the 100k variable resistors should be changed to 50k, or both the 0.047 capacitors should be changed to 0.022.

c.5 General Purpose Analogue Input/Output

This circuit is shown on page c-7. The output of the digital to analogue converter, ZN425E, is buffered by the 741 operational amplifier wired as a non-inverting amplifier. The range of the output voltage can be adjusted by the preset variable resistor; with 00H from Port 5 the output voltage will be 0v, and with FFH from Port 5 the output voltage should be adjusted to be about 4v.

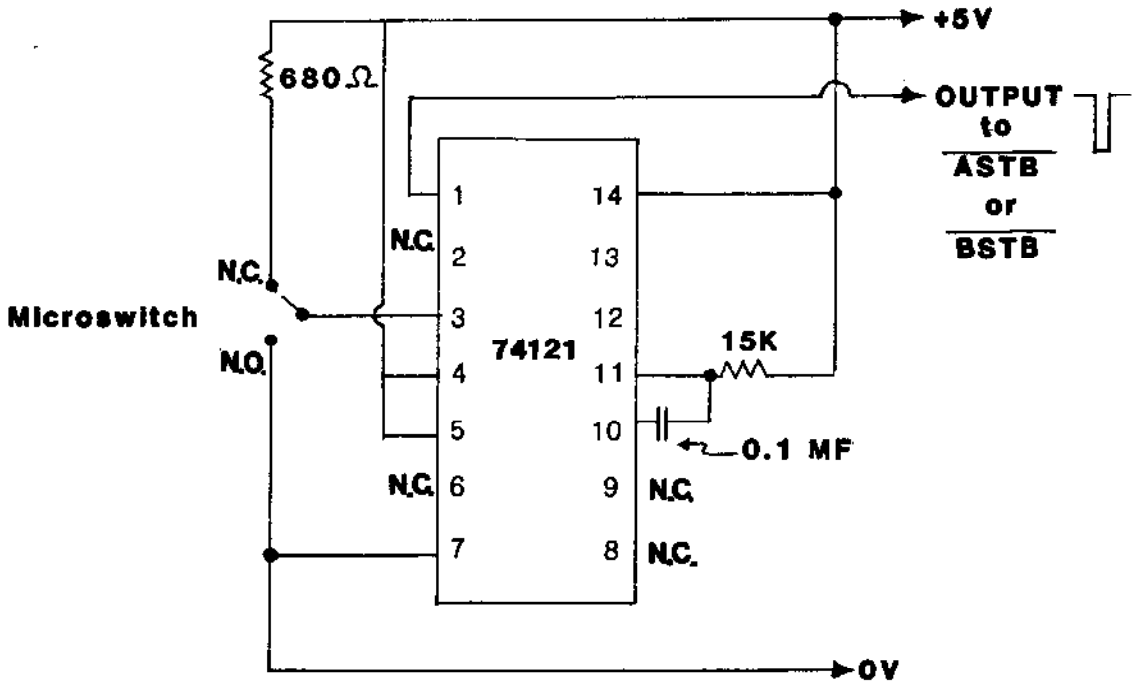
For use as as analogue to digital converter, the analogue output from the DAC is compared with the input voltage by the integrated circuit comparator, LM311.

Parts required:

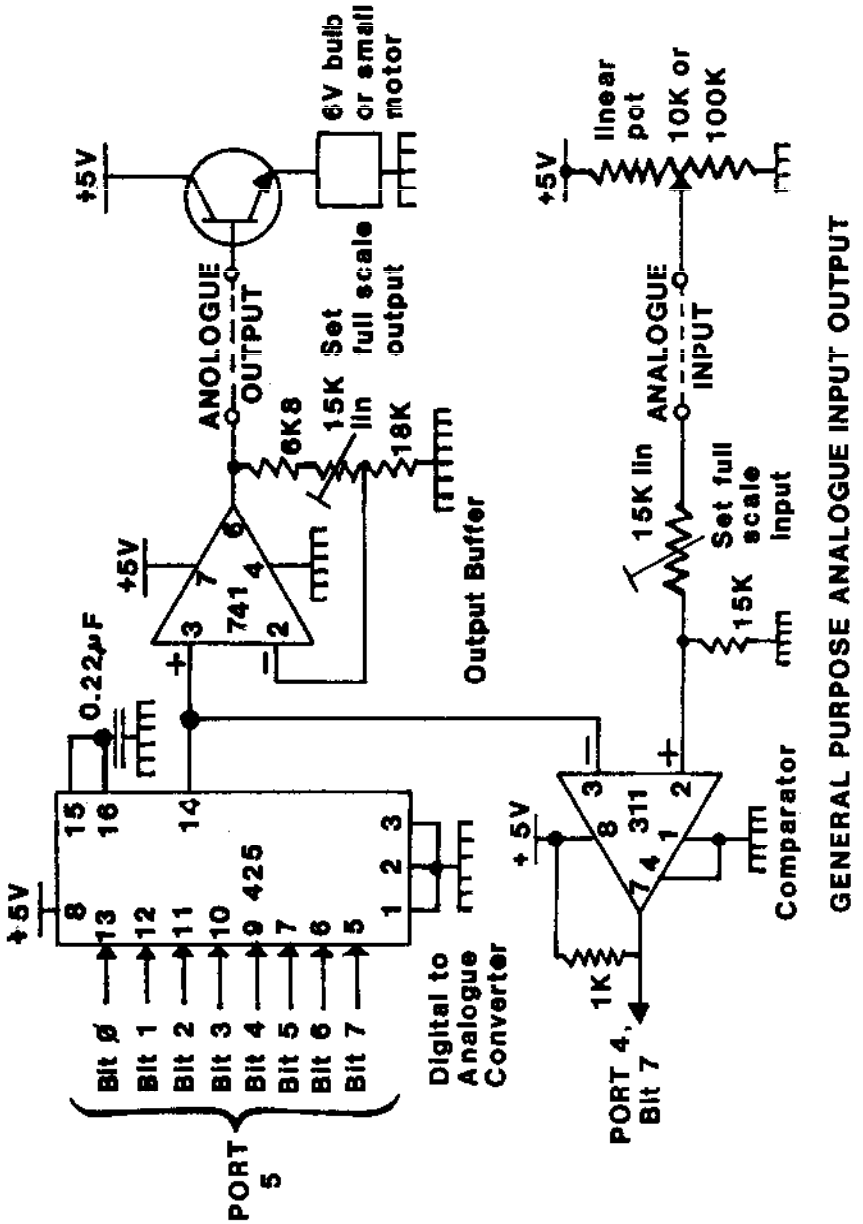
1 off Ferranti ZN425E DAC
1 off 741 operational amplifier
1 off National Semiconductor LM311 comparator
Resistors and capacitors as shown on the diagram

c-6 Pulser Circuit

This circuit, used for generating narrow interrupt pulses is simply a monostable.



Circuit diagram of pulser


GENERAL PURPOSE ANALOGUE INPUT OUTPUT
+5V
6V bulb or small motor
ANOLOGUE OUTPUT
6K8
15K lin
Set full scale output
18K
Output Buffer
741
0.22μF
Digital to Analogue Converter
Bit ø
Bit 1
Bit 2
Bit 3
Bit 4
Bit 5
Bit 6
Bit 7
PORT 5
425
linear pot
10K or 100K
ANALOGUE INPUT
15K lin
Set full scale input
15K
311
Comparator
1K
PORT 4, Bit 7

MACHINE CODE PROGRAMMING
for the NASCOM 1 & 2

Appendix d
Entering and Executing Programs

Immediately after pressing the RESET button the NASCOM begins to execute the monitor program called NAS-SYS. Of the many facilities provided by this program, the most important are those which allow easy entry, modification, and execution of the user program, ie. the program which the user wishes to run. The commands for these are given below.

Modify Memory command,M
eg. MD5C (nl) where (nl) is the NEWLINE key.
This is the command to display the content of memory location 0D5C.
The NASCOM response is 0D5C xx where xx is the current content of memory location 0D5C.

To change the content, simply overtype the displayed content, then press NEWLINE.
The NASCOM response is 0D5D yy that is,the content of the next location is automatically displayed.
To terminate the M command, key .(nl) .

Note 1: Several bytes may be entered in one line so long as they are separated by spaces.
eg. MD57(nl)
0D50 3E 0 C6 1 C6 2 C6 3 (nl)
0D08 xx
Note that leading zeroes need not be entered in this case.

Note 2: The memory address may be automatically decremented by keying :(nl)
eg. 0D57 xx now key :(nl)
response is 0D56 yy

Note 3: The modify command may be continued from yyyy by keying /yyyy(nl).

Execute command,E
eg. ED50 (nl)
causes the NASCOM to execute the program beginning at location 0D50.

Single-step command,S
eg. SD56 (nl)
causes the instruction at memory location 0D56 to be executed and the contents of the registers are displayed in the form
SP PC AF HL DE BC I IX IY Flags
To execute the next instruction of the program, simply press NEWLINE.

NAS - SYS to NASBUG
conversion

NAS-SYS supercedes earlier monitors such as NASBUG. In most cases, users of NASBUG may replace the NAS-SYS routines used in this book with the corresponding NASBUG routines listed below. Some care is required however since the routines are not identical; also, as the NASBUG routines are one or two bytes longer than than those in NAS-SYS, memory references will have to be changed too. This affects primarily the absolute jump instructions, JP xxxx, in the programs.

NAS-SYS routine		FUNCTION	Similar NASBUG routine
DF 68	.B2HEX	Displays contents of register A in hex.	CD 44 02 CALL B2HEX
DF 7B	.BLINK	Get input character into A. Modifies HL and DE.	CD 3E 00 CALL CHIN
DF 6A	.CRLF	Displays CR/LF. Sets A to 0DH.	CD 40 02 CALL CRLF
DF 6B	.ERRM	Display "Error"/CR	EF 45 72 72 72 6F 72 1F 00
DF 5B	.MRET	Returns control to NAS-SYS.	31 33 0C LD SP,0C33 C3 86 02 JP PARSE
EF	.PRS	Display string which follows. String is terminated by 00H.	EF CALL PRS
FF	.RDEL	Delay a maximum of 5.4ms (2.7ms on N2) proportional to A. Sets A to 00H.	CD 35 00 CALL KDEL (7ms)
F7	.ROUT	Displays character , code in A.	CD 4A 0C CALL $CRT
DF 69	.SPACE	Display space. Sets register A to 20H.	CD 3C 02 CALL SPACE
DF 5D	.TDEL	Delay 2.7s (1.35s on NASCOM 2) Sets A and B to zero.	