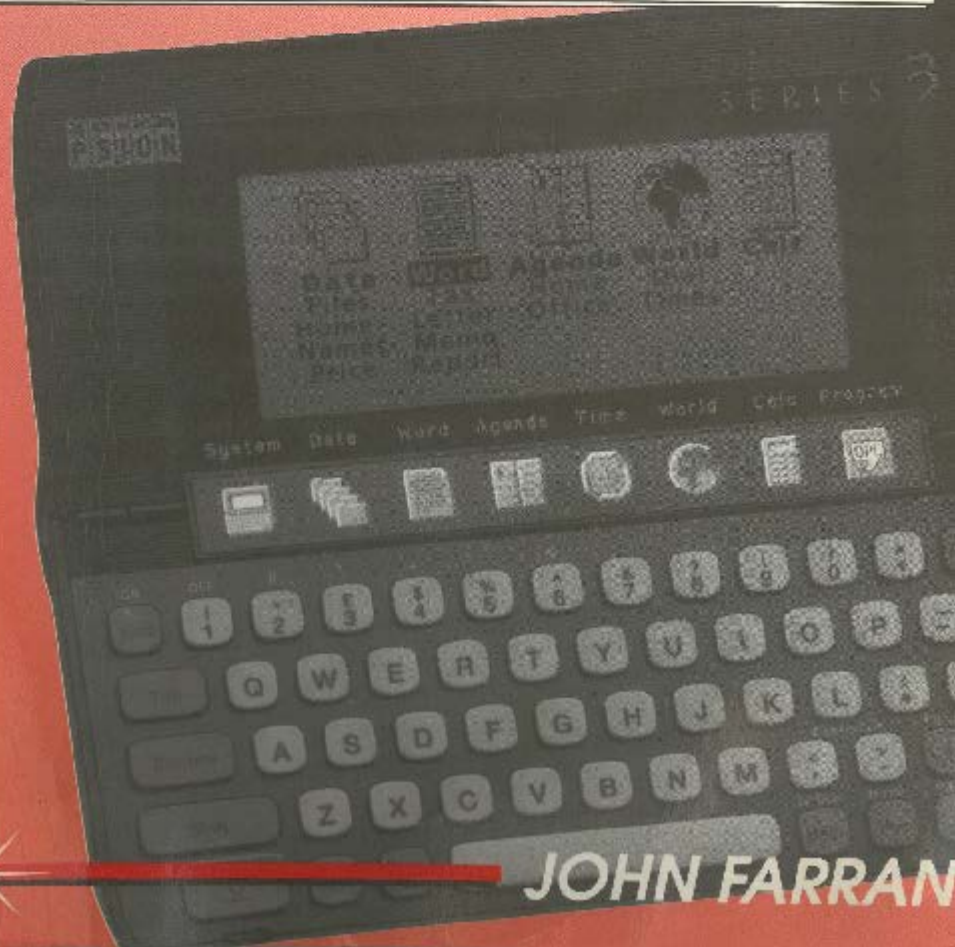


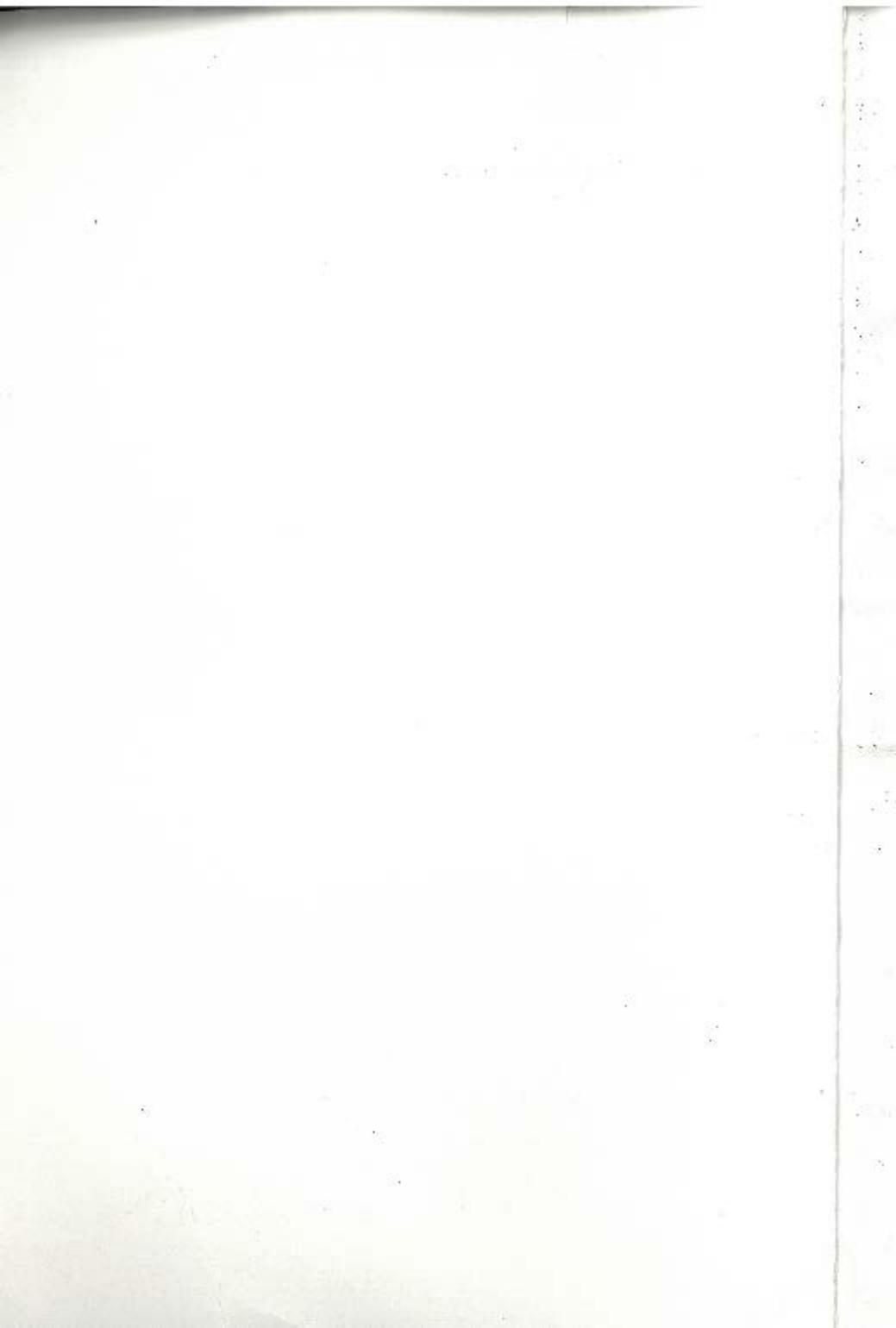
MACHINE CODE PROGRAMMING ON THE

PSION

SERIES 3A & 3



JOHN FARRAN



Machine Code Programming on the Psion Series 3A & 3

by

John Farrant



Machine Code Programming on the Psion Series 3A & 3

©1994 John Farrant

This book and the programs within are supplied in the belief that the contents are correct and that they operate as specified, but the authors and Kuma Books Ltd shall not be liable in any circumstances whatsoever for any direct or indirect loss or damage to property incurred or suffered by the customer or any other person as a result of any fault or defect in the information contained herein.

The moral right of the author has been asserted

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, scanning, recording or otherwise without the prior written permission of the author and the publisher.

ISBN 07457 0299 6

Published by: Kuma Books Ltd
12 Horseshoe Park, Pangbourne
Berks, RG8 7JW

Tel 01734 844335
Fax 01734 844339

Acknowledgement of Registered Trade Marks

While every effort has been made to qualify and accredit the information therein, the authors accept no liability or responsibility for any errors or inaccuracies contained. All copyright and trademarks are acknowledged.

Contents

Chapter 1	Numbers and Systems	1
Chapter 2	The 8086 Structure	16
Chapter 3	The Assembler Environment	28
Chapter 4	First steps in Programming	39
Chapter 5	Further Menu Options	50
Chapter 6	8086 Arithmetic Functions	57
Chapter 7	Subroutines and the Stack	71
Chapter 8	The Psion 3 Operating System	88
Appendix 1	V30 Microprocessor	103
Appendix 2	8086 Instruction Set	111
Appendix 3	Two-pass Assembler/Disassembler Listing	152
Appendix 4	Source Editor Listing	191
Appendix 5	8086 Mnemonic File	194
Appendix 6	System Error Messages	203
Appendix 7	Interrupts and Function Calls	207



Chapter 1 - Numbers and Systems

INTRODUCTION

The arrival of the Series 3 from Psion in 1991, marked a new chapter in the development of pocket sized computers. Based on an extremely complex but reliable architecture, this miniature marvel can be programmed using one of three types of computer language; OPL - a high-level and easy to use language, similar in many ways to BASIC but capable of greater sophistication, "C" using the available Software Development Kit (SDK) or standard 8086 machine code via a suitable assembler.

This book is aimed at anyone who wishes to progress on from OPL and increase the power and flexibility of their programs and applications without resorting to the problems associated with using "C". As you work your way through this book you will develop an understanding of 8086 programming techniques using a simple but effective two-pass assembler. The small sections of code you produce will allow you to communicate with the Psion's hardware and system resources at a level impossible or difficult using OPL. Along the way, you will gain a better understanding of the Psion 3's hardware design and operation.

THE PSION 3

The Psion 3 series of computers are built around three main integrated circuits. The NEC V30 series 16-bit microprocessor, which is an enhanced device based on the 8086 type architecture as used in desk top PC's, and two Psion customised chips know as ASIC1 and ASIC2. The 3a series combines these three chips into one larger integrated circuit. It is refreshing to note that the V30 is some 40% faster than the 8086 for the same clock speed and in addition has a larger instruction set. The increase in speed of the V30 over the 8086 means that the 3a running at 7.68 Mhz is faster than a conventional PC running at 10 Mhz.

A full description of the V30 microprocessor unit can be found in appendix 1, but for now we'll look at the problems to be overcome in writing, entering and

running programmes on a machine that appears to defy all attempts to break into the system.

The Psion 3 is a very well behaved computer. What I mean by this is that it is virtually impossible to cause it to lock up. You may already have experimented with machine code by using the PEEK and POKE instructions from OPL. Haphazard approaches to peeking and poking will almost certainly result in your Psion crashing and system error messages of the type shown in Figure 1.

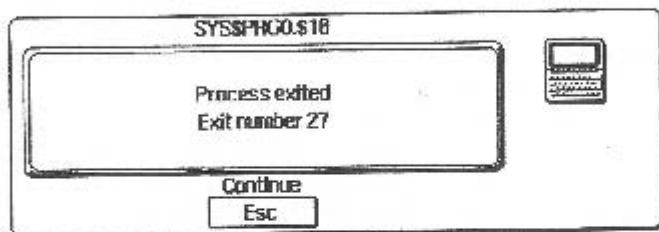


Figure 1

This happily provides some sort of safety net. It is important to remember that we are about to embark on a voyage filled with obstacles and pitfalls. At best, mistakes will produce simple error messages similar to that above. At worst your machine will lock up completely and you will have to perform a reset and suffer the possible consequences of losing all your important program and data files, etc.

Before continuing, I strongly advise you to save all important information to a Flash SSD or a floppy disk, if you possess the serial link.

MACHINE CODE CAN SERIOUSLY CORRUPT YOUR DATA.

In spite of this however, if you enter the examples carefully, I promise that I will ensure that your progress is relatively safe and trouble free.

THE SAFETY BARRIER

We'll begin with a simple program that causes the computer to safely crash.

Enter and run the following short program in OPL to see what I mean:

```
PROC A:
Local u%
Pokeb 4048,0
u%=USR(4048,0,0,0,0)
Endp
```

What we have done here is to poke an illegal 8086 instruction at memory address 4048 and then executed the single byte code. This however, is not the only problem with this short OPL program. We have failed to terminate the routine with a RETURN instruction. This must always be present at the end of any machine code program to allow a safe return back to the operating system. The decimal code for the RETURN instruction is 203. Enter the following and note the difference:

```
PROC A:
Local u%
Pokeb 4048,203
u%=USR(4048,0,0,0,0)
Endp
```

A safe return resulted!

The machine code program was only one instruction long, but at least it ran without crashing.

THE USR INSTRUCTION

The components of the USR instruction are worth examining:

```
u%=USR(4048,0,0,0)
```

The first number, in this case 4048, is the address that is loaded into the 8086 program counter register. The microprocessor always starts executing code at the address held in this register. The remaining four zeros are passed to the four main registers of the 8086 in the following order: AX, BX, CX and DX. In the example above we have simply set all the registers to zero.

We could of course have sent any number we like to these four registers. For example:

```
PROC A:
Local u%
Pokeb 4048,203
u%=USR(4048,255,0,0,0)
Print u%
Get
Endp
```

Here we have loaded the AX register with the number 255. Run this program and note that 255 is printed on the screen. The reason for this is that at the end of any machine code routine run from OPL the value of the AX register is passed back to the USR integer variable, u%.

Can we store a number larger than 255 in the AX register?

Yes, we can:

```
PROC A:
Local u&
Pokeb 4048,203
u&=USR(4048,32767,0,0,0)
```

```
Print u&
Get
Endp
```

Note that the integer variable `u%` has been replaced in this example by the long integer variable `u&`. Unlike integer variables which can only hold numbers between -32,768 to +32767, long integer variables can handle numbers in the range -2147483648 to +2147483647

THE ACCUMULATOR REGISTER

The Accumulator, or AX register as it is called, actually consists of two 8 bit registers known as AH and AL, Accumulator High and Accumulator Low. The two registers can be used together or separately depending on the size of the number being handled.

NEGATIVE THOUGHTS

The largest decimal number that an eight bit register such as AH or AL can store is 255. If you include zero, this means 256 different numbers, 0 to 255. As 255 times 255 equals 65535 you would imagine that 65535 is the largest number that can be contained in the 16 bit AX register. Try the following:

```
PROC A;
Local u&
Pokeb 4048,203
u&=USR(4048,65535,0,0,0)
Print u&
Get
Endp
```

This produces the unexpected -1 as a result. Change the 65535 to 65534 and run the program again.

The reasons for these odd results are that the microprocessor needs to be able to handle negative numbers as well as positive numbers. The only way it can

Chapter 1 - Numbers and Systems

do this is to limit the maximum size of decimal numbers it can cope with. The limit is -32768 to +32767.

65535 = minus 1
65534 = minus 2

.....
32769 = minus 32767
32768 = minus 32768
32767 = plus 32767
32766 = plus 32766

You'll learn to understand the reasoning behind this rather strange numbering system when we attempt addition and subtraction in machine code.

It might be as well at this stage to remind ourselves about what is meant by decimal, binary and hexadecimal.

NUMBER BASES

Information inside a computer is stored as bits. These are either one's or zero's. It would be nice if the microprocessor could handle decimal numbers as easily as we can. In other words, one, two, three, four, etc. This is not possible as our microprocessor contains thousand of electronic switches that can only be ON or OFF. These two switched states are called logic 1 and logic 0.

Every decimal number that we enter into our Psion computer has to be converted internally into a series of one's and zero's called bits or binary numbers. These bits are stored in RAM or in the processors internal registers. A single bit register would only be able to store a 1 or a 0. This would of course rather limit the mathematical capabilities of the machine and larger registers are therefore used.

A typical eight bit register can store numbers from 00000000 to 11111111 binary. Converted to decimal, this is 0 to 255.

In the binary system, the right-most bit represents two to the power of nought, which is 1. The next bit to the left represents 2 to the power 1, or decimal 2; the next, 2 to the power 2 = 4, and so on upto the left-most bit, 2 to the power 7 = 128. Adding all these powers gives us 255. This is the largest number that an eight bit register can hold:

$$128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$$

In other words, decimal 255 converted to binary is 11111111.

Any decimal number can easily be converted to binary by simply adding up the decimal numbers in the series wherever a binary 1 occurs.

For example, 0100 0001 binary is:

0 1 0 0 0 0 0 1

64 plus 1 which equals 65

Further examples:

Decimal	Binary
1	0000 0001
9	0000 1001
7	0000 0111
67	0100 0011
161	1010 0001

Notice how eight bit binary numbers are easier to read when they are split up into two groups of fours. These groups are called "nibbles". Therefore two nibbles make a byte. Also two bytes equal a word and four bytes, or two words, make a long word.

As an exercise convert the following binary numbers to decimal:

a) 0000 1111 b) 0001 0001 c) 0011 0011 d) 1111 0000 e) 1111 1111

BITS AND BYTES

As we have seen, eight bits arranged together are called a byte. A one mega byte memory chip can therefore hold one million eight bit numbers. The microprocessor in the Psion 3 can handle sixteen bit numbers, ranging from 0000 0000 0000 0000 to 1111 1111 1111 1111, or 1 to 65535 in decimal. These larger numbers are obtained in the same way as the eight bit binary number above, but now we have eight extra bits to play with. This means more powers of two with the left-most bit now representing 2^{15} to the power of 15 which is 32768. If you care to add this lot up you should arrive at 65535, which of course is the largest number which a 16 bit processor can handle.

HEXADECIMAL

If writing machine code involved entering large amounts of binary numbers by hand, we would have long ago gone mad and abandoned computers altogether. Luckily there are easier ways of programming computers.

Firstly, any number of binary bits can be grouped together into fours and turned into what is known as a hexadecimal number. It is really very easy, and the following table should explain:

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6

7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Table 1

Referring to table 1, binary 1010 is A in hexadecimal, 1110 is Hex E and so on.

Eight bit binary is represented by simply placing two hex numbers together. Firstly split the eight bit number into two four bit binary numbers and then look them up in the table above:

1010 1110 = AE hex
0001 0101 = 15 hex

In other words, all we have to do to convert an eight bit binary number to its hex equivalent is to split it up into two four bit parts and sort out each of the two nibbles in turn.

The same process applies to 16, 32 or 64 bit numbers:

1010 0001 1111 1100 = A1FC in hexadecimal

DECIMAL TO HEXADECIMAL

Converting a decimal number to its hex equivalent involves nothing more than firstly converting the number to binary using the method on page 7, splitting the resulting bits into groups of four and finally changing them into hex:

Chapter 1 - Numbers and Systems

Decimal $170 = 128 + 0 + 32 + 0 + 8 + 0 + 2 + 0$

Therefore decimal 170 equals 1010 1010 which in turn equals AAh (note the h for hex).

HEXADECIMAL TO DECIMAL

To convert a hex number back to its decimal equivalent, do the following:

Taking each hex component in turn, look it up in table 1.

A4 Hex: A = 1010 in binary and 4 = 0100 in binary

Therefore, A4h equals 1010 0100 in binary.

Substituting 128, 64, 32, 16, 8, 4, 2 or 1 where ever a binary 1 occurs and adding gives us:

1	0	1	0	0	1	0	0
128		32			4		

$$128 + 32 + 4 = 164$$

Therefore, A4 hex equals 164 decimal.

Prove the following:

a) B6h = 182 decimal b) CCh = 204 decimal c) 4Bh = 75 decimal.

Now convert the following hex numbers to decimal:

d) 6Eh e) 8Ah f) 72h.

Confirm your answers using the calculator function on your Psion. Just enter \$6E and press the enter key.

MACHINE CODE INSTRUCTIONS AND DATA

The 8086 has a wealth of instructions that it can understand. Examples of these are ADD, SUBTRACT, MULTIPLY, DIVIDE, JUMP to an address, and many more. Our microprocessor unfortunately cannot understand these instructions as they stand. They need to be changed into hex numbers before they can be executed. So our first problem is how do we convert these instructions into hex numbers which the 8086 can interpret?

With eight bit microprocessors, this task is relatively easy. You simply look up the relevant mnemonic, ADD, MUL, DIV etc, in a list of instruction codes for that particular microprocessor and poke the appropriate hex number into memory using a program called a hex loader. It is a laborious process, not for the faint hearted and is prone to errors. This method of program entry is not possible on 16 bit microprocessors like the 8086, which has many hundreds more instructions than a humble eight bit machine. Also it would be difficult to produce a useful table containing all the combinations of addressing modes and registers that are available.

What we need is somehow a method of converting the instruction mnemonics into their hex equivalent without having to look them up in a table of codes.

An assembler performs such a task. In its simplest form you would enter something like LOAD AX with 0051, LOAD BX with 0023, ADD AX to BX, RETURN. The assembler will then convert these four instructions into the appropriate hex codes. In this case, B85100 BB2300 01D8 CB.

Using conventional 8086 mnemonics we would write this as follows:

MOV AX,0051	B85100
MOV BX,0023	BB2300
ADD AX,BX	01D8
RETF	CB

Running this program will produce 74 hex as the correct answer.

Three things need to be noted about the above program. Firstly, when loading or moving data to and from registers, the two bytes that make up the complete word are entered the correct way round in the mnemonic form, but finish up in memory with the two code bytes swapped around. This is normal with most processors. It doesn't affect the way we write the mnemonics or source code as it is called, but it must be borne in mind when examining machine code using a debugger.

The second point to note is the ADD AX,BX instruction. Here, the contents of BX are added to the AX register and the result is stored in the AX register. The original contents of AX are lost but BX retains 23.

Finally, the RETURN instruction is entered as RETF. The F means that the return is what is known as a FAR RETURN. The 8086 allows two types of RETURNS: Near Returns and Far Returns. The reason why two types of return instructions are required is due to the internal architecture of 8086 type processors.

SEGMENTATION

Intel, in their wisdom, decided to split the addressing range of the 8086 family of processor into a number of blocks called segments. Each segment can be a maximum of 64 Kbytes long. Each segment has a segment number and this is stored in a segment register. Broadly speaking, if you want to address the first 64 k block you would store 0 in the segment register and set the program counter to the required address within the 64 k block. If however you wanted to access memory at address 99,234 for example, then the segment register would be loaded with 1 and the program counter with $99,234 \text{ minus } 65536 = 33,698$. It is somewhat more complicated than this as we will see shortly, but the basic principles apply.

The Segment Registers

The 8086 has four segment registers:-

DS - data segment, CS - code segment, SS - stack segment and ES - extra segment.

The segment names give us a clue as to their particular function. The CODE segment is where our program code will be stored, the STACK segment holds data that needs to be saved on the memory stack, and the DATA segment is where the processor stores and retrieves the data it needs. For example, `MOV AX,[23A1]` is the instruction for moving the contents of memory location 23A1 to the AX register. Here the data is collected not directly from memory location 23A1, but from this location PLUS the contents of the DATA segment (DS). The contents of the segment register are added to 23A1 to form the final address where the microprocessor will find the data.

In reality, what happens is that the contents of the appropriate segment register are multiplied by 16 before being added to the memory address part of the instruction. This in effect forms a complete 20 bit address allowing memory addressing up to the full 1,048,576 range. (2 raised to the power of 20 equals 1,048,576):

20-bit address = $(16 \times \text{DS}) + \text{address part of instruction code}$

For example:

DS register = 1000h

Address part of instruction code = 0234h

Then:

20-bit address = $(16 \times 1000\text{h}) + 0234\text{h}$

Equals 10000h (multiplied by 16, or shifted left 4 bits) + 0234h

Chapter 1 - Numbers and Systems

Therefore the final address where data is to be found is:

10234h

The same thing applies when the code segment is involved:

20-bit address = $(16 \times \text{CS}) + \text{contents of Instruction Pointer}$

20-bit address = $(16 \times \text{CS}) + \text{IP}$

For example:

CS register = 2100h

IP = 0234h

Then:

20-bit address = $(16 \times 2100\text{h}) + 0234\text{h}$

Equals 21000h (multiplied by 16, or shifted left 4 bits) + 0234h

The final address where data is to be found is:

21234h

Note in the above example that IP stands for the Instruction Pointer. This performs the same function as the Program Counter mentioned earlier.

RETURNING FROM A SUBROUTINE

To get back to our original problem that was what is the difference between NEAR RETURNS and FAR RETURNS?

If a return occurs within a 64 k code segment, then a NEAR RETURN is required. The mnemonic for this is RET. However, if a return will involve jumping across a segment boundary, in other words the address "gap" is greater than 64 K, then a FAR RETURN (RETF) must be used. Every program you write must end with a FAR RETURN. The distance between our code and Psion's operating system is much greater than 64 k and any attempt to use a near return at the end of your code will almost certainly result in a system crash.

As an exercise enter the following OPL program and execute it.

The decimal code for RET (near return) is 195:

```
PROC A:  
Local u%  
Pokeb 4048,195  
u%=USR(0,0,0,0,0)  
Endp
```

The Psion crashes and responds with an exit number. Replacing the 195 with 203, the code for RETF, will give us a safe return.

A list of exit or panic numbers as Psion calls them, can be found in appendix 6.

Chapter 2 - The 8086 Structure

The Microcomputer

Figure 1 shows a typical microcomputer system.

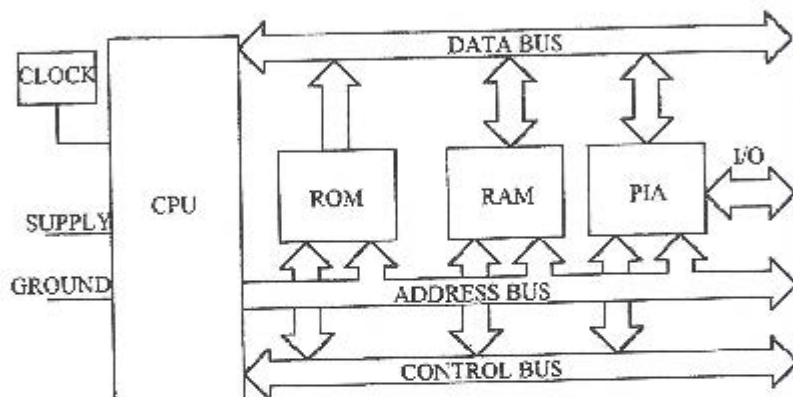


Figure 1

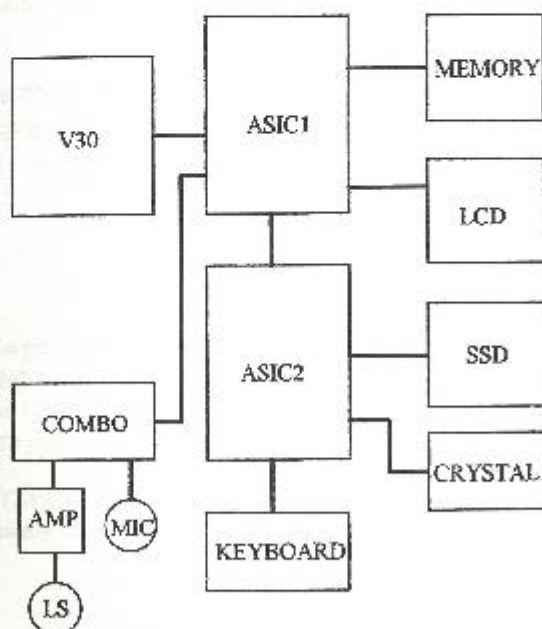
The Psion operating system is contained in 1 megabyte of ROM (Read Only Memory) whilst the user programs are held in RAM (Random Access Memory), which can be to as much as 0.5 megabyte (excluding the RAM slots). Bearing in mind the limitations imposed by a 20 bit address bus, this large amount of internal memory seems amazing. The reason that the Psion appears to be accessing more memory than is possible with the available address lines (20 lines = 1,048,576 bytes) is that extra memory, or shadow memory, is paged in and out only when required. The paging function is handled by the sophisticated memory management system within the Psion hardware.

The PLA, Peripheral Interface Adapter, provides the interface to the outside world. This would consist of peripherals such as a printer, the keyboard and display monitor

The Clock is the heartbeat of the CPU (Central Processing Unit). It determines the memory access speed and also the number of instructions per second that the processor can perform.

The control bus contains extra signals that the processor requires such as read/write and chip enable.

A greatly simplified block diagram of the Psion 3 is shown in below. Although it is far more complex then the typical microcomputer shown above, it behaves in a similar fashion.



ASIC1 is the main system controller chip which interfaces directly with the V30 microprocessor. Two of its main tasks are memory decoding and interrupt handling.

ASIC2 contains the system clock oscillator and is also responsible for the SSD and high speed serial link. Other tasks performed by this complex integrated circuit are keyboard handling and driving the sound system.

The two 1.5 volt batteries supply power to generate up to seven separate internal supplies. These are switched on depending on the particular state that the Psion is in at the time. There are four main states, standby, backup, idle and operating. In the standby state the V30 is powered down but ASIC1, ASIC2 and the memory are powered up. When the main batteries have expired, the backup state comes into play. This state is similar to the standby state except that the it is not possible to switch to the full operating state. The idle state can occur during the operating state and although the processor is powered up, it has entered an idle mode where it is waiting for an event to occur, for example an external interrupt.

In the Psion 3a the functions of V30, ASIC1 and ASIC2 are combined into one larger integrated circuit known as ASIC9. This chip gives an improvement in processing speed and also extends the memory address range by widening the address bus to 24 bits.

THE 8086 MICROPROCESSOR

A typical microprocessor unit looks something like figure 3 below:

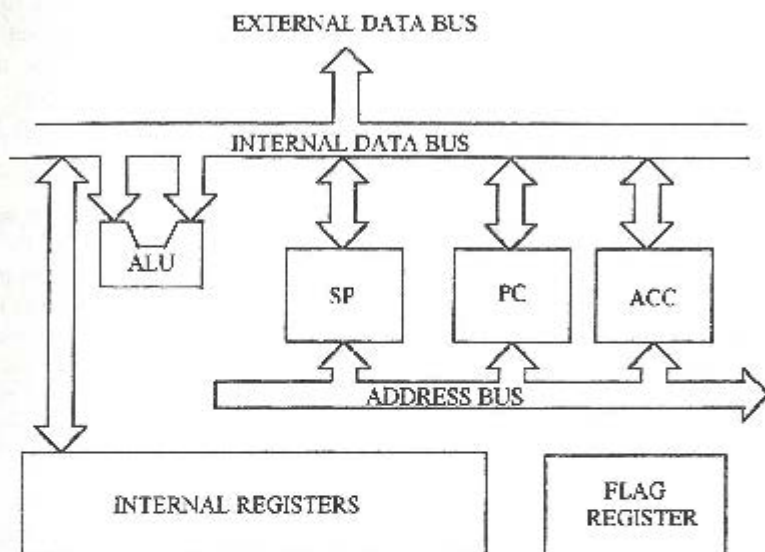


Figure 3

The Arithmetic Logic Unit (ALU) is the brains of our microprocessor. It performs all the arithmetic functions like add, subtract, multiply and divide. It can also shift and rotate bits within a byte left or right by one or more positions. This is an extremely useful function as we will see later.

The Accumulator (ACC) is where data is stored before being acted on by the ALU. It is one of many internal registers that a processor might possess.

The Stack Pointer (SP) contains the address of what is called the stack. This is an area of memory (RAM) where important information like return addresses and rapidly needed data are stored for later retrieval.

The Program Counter (PC) holds the address of the next instruction to be executed. In the 8086 family of microprocessors this register is called the Instruction Pointer (IP). The IP always points to the address of the next instruction to be executed. Although during program execution, the contents of the Instruction Pointer can be modified or saved, we can not access this register directly.

The Flag Register or condition code register as it is sometimes called, is made up of a number of single bit registers which can set or cleared by the programmer or under program control. For example, if an addition results in an overflow, the overflow flag is set or if a subtraction produces a zero result then the zero flag would be set. Note that if a particular flag is set then it contains a 1, otherwise a zero. The contents of a particular flag bit can be tested and the processor can alter its execution path based on the logical values of these codes. The 8086 instruction set presented later clearly indicates which instructions modify which condition flags.

8086 REGISTERS

The 8086 contains all of the above registers mentioned in the previous section plus the segment registers, CS, DS, SS, ES, and two index registers, SI and DI.

The 8086 registers divide themselves into two main groups: the DATA group, and the POINTER and INDEX group. Each of the registers is a full 16 bits wide.

AX, BX, CX and DX are the data group that can be used as a single 16 bit register or as two 8 bit registers. When used as two 8 bit registers, they are divided into an upper (H) and lower (L) half. These are labelled AH, AL, BH, BL, CH, CL, DH, and DL.

The pointer and index registers can only be used as 16 bit registers. We will learn later the reasoning behind these register names.

The four segment registers are each 16 bits wide and are widely used by the Psion operating system. We will not attempt to modify them during our programming exercises later, as unpredictable results could occur. As stated earlier, these registers are combined with other internal registers during program execution to form a complete 20 bit address for the 8086's external address bus. For example, when performing a PUSH or a POP instruction using the stack, the Stack Segment register is combined with other registers to form the memory address that points to the top of the stack.

THE FLAGS

The CONTROL flags consist of DF (direction flag), IF (interrupt enable flag) and TF (trap flag). The six 1 bit STATUS or condition flags are updated after an instruction has been executed. They represent the condition of a result after an arithmetic or logic operation. As we will see, the program execution path can be modified based on the value of these flags which of course can only be either a 1 or a 0.

AF Auxiliary Carry Flag. If this flag is set (=1), there has been a carry from the lower four bits of a byte to the higher four bits. Four bits are called a nibble, so we can say that a carry has occurred from the low nibble to the high nibble of a byte.

CF Carry Flag. This flag is set when there has been a carry or a borrow to the high-order-bit of the result. This result can be either 8 or 16 bits. The carry flag is set when, for example, 128 and 129 are added together. The result (257) is larger than 255 and a carry over occurs.

OF Overflow Flag. When this flag is set, an arithmetic overflow has occurred. This happens when by adding two large positive numbers, the last bit in the byte is set.

If you remember, in order to handle negative as well as positive numbers, the processor deals with an eight bit number not as being from 0 to 255, but as from -128 to +127. Therefore 127 plus 1 must result in an overflow because positive 128 is an invalid number.

SF Sign Flag. This flag tells us whether the result of an operation is a positive or negative number. A 0 in this flag register indicates a positive number, and a 1 indicates a negative number.

PF Parity Flag. If this flag is set, the result of an operation has an even number of 1's in it. For example, $10101000 + 00000010 = 10101010$ would set the parity flag to 1.

ZF Zero Flag. If the result of an operation is zero, then this flag will be set.

Figure 4 on page 23 shows the 8086 register and flag details.

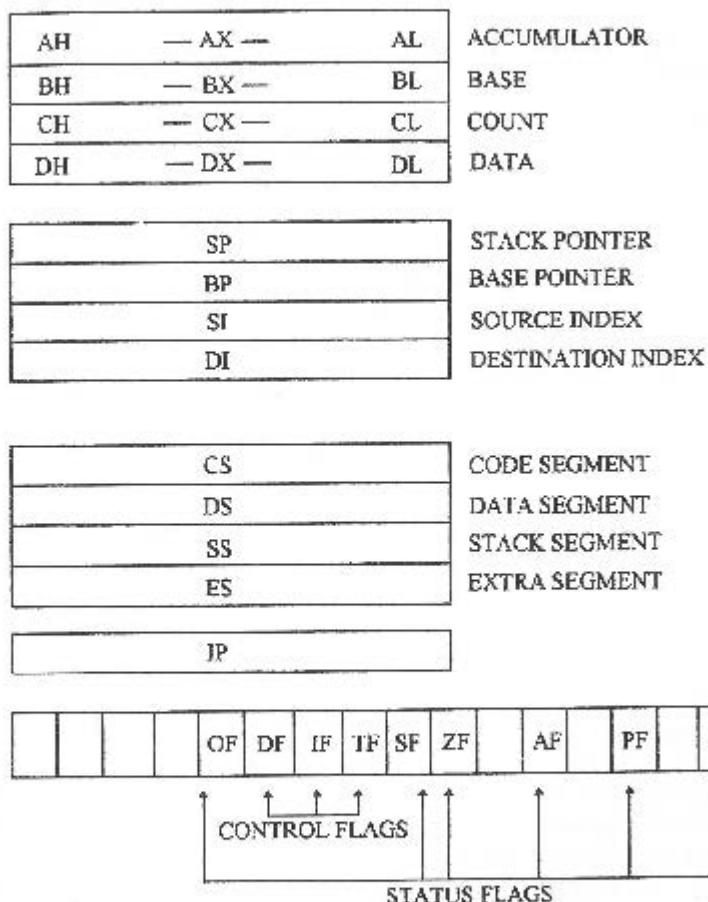


Figure 4

8086 MOV INSTRUCTIONS

THE MOV INSTRUCTION

The MOV instruction transfers a byte or word from the source operand to the destination operand. (Note: an operand is data to be operated on by the CPU).

The instruction is written as:

MOV destination, source.

IMMEDIATE ADDRESSING

In the immediate addressing mode, the operand appears in the instruction. An example of an immediate instruction is one that moves a constant value into an internal register (which, in the following example, is AX):

MOV AX,568

The constant 568 is loaded into the internal register AX. The number 568 will reside in the bytes of memory that comprise the instruction.

REGISTER ADDRESSING

The register addressing mode indicates that the operand to be used is contained in one of the general internal registers of the CPU. In the case of the AX, BX, CX, or DX registers, this register may be 8 or 16 bits. For example, the instruction MOV AX,BX will move the contents of the BX register into the AX register. A similar instruction, MOV AL,BL, will move the contents of the BL register (8 bits) into the AL register. When using register addressing, the CPU performs all operations internally. In other words no 20 bit address is generated to specify the operand.

DIRECT ADDRESSING

The direct addressing mode completely specifies within the instruction, the location in memory that contains the operand. In this type of addressing, a 20 bit address must be formed. This means that some internal address generation on the 8086 will take place. Here is an example of this type of addressing:

```
MOV CX,COUNT
```

The value of COUNT is a constant. It is used as the offset value in computing the 20 bit address. The 8086 always uses a segment register when computing an actual address. Which register will be used for this particular instruction? Unless instructed otherwise, the DS or data segment register will be used. This is the default segment value. However, any of the four segment values may be used. This is accomplished by specifying the register appropriate in the instruction. For example, suppose we wish to use the ES register, rather than the DS register, we would then specify:

```
ES:  
MOV CX,COUNT
```

Where ES is the "segment override prefix" for the instruction. Its value will be used as the segment register for computing the 20 bit actual (or physical) address.

REGISTER INDIRECT ADDRESSING

With the register index addressing mode, the 16 bit offset address is contained in a base or index register. That is, the address resides in the BX, BP, SI or DI register. Here is an example of this type of addressing:

```
MOV AX,[SI]
```

The 16 bit binary value contained in the SI register would be the offset used to compute the 20 bit address. Again, a segment register will be used to generate

the final address. The 16 bit value in SI is combined with the proper segment value for address generation

REGISTER INDIRECT WITH DISPLACEMENT

This type of addressing involves the two previous modes of addressing. The 16 bit offset address is computed by adding the 16 bit value specified by the internal register and a constant. For example, let's use the internal DI register and the constant value (displacement), where COUNT has been previously defined as some number equal to COUNT. The mnemonics for this type of addressing are:

```
MOV AX, COUNT [DI]
```

If COUNT = 0378H and DI = 04FAH, then the 16 bit offset address will be 0872H which is the sum of 0378H + 04FAH.

REGISTER INDIRECT WITH A BASE AND INDEX REGISTER

This addressing mode uses the sum of two internal registers on the 8086 to obtain the 16 bit offset address to be used in the computation of the 20 bit address. Here are examples of this type of instructions:

```
MOV, [BP] [DI], AX  
MOV AX, [BX] [SI]
```

In this instruction, the offset A equals BP + DI and the offset B equals BX + SI.

REGISTER INDIRECT WITH BASE + INDEX + CONSTANT

This is the most complex addressing mode. If it is taken one piece at a time, it is not too difficult. This addressing mode is identical to the previous

addressing mode, except that there is a constant added onto the final 16-bit sum. For example, let's suppose that the following register values are given:

DI = 0367H

BX = 7890H

COUNT = 0012H

This addressing mode indicates that the offset specified by the sum of DI + BX + COUNT be used to move the memory data into register AX. The instruction would be written as:

`MOV AX, COUNT [BX] [DI]`

The 16 bit offset address would be equal to 7C09H, which is the sum of 0367H + 7890H + 0012H. The complete 20-bit address would be computed by the equation $[16 \times \text{DS}] + 7\text{C}09\text{H}$. If the DS register equals 3000h, then the complete, physical 20 bit address would equal to $[16 \times 3000\text{h}] + 7\text{C}09\text{H} = 37\text{C}09\text{H}$.

Note: The assembler used to run the examples in this book will not handle all of the 8086's addressing modes listed above.

Chapter 3 - The Assembler Environment

THE ASSEMBLY PROCESS

The job of an assembler is to convert our source code, written in assembly mnemonics, into hexadecimal machine code that the 8086 can understand. It achieves this by taking each mnemonic each turn and looking it up in a look-up table where it extracts the appropriate hex number or numbers. Our look-up table is stored in a file called MNEMONIC.DBF.

This assembly process relies on the correct spelling, or syntax, for each instruction being used. In order to speed up the assembly process I have not used the official Intel mnemonics for the different addressing modes. The main differences are as follows:

MOV AX, 0123	enter as	#MOV AX,0123
MOV AL,03		#MOV AL,03
MOV AX,[3000]		>MOVAX,[3000]
MOV [3000],AX		<MOVAX,[3000]

Other differences will be become apparent as we enter the example programs.

TWO-PASS ASSEMBLY

If we had to manually calculate the absolute address for every JUMP destination, life would be very difficult. Symbolic addresses are the answer to this problem. Each destination address is given a label, which with our assembler, must be four characters long. In order to turn these symbolic addresses into actual addresses, the assembler needs to run through the source code twice. After the second pass, all the address labels are replaced with absolute addresses.

THE DISASSEMBLY PROCESS

Disassembly is the opposite to assembly. This process simply converts the machine code back to 8086 mnemonics. We can then check that our original source assembled satisfactory. Under the Assemble option in the menu, you will see the unassemble option. Enter either an address (hex) or press the "X" key to disassemble from the default location.

MACHINE CODE AT LAST!

We have examined the registers and various addressing modes of the 8086, and briefly looked at the assembly process. We now need to look at entering and running machine code on the Psion 3. In other words, the interesting part.

The three OPL programs and the mnemonic file listed in the appendix will need to be entered and translated first. These are `DEBUG.OPL`, `ADISS.OPL`, `DBASE.OPL` and `MNEMONIC.DBF`. The `REM` statements can be omitted but be particularly careful with the variable names. Notice that the `DEBUG` program is an application that makes use of a suitable `ICON` picture. If you don't possess an `ICON` designer just omit the first four lines from the `DEBUG.OPL` program.

The program was written to run on a Series 3a machine, but by modifying the `PRINT AT` statements and replacing references to `FINDFIELD` with `FIND`, it should run all right on a Series 3.

The source editor, `DBASE.OPL`, is the program we are going to use to enter and edit our mnemonics. This program opens up either an existing or a new `DBF` file. Each line of source code is entered and stored as an entry in the database file.

Although it is more difficult to edit lines of text using the database than if we had decided to use the built in word processor, it solves other major problems and also speeds up the assembly process considerably.

Chapter 3 - The Assembler Environment

The editor (DBASE) program is only loaded in by the DEBUG application as and when required:

```
PROC proc:
LOADM"LOC::M:\OPO\DBASE.OPO"
STATUSWIN OFF
FONT 10,16
ED:
UNLOADM"LOC::M:\OPO\DBASE.OPO"
ENDP
```

Enter the ADISS program in a similar fashion. This also, is only loaded in by the main program when required.

Finally, the mnemonics need to be entered carefully and saved as MNEMONIC.DBF. Remember to change the labels to TEXT, CODE and LENGTH before typing in the mnemonic data entries. The Edit Labels option on the Edit menu lets you rename or delete label names in the same way as you edit an entry. After changing the label names, press the TAB key before beginning the data entry. Use the delete key to remove any unwanted labels. Consult your user guide if you are not sure how to rename labels or how to type in the entries from the mnemonic list. For example, from the list in the appendix, "MOV AH,AL,88C4,2," would be entered as figure 5.



Figure 5

You don't of course have to enter all the mnemonics in one go, neither do they have to be in any particular order. What is important is that you enter them exactly as shown.

ADDING EXTRA CODES

If you have access to a PC you can add extra codes to the MNEMONIC.FILE:

1. At the C> prompt, type debug.
2. Type the single letter "a" followed by enter. Note the full address.
3. From appendix 1 or another suitable code source, enter the mnemonic required, i.e. mov bx,ax (upper or lower case).
4. Press enter twice.
5. Type the single letter "u" followed by the address noted above. For example: u 29EC:0100. Your address will be different of course.
6. The screen should show the following:

```
-u 29ec:0100
29EC:0100 89C3          MOV      BX,AX
etc..
```

7. Make a note of:
 - a) MOV BX,AX
 - b) 89C3
 - c) the code length, in this case 2.

You now have all the information required to add this code to your existing database file:

```
Text: MOV BX,AX
Code: 89C3
Length: 2
```

Ensure that when adding codes that use the immediate addressing mode, always use FF or FFFF as the operand in MNEMONIC.DBF file (don't enter the # symbol in the mnemonic file):

```
Text: MOV AX,FFFF
Code: B8FFFF
Length: 3
```

Also with the direct addressing mode:

```
Text: MOV AL,[FFFF]
Code: A0FFFF
Length: 3
```

Note the square brackets.

If you delete an entry from your database, the space it takes up is not automatically reclaimed. Also, when you re-edit an entry and press Enter to put in the new version, space is still taken up by the old data.

The "Compress File" option on the "File" menu will reclaim any such space taken up by the old information. Using this will help keep your file length down. Use it before copying your mnemonic file to a Flash SSD.

An important point to remember is that before our Psion Assembler can assemble anything, the MNEMONIC.DBF file must be on the internal drive in the \DAT\ directory. Copy it over if you have saved the file to flash SSD.

Also note, that I can only guarantee that the instructions in the example mnemonic file assemble and disassemble correctly. In order to handle all of the available addressing modes, index addressing, base addressing etc., available on the 8086, the assembler would have to be prohibitively complicated.

WHERE IN MEMORY?

Our first problem is where in memory do we store our assembled machine code? We could use the heap and allocate space as described in the programming manual. This however, can lead to complications and it would be better if we could make life easier for ourselves.

What we are going to do is declare 8192 integer variables that will give us 16 kbytes and then store our code there. This will almost certainly be sufficient space. The OPL instruction for declaring a large number of integer variables is:

```
GLOBAL BUFFER%(8192)
```

What we don't know of course is the address of this area of memory. The following line of code answers this question:

```
ORG&=ADDR(BUFFER())
```

The variable ORG& now contains the starting address of our protected area of memory.

The assembled code can be stored here by default with the assembler directive ORG ?, or we can decide to place it anywhere we like by using the ORG command followed by a suitable hex address:

```
ORG 1000
```

Remember that a suitable hex address is not necessary a safe address.

This, for example, will almost certainly result in a system crash:

```
ORG 0000
```

Stick with ORG ? to be on the safe side.

Having previously saved all your important data and program files to a flashpack or onto floppy, via the serial link, a crash shouldn't be the disaster it might be. Experience and common sense will lead you to safe memory areas.

A FIRST LISTING

A typical program in source form will look like this:

```
1  ORG?
2  *LOOP
3  #MOV BX,FFFF
4  *DLAY
5  DEC BX
6  JNZ*DLAY
7  #MOV AX,1201
8  .INT 8E
9  JNZ*LOOP
10 RETF
11 END
```

Don't attempt to enter this program. We'll just examine it carefully, line by line.

The ORG command tells the assembler where to place the machine code. Following this are nine lines of source code, terminated by END.

The line numbers appear on the left. When you come to use the text editor, you will notice that the line numbers become duplicated or out of sequence. This is not particularly important and won't affect the correct running of the assembler.

1. The ORG statement can be entered as ORG? or ORG ?. This informs the assembler that the code is to start in a safe area of memory as mentioned above.
2. *LOOP is an address label. Note the asterisk. This must precede all labels.
3. The # symbol before the MOV BX,FFFF tells the assembler that the instruction is immediate. In other words, FFFF hex is the data to be loaded into the AX register.
4. Again, note the * before the label DLAY.
5. DEC BX simply decrements the BX register. (BX minus 1)
6. JNZ*DLAY is an instruction that causes the program to jump back to the address called DLAY. This will only occur if the contents of BX are NOT ZERO. Note carefully the difference in syntax between:

1. JNZ*DLAY and
2. JZ *DLAY

When using JZ ensure that there is a space between the Z and the *.

7. Like line 3, this is an immediate instruction.
8. The .INT 8E instruction is an interrupt call to Psion's operating system. Note the full-stop before INT.
9. JNZ*LOOP is a Jump on Non Zero back to the label *LOOP. In actual fact this never results in zero and the program will continue indefinitely or until the escape key is pressed.
10. The importance of RETF was stated earlier. Although in this particular program it seems to serve no purpose (see 9 above) all your programs should finish with this safe FAR RETURN.

11. The END statement is only used in the editor. It allows you to exit the edit mode and return back to the dump screen. When the assembler encounters the END directive it will throw up an error but we can ignore this.

On leaving the editor, the file will be saved automatically and we can then assemble it or list it.

You can of course, return to the source file and edit it later if required.

During the first pass of the assembly process the label names are unresolved and appear as FFFF in the code listing. During the second pass these are correctly replaced by absolute addresses, which can be either forward (absolute address plus 127) or backwards (absolute address minus 128). In the example above, the relative jumps as these are called, are all backwards.

The first pass produces this:

1	ORG?		Start address: 82275 (8821 dec)	
2	*LOOP			
3	*MOV BX,FFFF	88FFFF		2275
4	*DELAY			
5	DEC BX	4B		2278
6	JNZ*DELAY	75FF	Jump label	2279
7	*MOV AX,1281	888112		227B
8	INT BE	CD8E		227E
9	JNZ*LOOP	75FF	Jump label	2280
10	RETF	CB		2282
Code length: 14. End address: 2282(hex)				
2 Errors				

Figure 6

And the second pass:

1	ORG?	Start address: \$2275 (8821 dec)	
2	*LOOP		
3	SHOW BX,FFFF	BBFFFF	2275
4	*DELAY		
5	DEC BX	4B	2278
6	JNZ*DELAY	75FD	2279
7	SHOW AX,1281	88B112	227B
8	JNT 8E	CD8E	227E
9	JNZ*LOOP	75F3	228B
10	RET	CB	2282
Code length: 14. End address: 2282(hex)			
0 Errors			

Figure 7

Notice the two errors produced after the first pass in Figure 6. As mentioned above, this is because of the unresolved address labels.

The program has been correctly assembled at address 2275 hex. On your machine, this address will almost certainly be different. This doesn't matter however, as your safe area is as safe as mine.

Notice also that the address labels, or symbolic addresses as they are more correctly called, have been replaced by actual addresses. We can only jump relative to our present address by +127 or -128. This means that any backward jumps will result in operands between 80 h and FF h, and forward jumps in numbers between 00 h and 7F h.

On completing its second pass the assembler will display a symbol table showing the actual addresses represented by the labels:

Symbol table:

LOOP...2275
DLAY...2278

Figure 8

Again, these addresses will be different on your machine.

Chapter 4 - First Steps in Programming

OUR FIRST PROGRAM

We can now proceed to enter, assemble and run our first program.

This program, called ADDER, will simply store a number in the AL register and a second number in the AH register. Later we will add these two numbers together, but for now we'll just move them into the appropriate registers.

Remembering that the AH and AL registers can only hold numbers between 0 and 255, we'll keep things simple:

```
ORG ?
MOV AH,05
MOV AL,03
RET
```

Start up DEBUG and selected Edit from the menu:

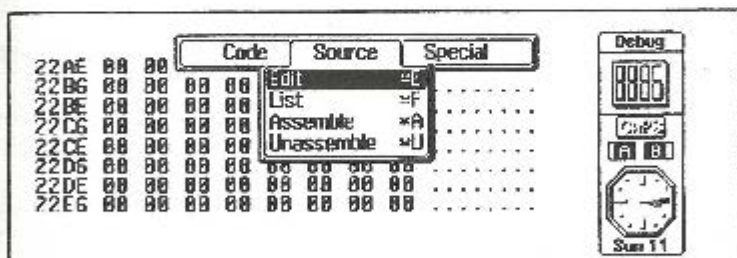


Figure 9

At the "Edit" prompt, type in ADDER and press the enter key. Debug will inform you that this is a new file.

Press enter after each line of code:

```
ORG ?  
#MOV AH,05  
#MOV AL,03  
RETF  
END
```

Notice how the final END statement leaves the edit mode.

Now press Q to quit the editor.

Before we go onto assemble this, we'll take a look at the DIR and LIST options:

Select Dir from the menu:

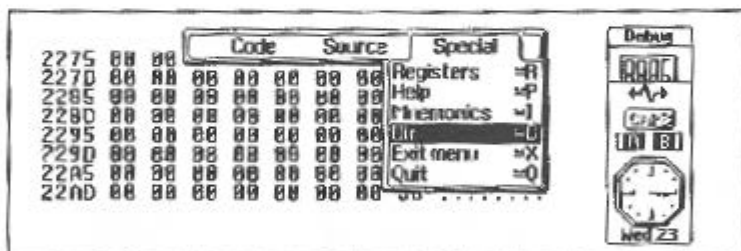


Figure 10

The filename ADDER.DBF should be displayed in the directory listing

We can List this file without editing it with the List option.

Use List from the menu:

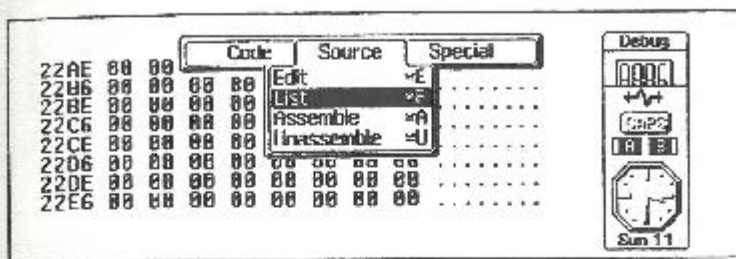


Figure 11

Press the space bar to move through the file. The Q key takes you out of the List mode.

Now select Edit from the menu and enter ADDER at the prompt. You are now back in the Editor. You can scroll through the listing with the space or arrow keys. The Help and Menu keys will display a simple help screen. From this you can see that we can Append, Insert or Delete a line from the source code.

EDITING THE SOURCE FILE

As an exercise, we'll delete one of our lines of code and replace it with a different line:

Using the space bar, move through the code until MOV #AL,03 appears: (Note that the up arrow can be used to scroll backwards through the listing.)

```
1 ORG ?
2 #MOV AH,05
3 #MOV AL,03
```

Now press the D (delete) key. The display shows line 4 RETF. Scroll through the listing and the line containing #MOV AL,03 has vanished. To replace it, scroll through the listing until #MOV AH,05 is displayed:

```
1 ORG ?  
2 #MOV AH,05
```

Press the I (insert) key. Now enter the new line of code:

```
MOV #AL,04
```

followed by Enter. Finally, using the space bar, scroll through the lines of code confirming that the new line has been entered correctly and in the right place:

```
1 ORG ?  
2 #MOV AH,05  
2 #MOV AL,04  
4 RETF  
5 END
```

Don't worry about the line numbers being all mixed up.

Practice deleting and replacing the above lines of code. I admit that it is a little bit awkward, but it works OK.

EXECUTING THE MACHINE CODE

When you are reasonably happy with your source code, press Q to return to the main screen, where we can now proceed to assemble and run the resulting machine code.

From the menu, select Assemble.

Choose ADDER from the file list and press return. After the first Pass, you should have the screen in figure 12 displayed:

```

1  ORG ?           Start address: $2275 (8921 dec)
2  MOV AH,05       8105      2275
3  MOV AL,04       8804      2277
4  HLT             CB        2279
5  END             remove END
Code length: 5. End address: 2279(hex)
1 Errors

```

Figure 12

As there are no symbolic addresses to resolve, the second pass, obtained by pressing the space bar, will be exactly the same as the first pass. Note that you can ignore the single error and the associated message "remove END". This is only a warning and whether you remove the END statement or leave it in is up to you.

Note that the displayed Symbol table will be empty.

Pressing the space key for the final time will display the screen in figure 13:

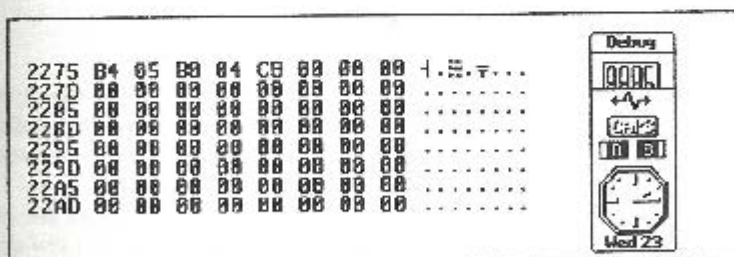


Figure 13

Again, note that your addresses will probably be different to the above. However, if the code itself is different, return to the editor and modify it as necessary.

Before we move on to execute the code, we'll disassemble it:

Select Unassemble from the option list. At the "Disassemble from (HEX):" prompt, enter X. Move through the listing with the space bar:

14D3:22AE	B405	MOV AH,5
14D3:22B0	0003	MOV AL,3
14D3:22B2	CB	RET
14D3:22B3	0000	???
14D3:22B5	0000	???
14D3:22B7	0000	???
14D3:22B9	0000	???

The 4 digit number before the colon is the segment address. The RETF marks the end of the disassembled code so you can ignore everything after this. Press D or U to disassemble again or Q to quit.

Finally to run your code do the following:

From the menu select Modify/Run, see figure 14 overleaf.

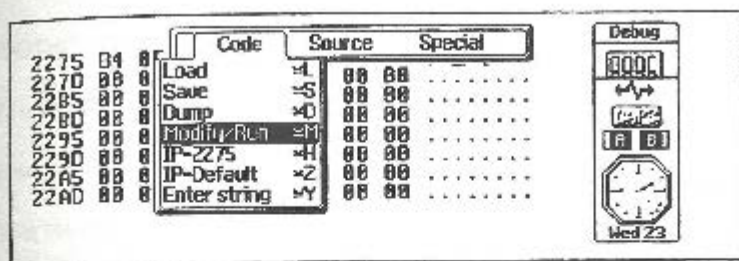


Figure 14

After selecting this option, the screen will be as figure 15:

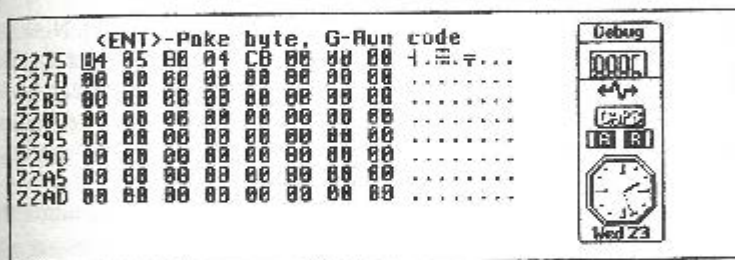


Figure 15

The flashing cursor can be moved around the display by using the arrow keys. Before running the code, ensure that the cursor is positioned at the start address, which on my machine is 2275 hex.

We will look at the Poke byte option shortly, but for now just press the G key. You will now be asked to confirm the starting address. If the starting address is incorrect, press the escape key. Assuming however that all is well, press the Y key. The display will confirm that the code has been executed and will return with the contents of the AX register:

Code executed ** AX=504

We have correctly stored 05 in the AH register and 04 in the AL register.

MODIFYING THE CODE

Let's try a little experiment.

We will replace the 05 with 06 and run the program again. It is not necessary to return to the editor and re-assemble, as we can poke the new code using the modify option as follows:

Press the space bar to return to the last screen above.

Using the cursor keys, position the cursor over the second byte (05). Note that pressing the space bar will toggle between the two nibbles. Select the first nibble and enter 06. At this stage the change is not permanent until you hit the enter key. If you hit the escape key, the previous contents will be displayed. This provides a little bit of insurance against incorrect entries.

Having changed the 05 to 06, press the enter key. A message will confirm that the modification has been made.

Finally position the cursor at the starting address (over the B4 byte) and press the G key to run the modified code.

The display will now show:

Code executed ** AX=604, which is correct!

ADDING AH and AL

We can now move onto adding these two numbers together.

To accomplish this we will modify our previous source code

Using the edit option in Debug, load in the file ADDER and modify the source as follows:

```
ORG ?  
#MOV AH,05  
#MOV AL,04  
ADD AL,AH  
DAA  
RETF  
END
```

The ADD command will add the contents of AH to AL, placing the result in the AL register overwriting its previous contents. The number in AH remains unchanged.

The meaning of DAA

DAA is the mnemonic for Decimal Adjust Accumulator. This instruction changes the contents of the AL register to a pair of valid packed decimal digits:

DAA

Before:	After:
---------	--------

AL = 8Ah	AL = 90h
----------	----------

See the full definition in appendix 2.

Chapter 4 -First steps in Programming

Once we run the ADDER program you will understand this instruction more clearly.

After pass 1 and pass 2 of the assembly stage, your screen should look like figure 16:

1	ORG ?	Start address: \$2275 (9821 dec)
2	#MOV AH,05	8405 2275
3	#MOV AL,04	8804 2277
4	ADD AL,AH	08E0 2279
5	DAA	27 227B
6	RET	C8 227C
7	END	
Code length: 8. End address: 227C(hex)		
1 Errors		

Figure 16

Once again bear in mind that your addresses and line numbers will be different to the above. The code and its length should both be identical however.

On running the machine code, AX will contain the 0509. In other words, AH = 05 and AL = 09 (05 + 04), which is correct.

We can now see the effect of the DAA instruction simply by replacing the 05 in AH with 06. Use the Poke byte option to change the second byte in the screen dump to 06. Do not forget to press the enter key.

Run the program from the start address. This time AH = 06 and AL = 10 and not 0A as expected. Remember that 06 + 04 = 0A in hexadecimal.

Replace the 27 (code for DAA) with 90 (code for NOP) and run the program again. This time AL contains 0A. We have removed the decimal adjust function.

Try replacing 06 and 04 with different numbers and swapping 90 with 27 and back again a few times. You will soon learn the importance of the DAA instruction.

Chapter 5 - Further Menu Options

Menu Options

We will now take a look at some of the other menu options in Debug.

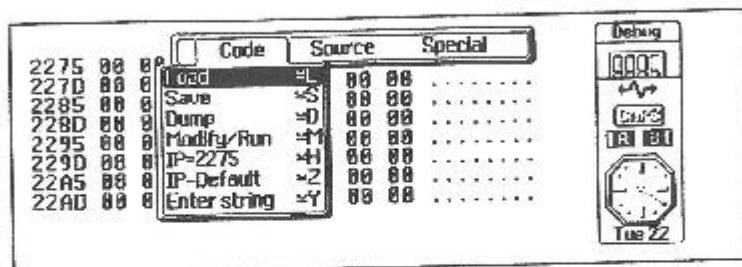


Figure 17

LOAD AND SAVE

Load and Save operate on machine code only. These options enable you to save a block of code and reload it, either at the original address or at a new address. For example, you could save a 100 byte block of zeroes, calling the file "BLANK", and reload this code each time you needed to clear an area of memory. You could reload small code sections consecutively, building up a larger chunk of code and resave. Each of these small sections can be assembled and run separately.

As an exercise, re-load the ADDER program saved earlier. Assemble the code and run through the following steps after noting the code's starting address:

1. Select SAVE from the menu.
2. Enter a new file name, e.g. TEST and press enter.
3. Enter the machine code starting address that you noted earlier, (2275 in my case). Note that you don't need to press the enter key here.

4. Byte length(HEX). Enter a suitable code length. In our example, 9 is adequate. Be sure to enter this as 0009 (no ENTER key).
5. The code file has now been saved as TEST.DBF

To reload the code:

1. Select LOAD from the menu.
2. Select the file (TEST) from the file list.
3. Enter a suitable load address. Be careful here, you could overwrite an important area of memory. Use the previous saved address plus 10. Once again, do not press enter after the fourth hex digit.
4. Your saved code should now be re-loaded at the selected address.

DUMP

This option allows you to examine areas of memory within the current 64 kbyte segment. Just enter a hex address at the prompt without pressing the enter key. Note that you cannot examine the ROM with the DUMP option.

The cursor keys do the following:

Down arrow - scroll forward a page. (Plus 40 bytes) Note: Don't scroll beyond FFFF.

Up arrow - scroll backward a page. Note: Don't scroll less then 0000!

Right arrow - move right by one byte. (Address plus 1).

Left arrow - move left by one byte.

Pscion key + down arrow - scroll forward by 8 bytes.

After scrolling around memory you can quickly return to your original address with the **IP = (last address)** menu option. Select this and notice how your original dump address becomes the screen starting address.

To return to the reset condition, select the **IP = Default** option. The dump address now returns to our "safe" area of memory. This also resets **IP = (last address)**.

ENTER STRING

This menu option allows you to enter a text string at the displayed starting address. The length of the string must be less than 255 characters. Press the enter key to terminate the string entry. You can save the string with the **SAVE** code option and reload it to another area of memory if required.

REGISTERS

The register option displays the contents of the various registers of the 8086. Note that the contents of **AX**, **BX**, **CX** and **DX** are displayed as empty. See figure 18.

8086 REGISTERS				
AX=0000	BX=0000	CX=0000	DX=0000	
CS=1024	DS=1024	SS=1024	ES=1024	
SI=0000	DI=1A6C	SP=07D0	BP=0B3C	

Figure 18

MNEMONICS

This option list the contents of the MNEMONIC.DBF file. It will prove useful to see if a particular code exists and needs to be added or modified.

I have not included every 8086 code, neither are the extra V30 instructions in the database file.

DIR

Use this to list all the DBF files on the internal drive.

HELP

This brings up the General Help menu.

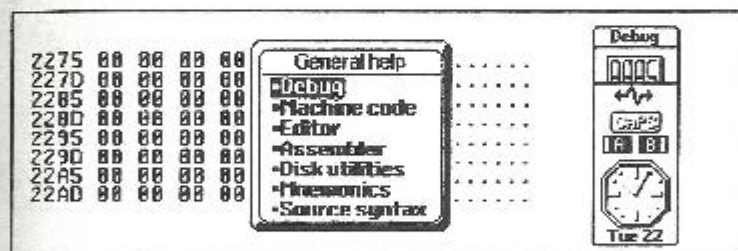


Figure 19 - General Help

Each of the seven help screens are shown in figures 20 to 25.

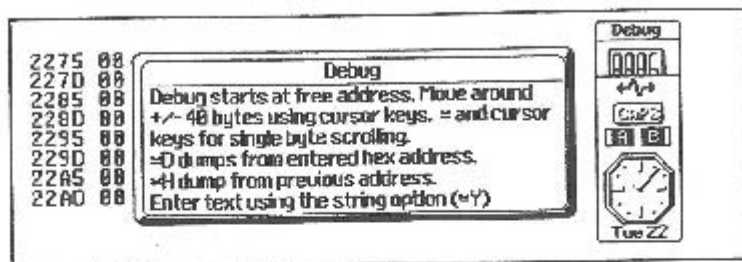


Figure 20 -Debug's dump option

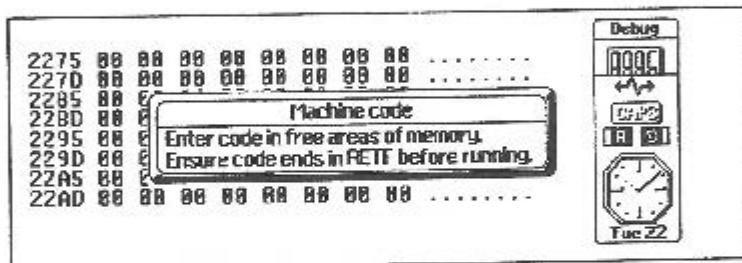


Figure 21 - General Machine Code help

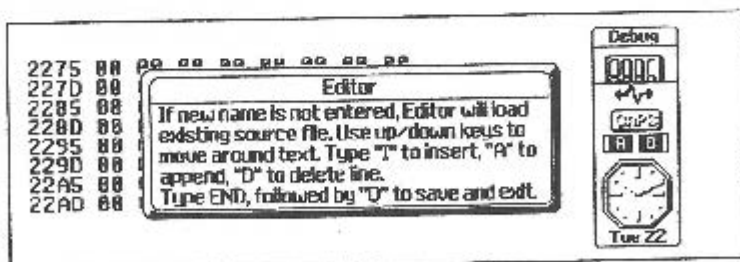


Figure 22 - Help with the Editor

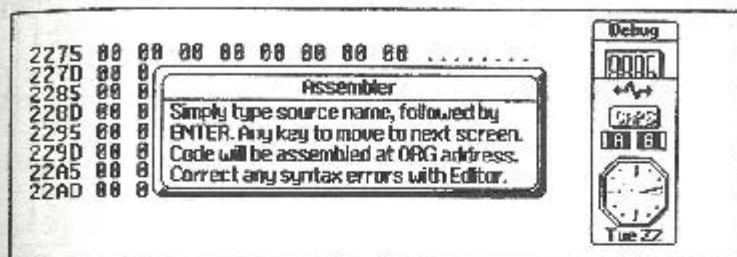


Figure 23 - Assembler help screen

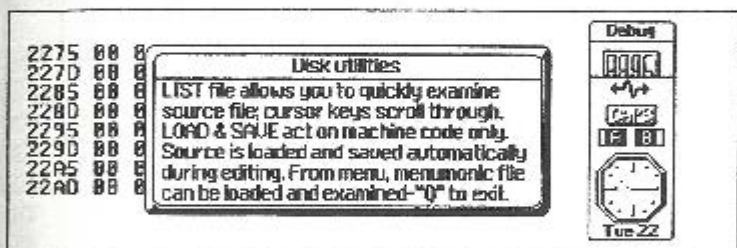


Figure 24 - Disk utilities help

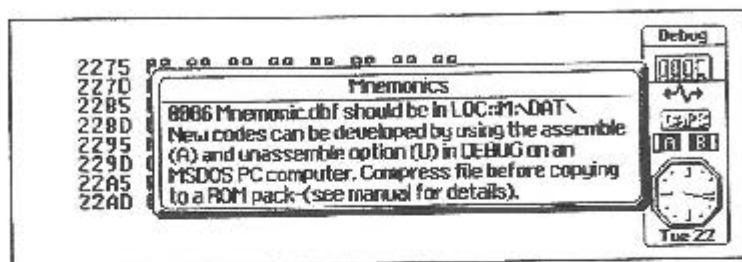


Figure 25 - Mnemonic help

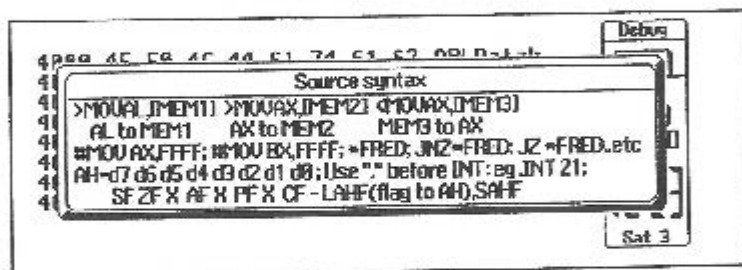


Figure 26 - Syntax help

Chapter 6 - 8086 Arithmetic Functions

ADDING AH AND AL WITH CARRY (ADC)

We will now modify our original ADDER program to take into account the effect of the result of the addition being greater than 255 or FF hex.

When this occurs after $AH + AL$, the CARRY flag will be set to 1.

Referring to figure 26, you can see that the CARRY flag is the first bit within the flag register i.e. bit d0. With the aid of the LAHF instruction, we can move the contents of the flag register into the AH register.

Before we carry out the ADC instruction we must first ensure that the CARRY flag is set to 0. The CLC, Clear Carry Flag, instruction does this.

Enter the following source code using the editor:

```
ORG ?  
CLC  
#MOV AH,01  
#MOV AL,FF  
ADC AL,AH  
LAHF  
RETF  
END
```

After Pass 1, you should have figure 28.

1	ORG ?	Start address: \$2275 (8821 dec)	
2	CLC	F8	2275
3	MOV AH,B1	B4B1	2276
4	MOV AL,FF	B8FF	2278
5	ADC AL,AH	18E0	227A
6	LAHF	9F	227C
7	RET	CB	227D
8	END		

Code length: 9. End address: 227D(hex)

1 Errors

Figure 28

After assembling the source run the machine code.

The contents of AX should be 5700, which in binary is:

0101 0111 0000 0000

If you look carefully at the separate bits of AH you will see that both the zero flag and the carry flag are both set:

AH = d7 d6 d5 d4 d3 d2 d1 d0

Flags = 0 1 0 1 0 1 1 1

d6 contains the zero flag, which is set if the result is zero, and d0 contains the carry flag which is set in this case.

This is exactly what we would expect as the result in AL is zero with a carry.

In binary, FF + 01 is:

1111 1111 + 0000 0001 = 0000 0000 plus a carry. Therefore CF = 1.

Whereas:

$1111\ 1100 + 0000\ 0001 = 1111\ 1101$ with no carry. Therefore $CF = 0$.

Try changing line four of the source to:

```
ORG ?
CLC
#MOV AH,01
#MOV AL,FE
ADC AL,AH
LAHF
RETF
END
```

Remember that you can use the Poke byte option if you prefer not to modify the source code.

After running the above, $AL = FF$ ($FE + 01$) and $AH = 86$. (Ignore the leading FFFF in the display).

In binary:

```
AH = 1000 0110
AL = 1111 1111
```

Again if you examine the separate bits of AH you will see that both the zero flag and the carry flag are both cleared:

```
AH = d7 d6 d5 d4 d3 d2 d1 d0
```

```
Flags = 1 0 0 0 0 1 1 0
```

d6 contains the zero flag, which is cleared if the result is not zero, and d0 contains the carry flag which is also cleared.

Once again this is exactly what we would expect as the result in AL this time is not zero and there has been no carry.

SUBTRACTION

The mnemonic for subtraction is SUB. Using this instruction the source is subtracted from the destination and the result is placed in the destination:

Destination = Destination - Source

Using the editor, enter the following:

```
ORG ?
#MOV AH,03
#MOV AL,09
SUB AL,AH
DAS
RETF
END
```

The SUB instruction will subtract the contents of AH from AL, placing the result in AL. The DAS is a special instruction that performs a Decimal Adjust for Subtraction.

After pass 1 and pass 2, your screen should look like figure 29.

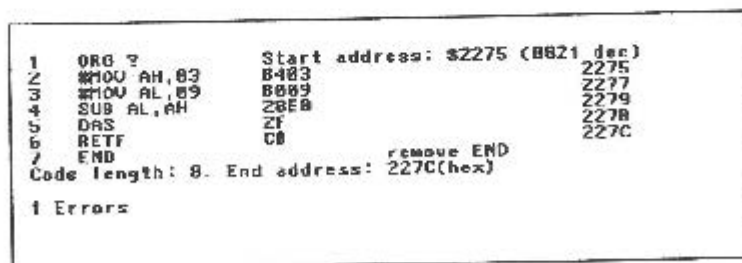


Figure 29

Run the resulting machine code and confirm that the subtraction produces `AL = 06`.

As an exercise, try different numbers in `AH` and `AL`.

MULTIPLICATION

The 8086 can multiply and divide a variety of operands together. For example, `MUL CX` will multiply the contents of `CX` with `AX` and `MUL CL` will multiply the contents of `CL` with `AL`. Whether `AL` or `AX` contains the result, depends on the source either being a byte or a word.

I have included four multiplication instructions in the mnemonic file, and the one we are going to use is `MUL CX`.

With the 8086, it is possible to multiply two 16-bit numbers together. When the operation is complete, the result is placed in two 16-bit registers. This is necessary because the result is a 32-bit number.

After a 16-bit multiplication, one of the numbers to multiply is stored in the `AX` register. The double length (32-bit) result is stored in the `DX` and `AX` registers, with the most-significant 16-bits in the `DX` register.

If the multiply is to be an 8-bit operation, one of the numbers must be in the `AL` register. The 16-bit result is returned in the `AX` register, with the most-significant byte in `AH`.

After a 16-bit multiplication, we will transfer the contents of the `AX` register to a suitable memory location with `>MOVAX,[MEM1]`. The 8086 has an instruction which will transfer the contents of `DX` directly to a memory location, `MOV [ADDR],DX`, but our assembler as it stands will not recognise this particular instruction due to the way in which the assembly process operates. This is not as major a problem as it would seem. All we have to do is transfer the contents of the `DX` (or `BX,CX` etc) to the `AX` register and move this into memory:

```
MOV AX,DX  
>MOVAX,[ADDR]
```

The only question left to answer is where in memory do we place our result.

There are 9 memories available to store 16 bit numbers, MEM1 to MEM9.

We can store the contents of AX in MEM1, and DX in MEM2.

Enter the following source code:

```
ORG ?  
#MOV AX,1234  
#MOV CX,5678  
MUL CX  
>MOVAX,[MEM1]  
MOV AX,DX  
>MOVAX,[MEM2]  
RETF  
END
```

Be especially careful with the syntax. Note the orientation of the greater and equal signs inserted before the MOV instruction.

The assembler should produce the code shown in figure 30 overleaf.

```

1  ORG 7          Start address: $2280 (8845 dec)
2  IMOU AX,1234   883412          2280
3  IMOU CX,5678   897856          2290
4  MUL CX         F7E1          2293
5  >IMOUX, (MEM1)  A3A422          2295
6  MOV AX, DX     89D8          2298
7  >IMOUX, (MEM2) A3AC22          229A
8  RETF          CB            229D
9  END           remove END
Code length: 17. End address: 229D(hex)

1 Errors

```

Figure 30

After running the code, the number in AX will be 0626. This is the most-significant part of the answer transferred from DX.

At this stage the contents of MEM1 and MEM2 remain at 00. Press the escape key twice to update the display.

Your screen should look like figure 31 (ignore the address differences):

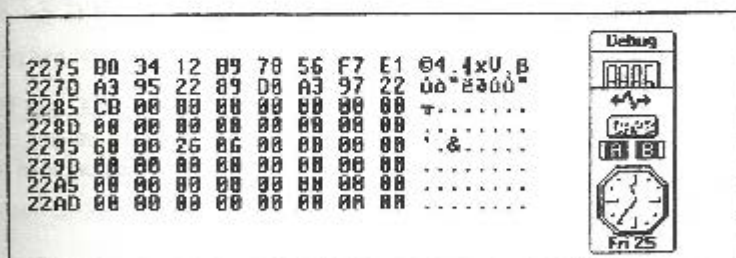


Figure 31

The final 32-bit result is 06260060 hex.

DIVISION

The DIV instruction performs a division of the accumulator by the source operand.

Enter and run the example in figure 32:

1	ORG ?	Start address: \$2275 (0021 dec)	
2	#MOV BX,0003	000300	2275
3	#MOV AX,0006	000600	2278
4	DIV BX	F7F3	227B
5	RET	C9	227D
6	END		
Code length: 9. End address: 227D(hex)			
1 Errors			

Figure 32

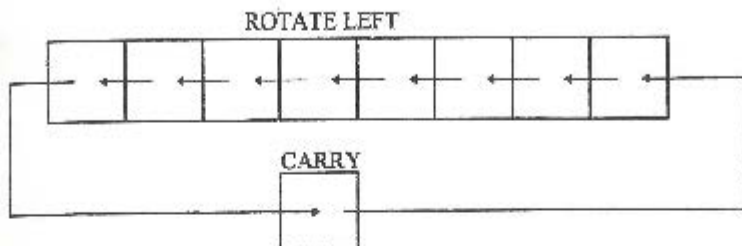
The result as expected is 2.

At the conclusion of the operation, the 8-bit quotient is returned to the AL register with the remainder being stored in the AH register.

When the dividend is 32-bits and the divisor is 16-bits, the AX and DX registers are used to hold the numbers. When the operation is complete, the 16-bit quotient is returned in the AX register and the 16-bit remainder is in the DX register.

See appendix 1 for further examples of MUL and DIV.

ROTATE RIGHT AND ROTATE LEFT



These important instructions rotate the bits in the operand to the right or to the left by the number of bits specified in the count operand. If the count equals 1, it may be specified directly. If the count is greater than 1, it must be set in the CL register before using these instruction.

The ROR and ROL instructions can act on many operands but our mnemonic file only includes the following:

ROR AX, ROR AL, ROR AH, ROL AX, ROL AL, ROL AH.

Why are these instructions so important?

To answer this question we must return to our multiplication and division programs.

Consider the effect of multiplying 3 by 2. In binary this is:

0011 times 0010

3 times 2 is 6, which in binary is:

0110

Chapter 6 - 8086 Arithmetic Functions

If we multiply 0110 (6) by 2, we get 12 of course, which is 1100 in binary.

0011 = 3, 0110 = 6 and 1100 = 12

If you examine the two set bits in each of the three numbers above, you will notice that they have been shifted left by one position.

In other words to:

Multiply by 2 we shift left 1 place
Multiply by 4 we shift left 2 places
Multiply by 8 we shift left 3 places

etc...

And to:

Divide by 2 we shift right 1 place
Divide by 4 we shift right 2 places
Divide by 8 we shift right 3 places

etc..

Enter the following:

```
ORG ?  
#MOV AX,FF00  
ROR AX,1  
RETF  
END
```

After running the code, notice the FF00 moved into AX has become 7F80.

In binary, 1111 1111 0000 0000 after ROR AX,1 becomes:

0111 1111 1000 0000

Try the following program and note the difference:

```
ORG ?  
#MOV AX,FF00  
ROL AX,1  
RETF  
END
```

This time the result is FE01 (ignore the leading four F's).

FE01 in binary is 1111 1110 0000 0001. Apart from the least significant bit, the remaining bits in the word have been rotated left one position.

Enter the following:

```
ORG ?  
#MOV AX,FF00  
ROR AX,1  
RETF  
END
```

After running the code, notice how the FF00 moved into AX has become 7F80.

FF00 hex in binary is 1111 1111 0000 0000 and after ROR AX,1 this becomes:

0111 1111 1000 0000

Try the following program and note the difference:

```
ORG ?  
#MOV AX,00FF  
ROL AX,1  
RETF  
END
```

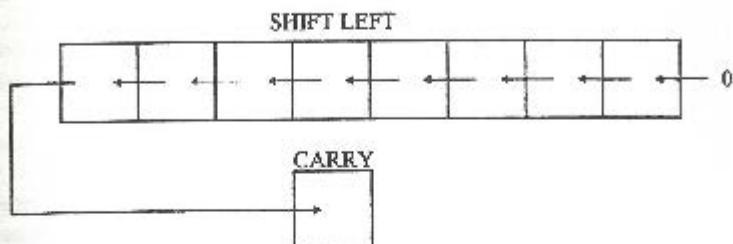
This time the result is 01FE which in binary is 0000 0001 1111 1110.

If you need to rotate left or right more than a single bit, use the CL register to hold the count:

```
ORG ?  
#MOV AX,00FF  
#MOV CL,04  
ROL AX,CL  
RETF  
END
```

The above program produces $AX = 0FF0$.

SHIFT LEFT AND RIGHT



SHL and SHR are similar to the rotate instructions, ROL and ROR, except that after a shift left operation, zeroes are shifted in on the right, and after a shift right, zeroes are moved in from the left.

Try the following two programs to see the difference between the shift and rotate instructions:

```
ORG 7
#MOV AX,FFFF
ROL AX,1
RETF
END
```

The ROL instruction produces $AX = FFFF$

```
ORG 7
#MOV AX,FFFF
SHL AX,1
RETF
END
```

The SHL instruction produces $AX = FFFE$

As an exercise, see if you can work out the result of:

```
ORG ?  
#MOV AX,FFFF  
#MOV CL,10  
SHL AX,CL  
RETF  
END
```

Chapter 7 - 8086 Subroutines and the Stack

THE STACK

An important area of memory is called the stack. It is here that return addresses are stored after a CALL instruction. We will be looking at subroutines and the use of CALL and RET later, but for the moment let's consider another important use for the stack.

The stack can be used to store data and intermediate results of calculations. It can be looked on as a scratchpad and treated as such. Where exactly the stack is in memory is of no real concern to us, but the 8086 needs to know where it is of course.

PUSH AND POP

The 8086 holds the address of the first free location of the stack in the SP (stack pointer) register. Every time something is pushed onto the stack, this register is decremented by a value equivalent to the byte length of the saved number. After this saved value is popped off the stack, the SP register is incremented in a similar fashion.

The following short program will give us the current SP contents:

```
ORG ?  
MOV AX,SP  
>MOV AX,[MEM1]  
RET  
END
```

There is also a menu option in Debug that will tell us the same thing.

In the 8086 architecture, it must be remembered that the actual physical memory address for the location of the stack is the sum of the contents of the Stack Segment register plus that of the Stack Pointer register. The Psion operating system makes frequent use of the stack, therefore changing the contents of the Stack Segment register will cause havoc.

THE STACK MECHANISM

Before we move on to our first program which makes use of the PUSH and POP instructions, it is very important that the stack mechanism is fully understood.

As we have seen, everytime a value is pushed onto the stack, the SP is decremented. How many times it is decremented depends on the size of the number pushed. Pushing a single byte will decrement the SP by 1, while pushing a word (2 bytes) will decrement it by two. In other words, PUSH AL and PUSH AX will have different effects on the stack pointer contents. It follows that the POP instruction operates similarly. POP AX increments the SP by two and POP AL increments the SP by 1.

As mentioned above, subroutine return addresses are stored on the stack. If the stack gets corrupted by unmatched PUSH and POP instructions, any saved return addresses will be lost and a system crash will be the immediate result.

Every POP instruction must be matched by an equivalent length PUSH instruction.

The following program, figure 33, illustrates the use of the stack. It simply moves the number five into the AX register. We then push AX onto the stack, clear AX and finally POP five off the stack back into the AX register.

1	ORG ?	Start address: \$2275 (8821 dec)	
2	MOV AX, 0005	000500	2275
3	PUSH AX	50	2278
4	MOV AX, 0000	000000	2279
5	POP AX	50	227C
6	RET	C0	227D
7	END		

Code length: 9. End address: 227D(hex)

1 Error*

Figure 33

Any legal register can be pushed and popped to and from the stack.

What will the code below do?:

```
ORG ?  
#MOV AX,00FF  
PUSH AX  
POP BX  
RETF
```

In effect it will perform the same function as MOV BX,AX.

See if you can work out why the following code will not work in a similar fashion to the program above:

```
ORG ?  
#MOV AX,00FF  
PUSH AX  
POP BX  
POP BX  
RETF
```

You have probably worked out by now that running the above program will result in a system crash!

The reason of course, is that at the end of our machine code routine we need a far return to return safely back to the Psion system and the RETF instruction pops the return address back to the operating system from off the stack. In the above program, the number of pops do not match the single push instruction. The 8086 can't find the correct return address because we removed it.

In the following example, the pushes and pops match up and the return address is not lost:

```

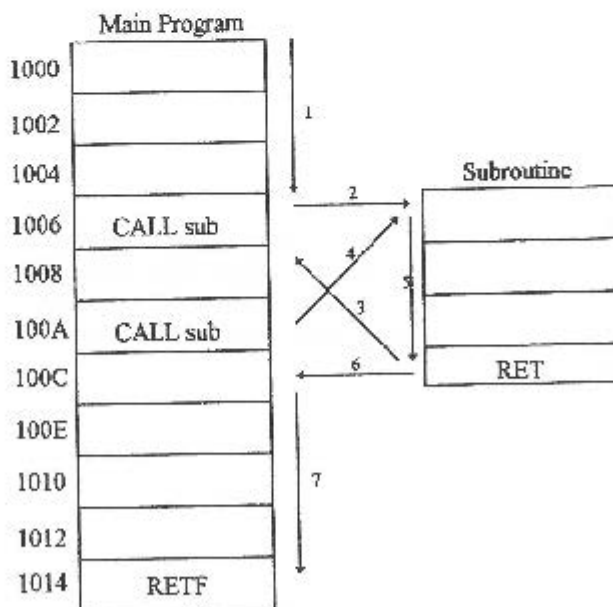
ORG ?
#MOV AX,00FF
PUSH AX
PUSH AX
POP BX
POP BX
RETF
    
```

SUBROUTINES

A subroutine is simply a section of code which can be repeatedly called from the main program by means of a CALL instruction.

A subroutine must end with either a near return, RET, or a far return, RETF.

Figure 34 shows how a subroutine is used.



The main program, or calling program, starts at address 1000. The subroutine can be located at any convenient memory address. If the main program does not extend further than address 1014, as in the above example, then subroutine could be located safely at 1020 onwards.

When the Instruction Pointer reaches address 1006, the CALL instruction is executed. This results, firstly in the return address, (1008) being PUSHed onto the stack, followed by a transfer to the subroutine section of the program. The instructions in the subroutine are then executed until the RET instruction is encountered. The return instruction causes the 8086 to POP 1008 off the stack and into the instruction pointer where program flow continues down to address 100A. Here, once again, a return address is pushed onto the stack (100C) and the subroutine is executed once more until the RET forces the processor to return to the main program. This finishes at address 1014 where the RETF instruction returns us back to the Psion operating system.

REASONS FOR USING SUBROUTINES

What is the use of a subroutine? The essential feature of a subroutine is that it can be called from any number of points in a main program, and the instructions in a subroutine can be used repeatedly without having to be rewritten each time. An advantage of this approach is that it saves memory space because the subroutine doesn't need to be rewritten each time. Another advantage is that you only need to design a specific subroutine once, as it can be used repeatedly. You will find that this significantly simplifies your program design process.

There are two types of subroutine calls.

INTRA-SEGMENT and INTER-SEGMENT

If the subroutine lies within the same 64 k block as the main calling program, this is called an INTRA-SEGMENT CALL. On encountering this type of CALL, the 8086 saves the IP register on the stack. At the end of the subroutine the RET instruction POPs the saved address off the stack and into the IP register where program execution continues.

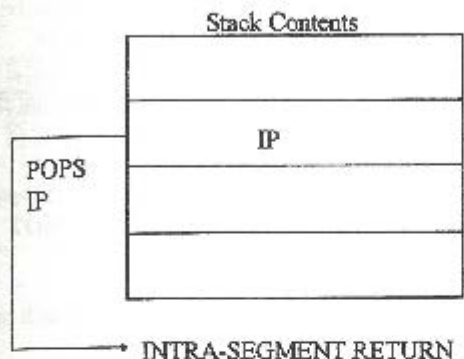
If the subroutine lies outside the 64 k block where the main program resides, this is known as an INTER-SEGMENT CALL. With this type of CALL the contents of the CS (Code Segment) register together with the IP register are both saved on the stack. At the end of the subroutine, the RETF instruction POPs the saved CS register and the IP register contents back to the relevant 8086 registers.

It is very important to use the correct return instruction at the end of your subroutines otherwise you will get what is known as a CALL-RETURN MISMATCH.

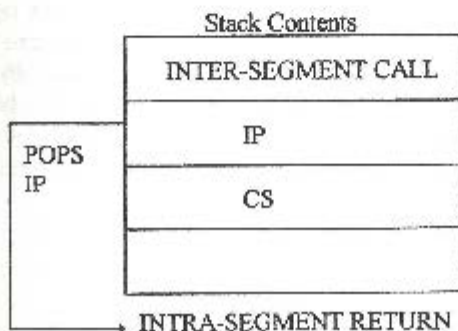
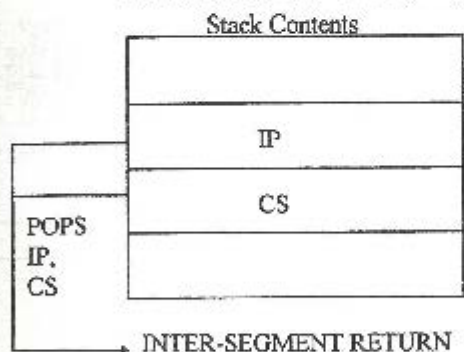
Our assembler as it stands, can only use subroutines that lie within the same 64 k block of memory as the main program. Always terminate your subroutines with the RET (near return) instruction.

The diagram on the next page should help you understand these points.

INTRA-SEGMENT CALL (NEAR)



INTER-SEGMENT CALL (FAR)



CALLING A SUBROUTINE

Before a call to a subroutine can be made, the subroutine's address must be loaded in either the BX, CX or DX registers. This address must be chosen with care, but by now you should have some idea where this can be.

To illustrate the use of the CALL instruction, we will run a program that calls a subroutine. See figures 37 and 38.

Enter and assemble the main program first of all. This will give you some idea where the starting address of the subroutine must be. DO NOT run the program at this stage.

The main program will begin with ORG ? whilst the subroutine will start with ORG ADDR, where ADDR is the chosen address.

1	ORG ?	Start address: \$2275 (8821 dec)	
2	M100 AX,8005	800500	2275
3	M100 DX,FFFF	BAFFFF	2278
4	CALL DX	FFD2	2278
5	RET	CB	227D
6	END		
Code length: 9. End address: 227D(hex)			
1 Errors			

Figure 36

In figure 36, notice the dummy value of FFFF loaded into the DX register. We will replace this shortly with the actual address of our subroutine. You will notice that after the assembly process, the last address used by the main program is 227D. This tells us that memory location 2280 will be perfectly safe for the location of our subroutine.

Load the main program back into the editor and change line 3 to the example below in figure 37, (remember your addresses will be different!):

```

1  ORG ?           Start address: $2275 (8821 dec)
2  MOV AX,0005     880500      2275
3  MOV DX,2200     8A0022      2278
4  CALL DX         FF02      227B
5  RET             C0         227D
6  END             remove END
Code length: 9. End address: 227D(hex)
1 Errors

```

Figure 37

If you are still in the editor, select N from the menu, and enter the following subroutine as shown in figure 38.

```

1  ORG 2280         Start address: 8832(dec)
2  DEC AX          4B         2280
3  RET             C3         2281
4  END             remove END
Code length: 2. End address: 2281(hex)
1 Errors

```

Figure 38

Note the ORG 2280 command and the RET instruction.

Assemble the main program first. If it assembles at what appears to be an incorrect address, select **IP = Default** from the main menu and reassemble. Finally assemble the subroutine. After this process, the starting address for the memory dump shown on the screen will have changed to the subroutine ORG

address. To return to the default location just select **IP = Default** from the menu.

After the assembly process your screen will look similar to mine, but with different addresses. See figure 39.

[illegible]

Figure 39

The hex numbers 48 and C3 are the codes for DEC AX and RET.

Run the program from the main program's starting address.

Although we moved 5 into the AX register during the main program, the DEC (decrement) instruction in the subroutine subtracted 1 from AX before returning.

DEC and INC

Using these two mnemonics as they stand, we can decrement or increment AX, BX, CX or DX.

As an illustration of the use of the DEC instruction, enter, assemble and run the following code:

1	ORG ?	
2	#MOV AX,FFFF	Move FFFF into AX
3	*LOOP	Address label
4	DEC AX	Decrement the AX register
5	JNZ*LOOP	If AX is not zero, jump back to LOOP
6	RETf	Return to caller
6	END	

The assembled code is shown in figure 40

After running this code the result in AX will be zero.

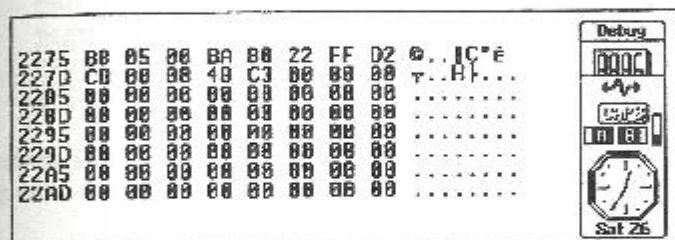


Figure 40

Note that the name chosen for the label can be any four character combination.

CONDITIONAL JUMPS

The JNZ instruction used in the previous program is what is known as a conditional jump. These instructions cause the program to jump to a different address if the condition is met. Appendix 2 describes them all in more detail.

DELAYS

The conditional jump instructions together with the decrement instruction can be used to generate delays within a program.

The length of any delay will be the time taken by the processor to pass around the loop. This can be calculated by multiplying the time to execute each instruction by the number of times each instruction is executed.

The following program will give a delay of about three seconds:

```
1  ORG ?
2  #MOV AX,0010
3  *LOP2
4  #MOV BX,FFFF
5  *LOOP
6  DEC BX
7  JNZ*LOOP
8  DEC AX
9  JNZ*LOP2
10 RETF
11 END
```

The program illustrates the use of what are known as "nested-loops". This is a loop inside a second loop.

If you experiment with this program by altering the contents of AX and BX, take care that you don't make the value in AX too large. The number of times the program will repeat will be the value in AX times that in BX.

DELAY TIMES

Every 8086 instruction takes a certain number of machine cycles to execute. It will take about 25 machine cycles to run once through the above instruction

list. If you multiply this number with the number of times around the loop ($\text{FFFF} \times 0010 = 1,048,576$ decimal) together with the reciprocal of the 3a's clock speed of 7.68 Mhz you get:

$$(1,048,576) \times (1/7.68 \text{ Mhz}) \times 25 = 3.41 \text{ seconds}$$

A delay of about one minute would require a value of 140 hex in AX:

```

1  ORG ?
2  #MOV AX,0140
3  *LOP2
4  #MOV BX,FFFF
5  *LOOP
6  DEC BX
7  JNZ*LOOP
8  DEC AX
9  JNZ*LOP2
10 RETF
11 END

```

Note that I haven't personally confirmed the above figures! If you enter too large a figure in AX, for example FFFF, you could be waiting around for:

$$(65535 \times 65535) \times (1/7.68 \text{ Mhz}) \times 25 = 3.9 \text{ hours! Give or take 20 minutes or so.}$$

FURTHER PROGRAMS

The following program takes a number stored in M1, squares it and stores the answer in a M2.

Program name: SQUARES

1	ORG ?	Org address = 228D (in my case)
2	#MOVAX,0005	Number to square into AX
3	>MOVAX,[MEM1]	Store number in M1

4	MOV BX,AX	Copy to BX and CX
5	MOV CX,AX	
6	XOR AX,AX	Clear AX
7	>MOVAX,[MEM2]	Clear M2
8	CLC	Clear carry flag
9	*LOOP	Address label
10	ADD AX,BX	AX = AX + BX
11	DEC CX	CX = CX - 1
12	JNZ*LOOP	Jump back to loop if CX <> 0
13	>MOVAX,[MEM2]	Store answer in M2
14	RETF	Far return
15	END	

Code length: 25

Symbol table: LOOP ... 229D

Compare the syntax of the above program written in our assembler to the same program using the assembler option in MSDOS Debug on a PC:

2	MOVAX,0005	Number to square into AX
3	MOV [MEM1],AX	Store number in M1
4	MOV BX,AX	Copy to BX and CX
5	MOV CX,AX	
6	XOR AX,AX	Clear AX
7	MOV [MEM2],AX	Clear M2
8	CLC	Clear carry flag
9	*LOOP	Address label
10	ADD AX,BX	AX = AX + BX
11	DEC CX	CX = CX - 1
12	JNZ*LOOP	Jump back to loop if CX <> 0
13	MOV [MEM2],AX	Store answer in M2
14	RETF	Far return
15	END	

Notice the use of the XOR instruction to clear the AX register. This being a two byte instruction rather than a three byte one, is quicker than MOV AX,0000.

When ever the contents of AX, or indeed any 2 byte register, is transferred to a memory location, AL goes to the First address and AH to the Second:

Assuming AX contains 1234, then AH = 12 and AL = 34.

Address	Before MOV [2292],AX	After MOV [2292],AX
2292	00	34
2293	00	12

If you still feel unsure about this, run the following code and examine the contents of address 3010 and 3011:

Program name: HILO

```

1  ORG 3000
2  #MOV AX,1234
3  >MOV AX,[3010]
4  RETF
5  END

```

The following program loops around 100 times. The count is stored in the AX register.

Program name: LOOP

1	ORG ?	Org address = default
2	#MOV AX,0001	Move 1 into AX
3	#MOV BX,0101	Move 101 hex into BX

4	*LOOP	Address label
5	INC AX	Increment AX
6	CMP AX,BX	Compare AX and BX
7	JNZ*LOOP	If AX<>BX jump back to label loop
8	RETf	Far return
9	END	

Code length: 12

Symbol table: LOOP ... 227B

STORE STRING AND REPEAT INSTRUCTION

This two instructions together provide a powerful combination.

The STOS instruction transfers a byte or word from register AX or AL to the string element addressed by DI, and then updates DI to point to the next location of the string.

The REP instruction can be used with the STOS instruction which is repeated by the value in CX. If CX = 0, then the STOS instruction is not performed.

Let's consider an example:

ORG ?	Default ORG address
#MOV CX,0010	Move hex 10 (16) in to CX
#MOV DI,3000	Pointer (3000) into DI
#MOV AX,5555	5555 into AX
REPZ	Repeat the next instruction if CX>0
STOSW	Store AX in [DI] and the increment DI
RETf	Far return

If you run this program, you will notice 32 bytes from address 3000 are filled with 5555. The count in CX is 10 hex, which is 16 decimal. Why are there 32 bytes of 55 and not 16?

As an exercise, starting at address 3000, fill 8 words (= 16 bytes) with 1234, followed by 8 words with 5678 as below:

```
3000 12 34 12 34 12 34 12 34
3008 12 34 12 34 12 34 12 34
3010 56 78 56 78 56 78 56 78
3018 00 00 00 00 00 00 00 00
```

Chapter 8 - The Psion 3 Operating System

The OS

The purpose of any operating system is to provide an interface between the user and the hardware. Updating the LCD display and scanning the keyboard are just two of over 500 services that the Psion OS provides. With the assembler as it stands it is not possible to access the OS ROM directly or display its memory contents. But there are considerable advantages in using some of the facilities available as well as providing us with considerable programming experience.

Psion have succeeded in providing us with a superb OS considering the size of the machine. In many ways it is more reliable than some desk top PC's, and is certainly friendlier. This unfortunately makes accessing the system directly extremely difficult. Luckily for us however, Psion has given us an fairly easy route into the system via what are called SOFTWARE INTERRUPTS.

Appendix 7 list some of the easier to use OS calls. I will not attempt to provide examples of the use of all of these, as most of them require extensive knowledge of the operating system. By all means experiment, but be prepared for more then the occasional complete system crash. As an added precaution, it would be wise to remove any Flash SSDs before diving in. As a final warning, I can only vouch for the examples listed. And even these, under certain circumstances, could cause problems.

ACCESSING THE OPERATING SYSTEM

Accessing the services provided by the Psion OS is via the 8086 software interrupt instruction INT. There are two types of interrupt services:

1. Single Service
2. Multi Service

SINGLE SERVICE

These interrupts are the easier to use, and are programmed as follows:

INT XX

where XX is the interrupt number in hex, e.g. INT A1.

MULTI SERVICE

Before multi service interrupts can be called, the AH register needs to be loaded with a service function number:

```
MOV AH, YY  
INT XX
```

where XX is the interrupt number and YY is the function number, both in hex.

For example the following program will switch off the Psion for about 4 seconds:

```
1  ORG ?  
2  #MOV CX,000F  
3  #MOV AH,14  
4  .INT 8E  
5  RETF
```

Note the syntax of the INT instruction. The full stop must always be included so that our assembler can recognise this instruction.

CALLING CONVENTIONS

The following applies to Multi Service and Single Service interrupt calls.

- 1 Except for AX, all registers are preserved, unless they contain returned values.
- 2 The carry flag will be set if an error occurs. Not all services return errors.
- 3 Register AL will indicate the error and registers containing returned values will be indeterminate.
- 4 Under normal circumstances, processes will execute with the DS, SS and ES registers all pointing to the same segment.
- 5 Valid handles are 16 bit, positive, non-zero and even in value.
- 6 Handles are passed in the BX register wherever possible and return values are always in the AX register.
- 7 Where a 32 bit value is required in a register pair the register pair will be shown as XX:YY, where XX is the most significant word and YY the least significant word.

EXAMPLE PROGRAMS USING INTERRUPTS

PROGRAM NAME: "SWOFF"

OPERATION: This program will switch off the Psion for about 4 seconds.

```
1  ORG ?
2  #MOV CX,000F
3  #MOV AH,14
4  .INT 8E
5  RETF
```

COMMENTS: The value in CX determines how long, in quarters of a second, the machine switches off for. If the value in CX is less than or equal to 8 this program will do nothing.

PROGRAM NAME: "RAMSIZE"

OPERATION: Returns the size of the system RAM.

```
1  ORG ?
2  #MOV AH,0D
4  .INT 8B
5  RETF
```

COMMENTS: This program returns the size of the RAM fitted to the machine in paragraphs. A paragraph is 16 bytes, so converting this figure to decimal and multiplying by 16 will give you either 524,288 or 262,144 depending on the size of your machine.

PROGRAM NAME: "LCD"

OPERATION: This program will cycle the LCD contrast. Press the Psion and escape key to exit this program.

```
1  ORG ?
2  *LOOP
3  #MOV BX,FFFF
4  *DLAY
5  DEC BX
6  JNZ*DLAY
7  #MOV AX,1201
8  .INT 8E
9  JNZ*LOOP
10 RETF
```

COMMENTS: The function number, 12 is in AH and AL contains a positive number to step the contrast up or a negative number to step the contrast down. Try replacing #MOV AX,1201 with #MOV AX,12FF to see the effect of the step down. For a faster cycle, reduce the value in the BX register.

PROGRAM NAME: "PIEZO"

OPERATION: This program will make a sound through the piezo buzzer.

```
1  ORG ?
2  #MOV BX,0012
3  #MOV AH,10
4  .INT 8B
5  *LOOP
6  #MOV AH,11
7  #MOV BX,000F
8  #MOV CX,0200
9  .INT 8B
10 #MOV AH,11
11 #MOV BX,000F
12 #MOV CX,0100
13 .INT 8B
14 JNZ*LOOP
15 RETF
```

COMMENTS: This program will make a two tone sound through the piezo buzzer. The sound flags must be set before this function can be called and lines 2 to 4 carry out this function. See the appendix for full details of our to set the sound flags, but in this example, the value of 12 in the BX register will give the loudest sound. With function 11, which makes the actual sound, BX contains the duration of the sound in ticks and CX holds the pitch of the note. This can be calculated as $(512/CX)$ kHz.

PROGRAM NAME: "MAKEDIR"

OPERATION: This program will make a new directory.

```

1  ORG ?
2  #MOV AH,0C
3  #MOV BX,(ADDR)
4  .INT 87
5  RETF

```

COMMENTS: The pointer to the new directory name must be in the BX register. In the example below, the new directory name is located at address 1040. Use the Enter text option from the menu. Enter as \NEWDIR\ where NEWDIR is the chosen directory name.

1	ORG 1000	Start address: 4056(dec)	
2	#MOV AX,0C00	00000C	1000
3	#MOV BX,1040	004010	1003
4	.INT 87	0087	1006
5	RETF	CB	1008
6	END		

Code length: 9. End address: 1008(hex)

1 Errors

PROGRAM NAME: "GETOFF"

OPERATION: This program retrieves the auto switch off time in seconds. The time will be in HEX.

```
1  ORG ?
2  #MOV AH,17
4  .INT 8B
5  RETF
```

COMMENTS: By default the, the auto-switch off time is set to 300 seconds. If the value returned in AX is -1, the auto switch off facility is disabled.

PROGRAM NAME: "SETOFF"

OPERATION: This program sets the auto switch off time in seconds.

```
1  ORG ?
2  #MOV AH,18
3  #MOV BX,0200
4  .INT 8B
5  RETF
```

COMMENTS: The time, in seconds, is passed in the BX register. Entering -1 here will disable the auto switch off function. In the above example, 200 hex = about 8.5 minutes.

PROGRAM NAME: "LANGUAGE"

OPERATION: This program simply returns the language code of your machine.

```
1  ORG ?
2  #MOV AH,1B
3  .INT 8B
4  RETF
```

COMMENTS: Some of the codes returned are, 1 = English, 2 = French, 3 = German.

PROGRAM NAME: "COPY"

OPERATION: This program will copy areas of memory.

```
1  ORG ?
2  #MOV SI,1040
3  #MOV DI,3000
4  #MOV CX,0010
5  .INT A1
6  RETF
```

COMMENTS: Register SI contains the pointer to the source buffer and DI contains the pointer to the destination buffer. The byte length is held in the CX register. Be very careful with your choice of addresses. The addresses in the example given are not particularly good. You can use this function to clear an area of "safe" memory or copy your code somewhere else.

PROGRAM NAME: "DECBIN"

OPERATION: This program will change the number in BX to binary, octal, decimal or hexadecimal and store the result in a buffer.

```
1  ORG ?
2  #MOV BX,0007
3  #MOV CX,0002
5  #MOV DI,3000
4  #MOV AH,00
8  .INT 8A
5  RETF
```

COMMENTS: The unsigned integer is in the BX register. DI contains a pointer to the buffer to receive the converted number. The number of characters written to the buffer is returned in the AX register. The conversion required must be loaded into the CX register prior to using this function. CX = 2 for binary, CX = 8 for octal, CX = 0A for decimal and CX = 10 for hexadecimal.

PROGRAM NAME: "CHARUP"

OPERATION: This program converts the characters in AH and AL to upper case.

```
1  ORG ?
2  #MOV AX,6162
3  .INT 9E
4  RETF
```

COMMENTS: In the example shown, 61 and 62 are the ASCII codes for "a" and "b". After running the program AX will return with AH = 41 and AL = 42 ("A" and "B").

PROGRAM NAME: "SEARCH"

OPERATION: This program will locate a character in a buffer.

```
1  ORG ?
2  #MOV SI,1000
3  #MOV CX,8000
4  #MOV AH,46
5  .INT A7
6  RETF
```

COMMENTS: Register SI contains a pointer to the buffer to be searched. AH holds the character to be searched for and CX contains the length of the buffer. In the example above, we are searching for the character "46" in a buffer starting at address 1000 and terminating at address 9000.

PROGRAM NAME: "DAYNAME"

OPERATION: This program returns the name of the day corresponding to the day of the week in AL.

```
1  ORG ?
2  #MOV BX,3000
3  #MOV AL,03
4  #MOV AH,0A
5  .INT 89
6  RETF
```

COMMENTS: Register AL contains the day number, 0 to 6. The pointer to a buffer to hold the result is contained in BX. Be careful with using 3000 as the buffer address. Choose your own value here.

PROGRAM NAME: "MTHNAME"

OPERATION: This program returns the name of the month corresponding to the month number in AL.

```
1  ORG ?
2  #MOV BX,3000
3  #MOV AL,03
4  #MOV AH,0B
5  .INT 89
6  RETF
```

COMMENTS: Register AL contains the month number, 0 to 11. The pointer to a buffer to hold the result is contained in BX. Be careful with using 3000 as the buffer address. Choose your own value here.

PROGRAM NAME: "WEEKNUM"

OPERATION: This program returns the week number corresponding to the number of days since 1st Jan 1900.

```
1  ORG ?
2  #MOV CX,0000
3  #MOV DX,0100
4  #MOV AH,0D
5  .INT 89
6  RETF
```

COMMENTS: Put the number of days in register CX and DX. The above is for 100 hex days, therefore 0000 will be in register CX and 0100 in register DX. Register AX returns with the week number (1 - 53).

PROGRAM NAME: "DAYABB"

OPERATION: This program returns with the abbreviated name of the day. E.g. day 2 (Wednesday) will be returned as Wed.

```
1  ORG ?
2  #MOV BX,3000
3  #MOV AL,02
4  #MOV AH,0E
5  .INT 89
6  RETF
```

COMMENTS: Register AL contains the day number, 0 to 6. The pointer to a buffer to hold the result is contained in BX. Be careful with using 3000 as the buffer address. Choose your own value here.

PROGRAM NAME: "MTHABB"

OPERATION: This program returns with the abbreviated name of the month. E.g. month 2 (March) will be returned as Mar.

```
1  ORG ?
2  #MOV BX,3000
3  #MOV AL,02
4  #MOV AH,0F
5  .INT 89
6  RETF
```

COMMENTS: Register AL contains the month number, 0 to 11. The pointer to a buffer to hold the result is contained in BX. Be careful with using 3000 as the buffer address. Choose your own value here.

PROGRAM NAME: "TIME"

OPERATION: This program returns with the number of seconds since 1st Jan 1970 at 00:00:00. AX contains the most significant word and BX contains the least significant word.

```
1  ORG 2295
2  #MOV AH,02
3  .INT 89
4  >MOVAX,[MEM1]
5  MOV AX,BX
6  >MOVAX,[MEM2]
7  RETF
```

COMMENTS: In the above example, AX and BX have been stored in memory locations M1 AND M2. The following results were obtained on my machine, 2ED7 and 8BFB. This (in HEX) is the total number of seconds from 1st Jan 1970 upto when the program was executed.

PROGRAM NAME: "SCANKEY"

OPERATION: This program scans the state of all the keys.

```
1  ORG ?
2  #MOV BX,3000
3  #MOV AH,28
4  .INT 8E
5  RETF
```

COMMENTS: The BX register contains a pointer to a ten word block of memory to take the scan codes.

The following diagram indicates the scan codes associated with each key. The number before the decimal point is the memory offset times two from the start of the memory block, and the number after the decimal point will be in the relevant location after the key has been pressed.

4.200	3.200	5.200	1.200	0.300	2.200	1.100	0.100																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																										</
-------	-------	-------	-------	-------	-------	-------	-------	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----

PROGRAM NAME: "SUBTEST"

OPERATION: This program cycles the LCD display. Each time through the main loop it calls the subroutine at address 3000. This subroutine calls the scan key interrupt service routine and stores the keys pressed in the array at 4000. If the main program detects that the TAB key has been pressed, it terminates. It may be better to assemble the subroutine first followed by the main program.

Main Program

ORG 2F00

*LOOP

#MOV DX,3000

CALL DX

#MOV BX,0004

```
CMP AX,BX
JZ *FINI
#MOV BX,FFFF
*DLAY
DEC BX
JNZ*DLAY
#MOV AX,1201
.INT 8E
JNZ*LOOP
*FINI
RETF
END
```

Subroutine

```
ORG 3000
#MOV BX,4000
#MOVAH,28
.INT 8E
<MOVAX,[4000]
RET
END
```

CONCLUSION

In this slim volume, it has not been possible to cover all that the Psion 3 has to offer the machine code programmer. Even with the limited instruction set which our assembler can accommodate, the possibilities for program experimentation are vast. Also there remains the 8086 instructions not included in the Mnemonic.Dbf file. These are waiting to be entered and tried out.

As you gain more and more experience in using the various 8086 instructions and software interrupts, you will discover new and creative ways to write applications.

So good luck and happy programming!

Appendix 1 - The V30 Microprocessor

NEC V20/30 SERIES

The NEC V series 16-bit microprocessors are enhanced devices based on the 8086/8088 type architecture. The V20 and V30 are directly pin compatible replacements for the Intel 8088 and 8086 respectively and are widely used in IBM compatible personal computer clones. By using improved address generation and pipelining of instructions the V20/V30 devices provide some 40 per cent greater execution speed compared with an 8088/8086 using the same clock frequency. The instruction set is compatible with that of the 8088/8086 but has been enhanced by the addition of a number of new instructions.

The V40 and V50 versions include on chip clock generation, three counter/timers, an interrupt priority controller and four DMA channels. These devices provide similar facilities to the Intel 80188 and 80186 respectively.

Devices available

- PD70108 (V20) 8-bit data bus version
- PD70116 (V30): 16-bit data bus version
- PD70208 (V40) V20 with on chip peripherals
- PD70216 (V50) V30 with on chip peripherals
- PD70136 (V33) Fast V30 with extended address bus.

Architecture

The architecture of the V30 is an enhanced version of the 8086 architecture. The main difference is in the addition of pipelining and other internal logic improvements to improve the effective execution speed over that of an 8086 running with the same clock frequency.

Package

V20/V30 40 pin DIL, 52-pin flat pack or 44-pin PLCC

V33	68-pin PGA
V40/V50	68-pin PLCC, 68-pin PGA or 80-pin flat pack.
V25/V35	84-pin PLCC or 94-pin flat pack.

Pin connections

For the V20 and V30 the pin connections are directly compatible with the Intel 8088 and 8086 respectively.

Power requirements

$V_{cc} = +4.75 \text{ V to } +5.25 \text{ V}$

Interrupt facilities

The V20/V30 processors can handle 256 different exceptions and the vectors for these are held in a vector table at the bottom of memory. Each vector takes up 4 bytes in memory so the complete vector table occupies the bottom 1 kbytes of memory. The vectors are allocated as follows:

Vector	Function
0	Divide error
1	Debug exception
2	Non-masked interrupt NMI
3	One byte interrupt INT
4	Interrupt on overflow INTO
5	Array bounds check BOUND
6	Invalid opcode

Vector	Function
7	Device not available
8	Double fault
9	Coprocessor segment overrun
10	Invalid TSS
11	Segment not present
12	Stack fault
13	General protection fault
14	Page fault
15	Reserved
16	Coprocessor error
17-32	Reserved
33-255	INT n trap instructions

Instruction set

The instruction set of the V series processors is basically the same as that of the 8086/88 type processors except that a few new instructions have been added.

The basic instructions provided on the V20/V30 are as follows:

AAA	ASCII adjust for add
AAD	ASCII adjust for divide
AAM	ASCII adjust for multiply
AAS	ASCII adjust for subtract
ADC	Add with carry
ADD	Add
AND	Logical AND
BSF	Scan bit forward
BSR	Scan bit reverse
BT	Test bit
BTC	Test bit and complement
BTR	Test bit and reset

BTS	Test bit and set
CALL	Call subroutine
CBW	Convert byte to word
CLC	Clear carry flag
CLO	Clear direction flag
CLI	Clear interrupt enable flag
CLTS	Clear task switched flag
CMC	Complement carry flag
CMP	Compare
CMPS	Compare string byte/word
CWD	Convert word to double word
DAA	Decimal adjust for add
DAS	Decimal adjust for subtract
DEC	Decrement
DIV	Unsigned divide
ENTER	Enter procedure
HLT	Halt
IDIV	Signed integer divide
IMUL	Signed integer multiply
IN	Input from port
INC	Increment
INS	Input string byte/word from DX port
INT	Interrupt
INTO	Interrupt 4 if overflow set
IRET	Interrupt return
JB/JNAE	Jump on below/not above or equal
JBE/JNA	Jump on below or equal/not above.
JCXZ	Jump on CX zero
JE/JZ	Jump on equal/zero
JECXZ	Jump on ECX zero
JL/JNGE	Jump on less/not greater or equal
JLE/JNG	Jump on less or equal/not greater
JMP	Unconditional jump
JNB/JAE	Jump on not below/above or equal
JNBE/JA	Jump on not below or equal/above
JNE/JNZ	Jump on not equal/not zero

JNL/JGE	Jump on not less/greater or equal
JNLE/JG	Jump not less or equal/greater
JNO	Jump on not overflow
JNP/JPO	Jump on not parity/parity odd
JNS	Jump on not sign
JO	Jump on overflow
JP/JPE	Jump on parity/parity even
JS	Jump on sign
LAHF	Load AH into flag register
LDS	Load pointer to DS
LEA	Load effective address
LEAVE	Leave procedure
LES	Load pointer to ES
LFS	Load pointer to FS
LGS	Load pointer to GS
LODS	Load string byte/word to AL/AX/EAX
LOOP	Loop CX times
LOOPZ/LOOPE	Loop while zero/equal
LOOPNZ/LOOPNE	Loop while not zero/not equal
LSS	Load pointer to SS
MOV	Move data
MOVS	Move string byte/word
MOVSB	Move with sign extension
MOVZX	Move with zero extension
MUL	Unsigned multiply
NEG	Negate
NOP	No operation
NOT	Logical complement
OR	Logical OR
OUT	Output to port
OUTS	Output string byte/word to DX port
POP	Pop data from stack
POPA	Pop all
POPF	Pop flags
PUSH	Push to stack
PUSHA	Push all

PUSHF	Push Flags
RCL	Rotate left via carry
RCR	Rotate right via carry
REPE CMPS	Repeated compare string (find no match)
REPNE CMPS	Repeated compare string (find match)
REP INS	Repeated input string
REP LODS	Repeated load string
REP MOVS	Repeated move string
REP OUTS	Repeated output string
REPE SCAS	Repeated scan string
REPNE SCAS	Repeated scan string
REP STOS	Repeated store string
RET	Return from call
ROL	Rotate left
ROR	Rotate right
SAHF	Store flags to AH
SAL	Arithmetic shift left
SAR	Arithmetic shift right
SBB	Subtract with borrow
SCAS	Scan string byte/word
SETB/SETNAE	Set byte on below/not above or equal
SETBE/SETNA	Set byte on below or equal/not above
SETE/SETZ	Set byte on equal/zero
SETL/SETNGE	Set byte on less/not greater or equal
SETLE/SETNG	Set byte on less or equal/not greater
SETNB	Set byte on not below
SETNBE/SEA	Set byte on not below or equal/above
SETNE/SETNZ	Set byte on not equal/not zero
SETNL/SETGE	Set byte on not less/greater or equal
SETNLE/SETG	Set byte on not less or equal/greater
SETNO	Set byte on not overflow
SETNP/SETPO	Set byte on not parity/parity odd
SETNS	Set byte on not sign
SETO	Set byte on overflow
SETP/SETPE	Set byte on parity/parity even
SETS	Set byte on sign

SHL	Shift left
SHLD	Shift left double word
SHR	Shift right
SHRD	Shift right double word
STC	Set carry flag
STD	Set direction flag
STI	Set interrupt enable flag
STOS	Store string byte/word from AL/AX/EAX
SUB	Subtract
TEST	Test data
WAIT	Wait until busy pin is negated
XCHG	Exchange data
XLAT	Translate string
XOR	Logical EXCLUSIVE OR

New instructions available on the V series processors are:

CLR1	Clear a bit
NOT1	Complement a bit
SET1	Set a bit
TEST1	Test a bit
INS	Insert bit field
EXT	Extract bit field
ADD4S	Add BCD string
CMP4S	Compare BCD string
SUB4S	Subtract BCD string
ROL4S	Rotate left BCD string
ROR4S	Rotate right BCD string
POP R	Pop all registers
PUSH R	Push all registers
PUSH	Push immediate data
CHKIND	Check index
INM	Input string
OUTM	Output string

Appendix 1 - The V30 Microprocessor

REPC	Repeat if carry set
REPNC	Repeat if carry clear

There are also PREPARE and DISPOSE instructions for stack frame maintenance. A new immediate addressing mode has been added for the instructions MUL, SHR, SHL, ROR, ROL, ROLC, RORC and SHRA.

Timing

The bus cycle sequence for these processors is the same as that for the Intel 8086 type.

The V20 and V30 are available in 5 MHz, 8 MHz and 10 MHz versions. The V40 and V50 types are available in 8MHz and 10MHz versions.

Appendix 2 - 8086 Instruction Set

AAA ASCII Adjust for addition

Syntax AAA

Function If D3, D2, D1, D0 of AL > 9 or AF flag = 1, then AL = AL + 6, AH = +1, AF = 1, CF = AF, AL = AL & 0F.

Flags OF DF IF TF SF ZF AF PF CF
 * *

Description This instruction changes the value of the AL register to a valid unpacked BCD number. D7-D4 of the AL register are set to zero.

Example AAA

AAD Adjust for division

Syntax AAD

Function $AL = AH * 10 + AL$, AH = 0

Flags OF DF IF TF SF ZF AF PF CF
 * *

Description This instruction converts a two-byte BCD number in AX to an equivalent binary number in AL. AH is set to zero.

Example AAD

AAM ASCII Adjust for multiply

Syntax AAM

Function $AH = AL / 0Ah$, $AL = \text{remainder}$

Flags OF DF IF TF SF* ZF* AF PF* CF

Description This instruction converts an 8-bit number less than 100, in AL, into a two-byte unpacked BCD number in AX.

Example AAM

AAS Adjust for subtraction

Syntax AAS

Function If $D3, D2, D1, D0$ of $AL > 9$ or $AF \text{ flag} = 1$, then $AL = AL - 6$, $AH = -1$, $AF = 1$, $CF = AF$, $AL = AL \& 0F$.

Flags OF DF IF TF SF* ZF* AF PF* CF

Description This instruction corrects the result of a previous subtraction of two unpacked decimal numbers. AL changes to a valid unpacked decimal number. The high-nibble of AL is set to zero.

Example AAS

ADC Add with carry

Syntax ADC destination, source

Function Destination = destination + source + CF

Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
	*				*	*	*	*	*

Description This instruction sums the byte or word operand, and adds 1 if the carry flag is set.

Example ADC AX,BX

ADD Addition

Syntax ADD destination, source

Function Destination = destination + source

Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
	*				*	*	*	*	*

Description This instruction adds the destination and source operations. The result is placed in the destination operand.

Example ADD DX,CX

AND Logical And

Syntax AND destination, source

Function Destination = destination AND source

Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
	0				*	*		*	0

Description This instruction performs the logical AND of the two byte or word operands. The result is in the destination operand.

Example AND AX,BX

CALL Call procedure

Syntax CALL destination

Function push (CS), push (IP), CS:IP = destination (Far call)
 push (IP), IP = destination (Near call)

Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
--------------	----	----	----	----	----	----	----	----	----

Description This instruction transfers control to the address specified in the operand. Called procedure must terminate with RETF or RET instruction.

Example CALL AX

CBW Convert byte to word

Syntax CBW

Function	If AL < 80h then AH = 00 If AL > 80h then AH = FF
-----------------	--

Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
-------	----	----	----	----	----	----	----	----	----

Description: This instruction converts a signed byte in AL into a signed word in AX.

Example CBW

CLC Clear carry flag

Syntax CLC

Function CF - 0

[illegible]

Description This instruction clears the Carry flag

Example **CLC**

CLD Clear Direction Flag

Syntax CLD

Function DF = 0

Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
		0							

Description This instruction sets the direction flag to zero, causing the string instruction to autoincrement the SI and DI index registers.

Example CLD

CLI Clear Interrupt Enable Flag

Syntax CLI

Function IF = 0

Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
			0						

Description This instruction clears the interrupt flag

Example CLI

CMC Complement the Carry Flag

Syntax CMC

Function If $CF = 0$ then $CF = 1$. If $CF = 1$ then $CF = 0$.

Flags OF DF IF TF SF ZF AF PF CF
*

Description This instruction toggles the carry flag to the opposite state.

Example CMC

CMP Compare

Syntax CMP destination, source

Function Destination - source.

Flags OF DF IF TF SF ZF AF PF CF
* * * * *

Description This instruction subtracts the source from the destination and sets the appropriate flags. It does not return the result.

Example CMP

CMPS Compare strings

Syntax CMPS destination_string, source_string.

Function If DF = 0 then SI = SI + operand size, DI = DI + operand size
 If DF = 1 then SI = SI - operand size, DI = DI - operand size

Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
	*				*	*	*	*	*

Description This instruction subtracts the destination byte or word (addressed by DI) from the source byte or word (addressed by SI). DI and SI are altered according to the size of the operands and the status of DF. This instruction is generally used with the repeat instructions.

Example CMPS source,destination

CWD Convert Word to Double Word

Syntax CWD

Function If AX < 8000h then DX = 0
 If AX > 8000h then DX = FFFFh

Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
--------------	----	----	----	----	----	----	----	----	----

Description This instruction converts the signed word in AX to a signed doubleword in DX:AX.

Example CWD

DAA Decimal Adjust for Addition**Syntax** DAA**Function** If D3, D2, D1, D0 of AL > 9 or AF = 1, then AL = AL + 6; AF = 1
If AL > 9Fh or CF = 1, then AL = AL + 60h; CF = 1**Flags** OF DF IF TF SF ZF AF PF CF
* * * * ***Description** This instruction converts the content of AL to a pair of valid packed decimal number.**Example** DAA**DAS Decimal Adjust for Subtraction****Syntax** DAS**Function** If D3, D2, D1, D0 of AL > 9 or AF = 1, then AL = AL - 6; AF = 1
If AL > 9Fh or CF = 1, then AL = AL - 60h; CF = 1**Flags** OF DF IF TF SF ZF AF PF CF
* * * * ***Description** This instruction changes the content of AL to a pair of valid packed decimal numbers which resulted from a previous subtraction.**Example** DAS

DEC Decrement

Syntax DEC operand

Function Operand = operand - 1

Flags OF DF IF TF SF ZF AF PF CF
 * * * * * * * *

Description This instruction subtracts 1 from the operand.

Example DEC

DIV Divide

Syntax DIV source

Function If operand = byte, then AX/source, AH = quotient, AL = remainder.
 If operand = word, then DX:AX/source, AX = quotient, DX = remainder.

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction performs an unsigned division of unsigned integers contained in AX or DX:AX.

Example DIV AX,25

ESC Escape

Syntax ESC external opcode, address

Function Place instruction on the data bus

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction is used with co-processors and should not be executed on the Psion 3.

Example ESC 7,AL

HLT Halt

Syntax HLT

Function None

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction stops the CPU until a reset or NMI occurs.

Example HALT

IDIV Integer Division

Syntax IDIV source

Function If source = byte, then AL = quotient, AH = remainder.
If source = word, then AX = quotient, DX = remainder.

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction performs a division of the byte in AX or word in DX:AX by the operand.

Example IDIV CL

IMUL Integer Multiplication

Syntax IMUL source

Function If source = byte, then AX = AL x signed source
If source = word, then DX = AX x signed source.

Flags OF DF IF TF SF ZF AF PF CF
* * * *

Description This instruction performs an unsigned multiplication of unsigned integers contained in AX or DX:AX.

Example IMUL BL

IN Input Byte from I/O Port

Syntax IN Accumulator, port_address
 IN accumulator, DX

Function If byte, then AL = data from port number or from address in DX
 If word, then AX = data from port number or from address in DX.

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction moves a byte or word from an input port to AL or AX. The port number can be between 0 - 255 or 0 - 65535 in DX.

Example IN AL, 5A

INC Increment

Syntax INC destination

Function Destination = destination + 1

Flags OF DF IF TF SF ZF AF PF CF
 + * * + *

Description This instruction adds 1 to the destination operand.

Example INC AX

INT Interrupt

Syntax INT int_number

Function SP = SP - 2; Flags pushed; IF = 0; TF = 0; SP = 0; CS pushed; SP = SP - 2; IP pushed;
CS = data at memory address (type * 4 + 2)
IP = data at memory address (type * 4)

Flags OF DF IF TF SF ZF AF PF CF
0 0

Description This instruction generates a software interrupt specified by the int number which must be 0 - 255.

Example INT 32

INTO Interrupt on Overflow

Syntax INTO

Function If OF = 1, then SP = SP - 2; Flags pushed; IF = 0; TF = 0;; CS pushed; SP = SP - 2; CS pushed;
CS = data at memory address 12h
SP = SP - 2; IP pushed; new IP = data at memory address 10h

Flags OF DF IF TF SF ZF AF PF CF
* *

Description This instruction generates a software interrupt if the overflow flag is set. If OF = 0 then control proceeds to the following instruction without activating the interrupt call. **Example** INTO

IRET Interrupt Return**Syntax** IRET**Function** IP popped from stack; $SP = SP + 2$; CS popped from stack; $SP = SP + 2$;
Flags popped from stack; $SP = SP + 2$.**Flags** OF DF IF TF SF ZF AF PF CF
 * * * * * * * * ***Description** This instruction transfers control back to the instruction after which the interrupt occurred.**Example** IRET**JA/JNBE Jump if Above/ Jump if Not Below or Equal****Syntax** JA/JNBE label**Function** If CF and $ZF = 0$, then $IP = IP + 8\text{-bit displacement} (-128 \text{ to } +127)$.**Flags** OF DF IF TF SF ZF AF PF CF**Description** These instructions, which generate the same instruction code, transfer control to the target operand.**Example** JNBE label

JAE/ JNB/ JNC Jump if Above or Equal/ Not Below/ No Carry

Syntax JAE/ JNB/ JNC label

Function If $CF = 0$, then $IP = IP + 8\text{-bit displacement} (-128 \text{ to } +127)$.

Flags OF DF IF TF SF ZF AF PF CF

Description These instructions transfer control to the target operand

Example JAE label

JB/ JNAE/ JC Jump Below/ Not Above or Equal/ Carry

Syntax JB/ JNAE/ JC label

Function If $CF = 1$, then $IP = IP + 8\text{-bit displacement} (-128 \text{ to } +127)$.

Flags OF DF IF TF SF ZF AF PF CF

Description These instructions transfer control to the target operand

Example JNAE label

JBE/ JNA Jump if Below or Equal/ Jump if Not Above

Syntax JBE label

Function If CF or ZF = 1, then IP = IP + 8-bit displacement (-128 to +127).

Flags OF DF IF TF SF ZF AF PF CF

Description These instructions transfer control to the target operand

Example JBE label

JCXZ Jump if CX Register Zero

Syntax JCXZ label

Function If CX = 0, then IP = IP + 8-bit displacement (-128 to +127).

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction transfers control to the target operand

Example JCXZ label

JE/ JZ Jump if Equal/ Jump if Zero

Syntax JE label
 JZ label

Function If ZF = 1, then IP = IP + 8-bit displacement (-128 to +127).

Flags OF DF IF TF SF ZF AF PF CF

Description These instructions transfer control to the target operand

Example JE label

JG/ JNLE Jump if Greater/ Jump if Not Less Than or Equal

Syntax JG label
 JNLE label

Function If SF = OF and ZF = 0, then IP = IP + 8-bit displacement (-128 to +127).

Flags OF DF IF TF SF ZF AF PF CF

Description These instructions transfer control to the target operand

Example JG label

JGE/ JNL Jump if Greater or Equal/ Jump if Not Less Than

Syntax JGE label
 JNL label

Function If SF = OF, then IP = IP + 8-bit displacement (-128 to +127).

Flags OF DF IF TF SF ZF AF PF CF

Description These instructions transfer control to the target operand

Example JGE label

JNE/ JNZ Jump if Not Equal/ Not Zero

Syntax JNE label
 JNZ label

Function If ZF = 0, then IP = IP + 8-bit displacement (-128 to +127).

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction transfers control to the target operand

Example JNE label

JMP Jump Unconditionally

Syntax JMP label

Function IP = IP + signed offset

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction transfers control to the target location.

Example JMP label

LAHF Load Flags into AH

Syntax LAHF

Function	AH ← d7	d6	d5	d4	d3	d2	d1	d0
	SF	ZF	X	AF	X	PF	X	CF

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction transfers the flag contents into AH.

Example LAHF

LDS Load Pointer using DS**Syntax** LDS destination, source**Function** Destination = source
 seg = source + 2**Flags** OF DF IF TF SF ZF AF PF CF

Description This instruction transfers a 32 bit pointer variable from a memory operand to the destination operand and register DS. The offset word of the pointer is transferred to the destination operand, which may be any register. The segment word of the pointer is transferred to register DS.

Example LDS SI, label

LEA Load Effective Address**Syntax** LEA destination, source**Function** destination = address(source)**Flags** OF DF IF TF SF ZF AF PF CF

Description This instruction transfers the offset of the source operand to the destination operand.

Example LEA CX, label

LODS Load String (Byte or Word)

Syntax LODS *source_string*
 LODSB
 LODSW

Function If byte-string, then AL = data at memory addressed by SI
 If DF = 0, then SI = SI + 1
 If DF = 1, then SI = SI - 1
 If word-string, then AX = data at memory addressed by SI
 If DF = 0, then SI = SI + 2
 If DF = 1, then SI = SI - 2

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction transfers the byte, word or double-word from DS:SI into the accumulator.

Example LODS STR

LOOP Loop if CX Not Zero

Syntax LOOP *label*

Function CX = CX - 1; If CX $\neq$ 0 then IP = IP + 8-bit displacement.

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction decrements the CX register. If the result is not zero, control is transferred to the address specified by the operand. (-128 to +127)

Example LOOP label

LOOPE/LOOPZ Loop While Equal/ Loop While Zero

Syntax LOOPE label
LOOPZ label

Function $CX = CX - 1$; If $CX \neq 0$ and $ZF = 1$, then $IP = IP + 8\text{-bit displacement}$.

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction decrements the CX register. If the result is not zero, and $ZF = 1$ control is transferred to the address specified by the operand.
(-128 to +127)

Example LOOPE label

LOOPNE/ LOOPNZ Loop While Not Equal/ Loop While Not Zero

Syntax LOOPNE label
 LOOPNZ label

Function $CX = CX - 1$; If $CX \neq 0$ and $ZF = 0$, then $IP = IP + 8\text{-bit displacement}$.

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction decrements the CX register. If the result is not zero, and $ZF = 0$ control is transferred to the address specified by the operand.
(-128 to -127)

Example LOOPNE label

MOV Move Byte or Word

Syntax MOV destination, source

Function Destination = source

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction copies the source data to the destination operand.

Example MOV BX,AX
 MOV AL,09

NEG Negate (Two's complement)

Syntax NEG operand

Function Destination = 0 - Destination

Flags OF DF IF TF SF ZF AF PF CF
 * * * *

Description This instruction subtracts the operand from 0 and transfers the two's complement into the operand.

Example NEG AX

NOP No Operation

Syntax NOP

Function This instruction has no effect.

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction performs no operation except to delay the CPU by 3 clock cycles.

Example NOP

NOT Logical Not

Syntax NOT destination

Function Destination = one's complement of destination.

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction inverts the bits of the byte or word operand.

Example NOT AX

OR Logical Or

Syntax OR destination, source

Function Destination = Destination OR Source

Flags OF DF IF TF SF ZF AF PF CF
0 * * * *

Description This instruction performs a bitwise OR of the destination and the source and transfers the result to the destination operand.

Example OR AL,CL

OUT Output

Syntax OUT port, accumulator

Function Port = Accumulator

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction transfers the byte in the accumulator to the specified I/O port. In the immediate mode, byte = 0 - 255. For 0 - 65535, port address is in DX.

Example OUT 66,AX

POP Pop Value off Stack

Syntax POP destination

Function Destination = data on top of stack; SP = SP + 2

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction transfers the word at the current top of stack to the destination operand, and then increments the SP by 2.

Example POP AX

POPF Pop Flags**Syntax** POPF**Function** Flag bits are set to the data from the top of the stack; $SP = SP + 2$

Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
	*	*	*	+	*	*	*	*	*

Description This instruction transfers the specific bits from the word at the current top of stack into the flags. SP is then incremented by 2.

Example POPF

PUSH Push Data on Top Of Stack**Syntax** PUSH source**Function** $SP = SP - 2$; Source is pushed onto the top of the stack.

Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
--------------	----	----	----	----	----	----	----	----	----

Description This instruction first decrements the SP by 2, and then transfers a word from the source operand to the top of the stack pointed to by SP.

Example PUSH AX

PUSHF Push Flags

Syntax **PUSHF**

Function SP = SP - 2; The Flag register is pushed onto the top of the stack.

Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
-------	----	----	----	----	----	----	----	----	----

Description This instruction decrements SP by 2 and then transfers all the flags to the word at the top of the stack.

Example PUSHF

RCL Rotate Through Carry Left

Syntax RCL destination, count

Function (temp) = count; if count ≤ 0 , then (rcarry) = CF = high order bit of destination; destination = (destination $\times 2$) + (rcarry); (temp) = temp - 1; repeat function until temp = 0.
OF = If count = 1, then if high-order bit of dest $\neq$ CF, then OF = 1; Else, 0. If count ≤ 1 , then OF is undefined.

Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
	*								*

Description This instruction rotates the bits in the word or byte destination operand left by the number of bits specified in the count operand.

Example RCL AH.1

RCR Rotate Through Carry Right**Syntax** RCR destination, count

Function (temp) = count; if count $\neq$ 0, then (rcarry) = CF = high order bit of destination = destination/2 high-order bit of destination = (rcarry):(temp) = (temp) - 1. If count = 1, then if high-order bit of dest $\neq$ next high-order bit of destination, then OF = 1; Else, OF = 0. If count $\neq$ 1, then OF is undefined.

Flags OF DF IF TF SF ZF AF PF CF
* *

Description This instruction rotates the bits in the word or byte destination operand left by the number of bits specified in the count operand.

Example RCR AH,3

REP/ REPE/ REPNE Repeat/ Repeat While Equal/ Repeat While Zero**Syntax** REP MOVSB destination,source

Flags OF DF IF TF SF ZF AF PF CF

Description These instructions can be used with any string instruction (CMPS, LODS, MOVSB, SCAS, STOS). The instruction is repeated by the value in CX. If CX = 0, then the instruction is not performed.

Example REP MOVSB AX, string

REPNE/ REPNZ Repeat While Not Equal/ Repeat While Not Zero

Syntax REPNE SCAS destination

Flags OF DF IF TF SF ZF AF PF CF

Description These instructions, together with REP, operate identically. ZF must be cleared or the repeat function terminates.

Example REPNZ SCAS destination

RET/ RETF Return Near/ Return Far

Syntax RET
 RETF

Function If within segment, IP is popped off stack; $SP = SP + 2$
 If outside segment, CS is popped off stack; $SP = SP + 2$,
 IP is popped off stack; $SP = SP + 2$

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction transfers control from a subroutine back to the instruction following the CALL that called the subroutine.

Example RETF

ROL Rotate Left

Syntax ROL destination, count

Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
	*								*

Description This instruction rotates the bits in the byte or word destination operand left by the number of bits specified in the count operand. If the count = 1, then it can be specified directly. If the count > 1, then it must be loaded into CL before using this instruction.

Example ROL AL,1

ROR Rotate Right

Syntax ROR destination, count

Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
	*								*

Description This instruction rotates the bits in the byte or word destination operand right by the number of bits specified in the count operand. If count = 1, then it may be specified directly. If count > 1, then it must be loaded into CL before using this instruction.

Example ROR BL,CL

SAHF Store Register AH into flags

Syntax	SAHF								
Function	SF	ZF	X	AF	Z	PF	X	CF	
	d7	d6	d5	d4	d3	d2	d1	d0	
Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
					*	*	*	*	*

Description This instruction transfers bits 7, 6, 4, 2 and 0 from register AH into the appropriate flag registers.

Example SAHF

SAL/SHL Shift Arithmetic Left, Shift Logical Left

Syntax	SAL destination, count SHL destination, count								
Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
	*				*	*		*	*

Description These instructions perform the same operation. The destination byte or word is shifted left by the number of bits specified in the count operand. Zeroes are shifted in from the right. If count = 1 then it may be loaded directly. If count > 1, then the value must be loaded into CL before using this instruction.

Example SAL BL,1

SAR Shift Arithmetic Right**Syntax** SAR destination, count

Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
	*				*	*		*	*

Description This instruction shifts the bits in the byte or word destination operand right by the number of bits specified in the count operand. If the count = 1, then it may be loaded directly. If count > 1 then it must be loaded into CL before using this instruction.

Example SAR AL,1**SBB Subtract With Borrow****Syntax** SBB destination, source

Function If CF = 1, then destination = destination - source - 1
 If CF = 0, then destination = destination - source

Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
	*				*	*	*	*	*

Description This instruction subtracts the source from the destination, then subtracts 1 if CF = 1 (set), and returns the result in the destination operand.

Example SBB AH,BH

SCAS Scan String

Syntax SCAS destination_string

Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
	*				*	*	*	*	*

Description This instruction subtracts the byte or word destination string element addressed by DI from the content of AL or AX and then updates the flags, but does not change the destination string or the accumulator. SCAS also updates DI to point to the next string element. This instruction may be used with REPE, REPZ, REPNE or REPNZ.

Example SCAS

SHR Shift Logical Right

Syntax SHR destination, source

Flags	OF	DF	IF	TF	SF	ZF	AF	PF	CF
	*				*	*		*	*

Description This instruction shifts the bits in the destination operand to the right by the number of bits specified in the count operand. Zeroes are shifted in from the left. If the count = 1, then it may be specified directly. If the count > 1, then it must be loaded into CL before this instruction can be used.

Example SHR AL,1

STC Set Carry Flag**Syntax** STC**Function** CF = 1**Flags** OF DF IF TF SF ZF AF PF CF
1**Description** This instruction sets the carry flag to 1.**Example** STC

STD Set Direction Flag**Syntax** STD**Function** DF = 1**Flags** OF DF IF TF SF ZF AF PF CF
1**Description** This instruction sets the direction flag to 1. This causes the string instructions to autodecrement the SI and/or DI index registers. If DF = 0, then SI and/or DI are autoincremented.**Example** STD

STI Set Interrupt Enable Flag

Syntax STI

Function IF = 1

Flags OF DF IF TF SF ZF AF PF CF
1

Description This instruction sets the interrupt enable flag to 1. This allows the CPU to recognise interrupts occurring on the INTR line.

Example STI

STOS Store String Data

Syntax STOS destination_string

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction transfers a byte or word from register AX or AL to the string element addressed by DI, and updates DI to point to the next location in the string.

Example STOS

SUB Subtract

Syntax SUB destination, source

Function Destination = destination - source

Flags OF DF IF TF SF ZF AF PF CF
 * * * * * * * *

Description This instruction subtracts the byte or word source operand from the destination operand, and places the result in the destination operand.

Example SUB AX,BX

TEST Logical Compare

Syntax TEST destination, source

Function Destination is ANDED with source

Flags OF DF IF TF SF ZF AF PF CF
 0 * * * * * * 0

Description This instruction performs a logical AND of the two operands and updates the flags without returning the result.

Example TEST AL,2Ah

WAIT Wait

Syntax WAIT

Function None

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction causes the CPU to enter a wait state.

Example WAIT

XCHG Exchange

Syntax XCHG destination, source

Function destination = source; source = destination

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction swaps the contents of the source and destination operands.

Example XCHG AX,CX

XLAT Translate by Look-Up Table

Syntax XLAT

Function AL = DS:(BX + AL).

Flags OF DF IF TF SF ZF AF PF CF

Description This instruction uses the value in the AL register as an offset from the beginning of the table placed at the memory location DS:BX. The byte at this offset replaces the contents of the AL register.

Example XLAT TABLE

XOR Exclusive OR

Syntax XOR destination, source

Function Destination = destination XOR source.

Flags OF DF IF TF SF ZF AF PF CF
0 * * * 0

Description This instruction performs an exclusive or of the two byte or word operands and returns the result to the destination operand.

Example XOR AX,BX

Appendix 3 - Two-Pass Assembler/Disassembler

```
APP Debug
TYPE $1000
ICON "PIC\GRAY.PIC"
ENDA
```

```
PROC DEBUG:
GLOBAL A&,C&,BYTE%,BYTE2%,COL%,ADD$(4),ZEROS$(4),L%
GLOBAL GS$(4),JEF%,BY$(2),ROW%,SKEY%,FL%,KF%,TC%
GLOBAL T%(2),CS%,CSS$(4),E%,CSS$(5),ASS$(2),ASS$(20)
GLOBAL CX%,CY%,TOTAL&,A$(1),D%(6),PAR%,STR$(128)
GLOBAL BITE1%,BITE2%,BITETOT%,NAD$(4),NADD$(4)
GLOBAL PADD&,BITE%,K%,ORG&,X%,RES%,LOADNMS$(20)
GLOBAL BUFFER%(8192)
RESET::
CAPS:
RES%=0
ORG&=ADDR(BUFFER%())
FL%=0 :KF%=1
ZEROS$="0"
T%(1)=$C88C
T%(2)=$CB
CS%=USR(ADDR(T%(1)),0,0,0,0)
CSS=HEX$(CS%)
L%=4-LEN(CSS)
CSS$=REPT$(ZEROS$,L%)+CSS
A&=ORG&
WHILE 1
CURSOR OFF
CLS
ROW%=0
STATUSWIN ON :FONT 11,16
```

```
DO
PRINT ""
COL%=0
ADD$=HEX$(A&)
L%=4-LEN(ADD$)
PRINT REPT$(ZEROS$,L%);ADD$;" ";
DO
BYTE%=PEEKB(A&)
IF BYTE%>32
ASS$=CHR$(BYTE%)
ELSE
ASS$="."
ENDIF
ASS$=ASS$+ASS$
BY$=(HEX$(BYTE%))
L%=2-LEN(BY$)
PRINT REPT$(ZEROS$,L%);BY$;" ";
COL%=COL%+1
A&=A&+1
UNTIL COL%=8
PRINT ASS$;
ASS$=""
ROW%=ROW%+1
UNTIL ROW%=8
A&=A&-$40
POSTHELP::
SKEY%=GET
```

```
CAPS:
IF SKEY%=290 REM MENU KEY
K%=0
GOMENU:
IF RES%=1
```

Appendix 3 - Two-pass Assembler/Disassembler Listing

```
GOTO RESET::
ENDIF
IF X%=0
  A&=ORG&
ENDIF
ELSEIF SKEY% AND $200 REM HOT KEY
  K%=SKEY%-$200
  GOMENU:
  IF RES%=1
    GOTO RESET::
  ENDIF
  IF X%=0
    A&=ORG&
  ENDIF
  ELSEIF SKEY%=291 REM HELP KEY
    SAVE:
    GOHELP:
    LOAD:
    GOTO POSTHELP::
  ELSEIF SKEY%=256
    IF A&>$3F
      A&=A&-$40
    ELSE
      A&=0
    ENDIF
  ELSEIF SKEY%=259
    IF A&>0
      A&=A&-1
    ENDIF
  ELSEIF SKEY%=258
    A&=A&+1
  ELSEIF SKEY%=257
    A&=A&+$40
```

```
ELSEIF SKEY%=261
  A&=A&+$8
ELSEIF SKEY%=260
  IF A&>7
    A&=A&-$8
  ELSE
    A&=0
  ENDIF
ELSEIF SKEY%=%Q OR SKEY%=27
  CLS :AT 16,4 :PRINT"QUIT(Y/N)"
  IF GET=%Y
    STOP
  ENDIF
ENDIF
SKEY%=0
ENDWH
ENDP

PROC procz:
  RES%=1
ENDP

PROC proch:
  STATUSWIN OFF
  FONT 10,16
  A&=ORG&
ENDP
```

```
PROC proce:
LOADM"LOC::M:\OPO\DBASE.OPO"
STATUSWIN OFF
FONT 10,16
ED:
UNLOADM"LOC::M:\OPO\DBASE.OPO"
ENDP
```

```
PROC procf:
STATUSWIN OFF
FONT 10,16
LST:
ENDP
```

```
PROC procm:
GSAVEBIT "LOC::M:\OPD\DUMP.PIC"
MODIFY:
ENDP
```

```
PROC procd:
HEX:
IF A$<"Q"
PAUSE 5
A&=TOTAL&
ORG&=TOTAL&
ENDIF
ENDP
```

```
PROC procr:
LOCAL T%(2),U%,AX%,BX%,CX%,DX%,CS%,DS%,SS%,ES%,DI%,SI%
LOCAL SP%,BP%
CLS
T%(1)=$D889
T%(2)=$CB
BX%=USR(ADDR(T%(1)),0,0,0,0)
T%(1)=$C889
T%(2)=$CB
CX%=USR(ADDR(T%(1)),0,0,0,0)
T%(1)=$D089
T%(2)=$CB
DX%=USR(ADDR(T%(1)),0,0,0,0)
T%(1)=$C88C
T%(2)=$CB
CS%=USR(ADDR(T%(1)),0,0,0,0)
T%(1)=$D88C
T%(2)=$CB
DS%=USR(ADDR(T%(1)),0,0,0,0)
T%(1)=$D08C
T%(2)=$CB
SS%=USR(ADDR(T%(1)),0,0,0,0)
T%(1)=$C08C
T%(2)=$CB
ES%=USR(ADDR(T%(1)),0,0,0,0)
T%(1)=$F089
T%(2)=$CB
SI%=USR(ADDR(T%(1)),0,0,0,0)
T%(1)=$F889
T%(2)=$CB
DI%=USR(ADDR(T%(1)),0,0,0,0)
T%(1)=$E089
T%(2)=$CB
```

Appendix 3 - Two-pass Assembler/Disassembler Listing

```
SP%=USR(ADDR(T%(1)),0,0,0,0)
T%(1)=$E889
T%(2)=$CB
BP%=USR(ADDR(T%(1)),0,0,0,0)
PRINT"      8086 REGISTERS"
PRINT
PRINT
PRINT" AX=0000"; BX=";000"+HEX$(BX%);"
CX=";000"+HEX$(CX%); DX=";000"+HEX$(DX%)
PRINT
PRINT" CS=";HEX$(CS%); DS=";HEX$(DS%); SS=";HEX$(SS%);"
ES=";HEX$(ES%)
PRINT
PRINT" SI=";000"+HEX$(SI%); DI=";HEX$(DI%);"
SP=";0"+HEX$(SP%); BP=";0"+HEX$(BP%)
GET
CLS
ENDP

PROC GETSEG:
T%(1)=$C88C
T%(2)=$CB
CS%=USR(ADDR(T%(1)),0,0,0,0)
CS$=HEX$(CS%)
L%=4-LEN(CS$)
CSS$=REPT$(ZEROS$,L%)+CS$
ENDP

PROC CSEG:
CLS
PRINT"Code segment value: " :INPUT C&
ENDP
```

```
PROC procy:
LOCAL WORD$(255),CP%,PS%,AS&
CLS
PRINT"Address: $";HEX$(A&)
PRINT"Enter text.", :INPUT WORD$
CP%=LEN(WORD$)+1
PS%=1
AS&=A&
DO
POKEB AS&,ASC(MID$(WORD$,PS%,1))
AS&=AS&+1
PS%=PS%+1
UNTIL PS%=CP%
ENDP
```

```
PROC MODIFY:
CX%=6 :CY%=2 :KF%=1
AT 5,1 :PRINT" <ENT>-Poke byte, G-Run code "
DO
AT CX%,CY%
CURSOR ON
SKEY%=GET
CAPS:
IF FL%=1
FL%=0
RETURN
ENDIF
IF SKEY%=256
CY%=CY%-1
IF CY%<2
CY%=CY%+1
ENDIF
ELSEIF SKEY%=259
```

```

CX%=CX%-3
IF CX%<6
  CX%=CX%+3
ENDIF
ELSEIF SKEY%=257
  CY%=CY%+1
  IF CY%>9
    CY%=CY%-1
  ENDIF
ELSEIF SKEY%=258
  CX%=CX%+3
  IF CX%>28
    CX%=CX%-3
  ENDIF
ELSEIF SKEY%=13
  CURSOR OFF
  BITETOT%=BITE1%+BITE2%
  PADD&=A&+(CY%-2)*8+CX%/3-2
  POKEB PADD&,BITETOT%
AT 5,1 :PRINT " Byte poked at address: ";HEX$(PADD&); " "
  PAUSE 25
AT 5,1 :PRINT " <ENT>-Poke byte, G-Run code "
ELSEIF SKEY%=%G
  CURSOR OFF
  PADD&=A&+(CY%-2)*8+CX%/3-2
  AT 1,1
  PRINT " Y-Confirm, address=";HEX$(PADD&); " "
  SKEY%=GET
  IF SKEY%=%Y
    PADD&=A&+(CY%-2)*8+CX%/3-2
    CS%=USR(PADD&,0,0,0,0)
    AT 1,1
    PRINT " Code executed ** AX=";HEX$(CS%); " "
  
```

```

GET
AT 5,1 :PRINT" <ENT>-Poke byte, G-Run code "
ELSEIF SKEY%=%Q
RETURN
ENDIF
CURSOR ON
ELSEIF KF%=2
ONE:
CX%=CX%-1
KF%=1
BITE1%=BITE%
ELSEIF KF%=1
ONE:
CX%=CX%+1
KF%=2
BITE2%=BITE%*16
ENDIF
UNTIL SKEY%=%Q OR SKEY%=27
CURSOR OFF
ENDP

```

```

PROC CAPS:
IF SKEY%>96
IF SKEY%<200
CURSOR OFF
CLS :AT 11,4 :PRINT"Turn Caps Lock ON"
FL%=1
PAUSE 25
ENDIF
ENDIF
ENDP

```

Appendix 3 - Two-pass Assembler/Disassembler Listing

```
PROC SAVE:
GSAVEBIT ("LOC::M:\OPD\DUMP.PIC")
ENDP
```

```
PROC LOAD:
LOCAL ID%
ID%=GLOADBIT ("LOC::M:\OPD\DUMP.PIC")
GUSE 1
GAT 0,0
GCOPY ID%,0,0,240,80,3
GCLOSE ID%
ENDP
```

```
PROC ONE:
IF SKEY%=%1
  AT CX%,CY%
  PRINT "1"
  BITE%=1
ELSEIF SKEY%=%2
  AT CX%,CY%
  BITE%=2
  PRINT "2"
ELSEIF SKEY%=%3
  AT CX%,CY%
  BITE%=3
  PRINT "3"
ELSEIF SKEY%=%4
  AT CX%,CY%
  BITE%=4
  PRINT "4"
ELSEIF SKEY%=%5
  AT CX%,CY%
  BITE%=5
```

```

PRINT "5"
ELSEIF SKEY%=%6
  AT CX%,CY%
  BITE%=6
  PRINT "6"
ELSEIF SKEY%=%7
  AT CX%,CY%
  BITE%=7
  PRINT "7"
ELSEIF SKEY%=%8
  AT CX%,CY%
  BITE%=8
  PRINT "8"
ELSEIF SKEY%=%9
  AT CX%,CY%
  BITE%=9
  PRINT "9"
ELSEIF SKEY%=%0
  AT CX%,CY%
  BITE%=0
  PRINT "0"
ELSEIF SKEY%=%A
  AT CX%,CY%
  BITE%=10
  PRINT "A"
ELSEIF SKEY%=%B
  AT CX%,CY%
  BITE%=11
  PRINT "B"
ELSEIF SKEY%=%C
  AT CX%,CY%
  BITE%=12
  PRINT "C"

```

```
ELSEIF SKEY%=%D
  AT CX%,CY%
  BITE%=13
  PRINT "D"
ELSEIF SKEY%=%E
  AT CX%,CY%
  BITE%=14
  PRINT "E"
ELSEIF SKEY%=%F
  AT CX%,CY%
  BITE%=15
  PRINT "F"
ENDIF
ENDP
```

```
PROC HEX:
LOCAL NUM%,CO%
START::
CLS
PRINT"Address(HEX): ";
CO%=0 :TOTAL&=0
DO
  A$=GET$
  IF A$="Q"
    RETURN
  ENDIF
  PRINT A$;
  NUM%=ASC(A$)
  IF NUM%<48
    GOTO START::
  ENDIF
  IF NUM%>57
    IF NUM%<65
```

```
GOTO START::
ENDIF
ENDIF
NUM%=NUM%-48
IF NUM%>9
    NUM%=NUM%-7
    IF NUM%>15
        GOTO START::
    ENDIF
ENDIF
TOTAL&=TOTAL&*16+NUM%
CO%=CO%+1
UNTIL CO%=4
ENDP
```

```
PROC procs:
LOCAL RET%,HANDLE%,MODE%,LENGTH%
LOCAL SEEK%,NAS(9),CHOICE$(1)
GLOBAL NUM2%,CO%,TOTAL2%,A2$(1),TYPEIT$(20)
GLOBAL START%,A$(1)
LENGTH%=$0100
CLS
PAR%=17
DATA:
IF STR$=""
    RETURN
ENDIF
MODE%=$0102
TYPEIT$="Address(HEX): "
HEX2:
START%=TOTAL2%
TYPEIT$="Byte length(HEX): "
```

HEX2:

LENGTH%=TOTAL2%

RET%=IOOPEN(HANDLE%,STR\$,MODE%)

RET%=IOWRITE(HANDLE%,START%,LENGTH%)

RET%=IOCLOSE(HANDLE%)

ENDP

PROC HEX2:

LOCAL CO2%

START::

CLS

PRINT TYPEIT\$;

CO2%=0 :TOTAL2%=0

DO

A\$=GET\$

IF A\$="Q"

RETURN

ENDIF

PRINT A\$;

NUM2%=ASC(A\$)

IF NUM2%<48

GOTO START::

ENDIF

IF NUM2%>57

IF NUM2%<65

GOTO START::

ENDIF

ENDIF

NUM2%=NUM2%-48

IF NUM2%>9

NUM2%=NUM2%-7

IF NUM2%>15

GOTO START::

```
ENDIF
ENDIF
TOTAL2%=TOTAL2%*16+NUM2%
CO2%=CO2%+1
UNTIL CO2%=4
PAUSE 10
ENDP

PROC procl:
LOCAL RET%,HANDLE%,MODE%,LENGTH%
LOCAL SEEK%,NAS$(9),CHOICES$(1),LE%,AAAD%
GLOBAL TYPEITS$(30),TOTAL2%,NUM2%
CLS
PAR%=64
DATA:
IF STR$=""
RETURN
ENDIF
CLS
TYPEITS$="Load Address(HEX): "
HEX2:
AAAD%=TOTAL2%
LE%=16384
MODE%=0
RET%=IOOPEN(HANDLE%,STR$,MODE%)
RET%=IOREAD(HANDLE%,AAAD%,LE%)
RET%=IOCLOSE(HANDLE%)
RETURN
ENDP

PROC procp:
GOHELP:
ENDP
```

Appendix 3 - Two-pass Assembler/Disassembler Listing

```
PROC proci:
LOCAL SPCE%,SPA%,SKEY%
OPEN "LOC::M:DAT\MNEMONIC.DBF",A,TEX$,COD$,LECS$
FIRST
CLS
DO
  SPCE%=20-LEN(A.TEX$)
  PRINT POS,A.TEX$;
  SPA%=0
  DO
    PRINT " ";
    SPA%=SPA%+1
  UNTIL SPA%=SPCE%
  PRINT A.COD$
  SKEY%=GET
  IF SKEY%=%Q OR SKEY%=27
    CLOSE
    RETURN
  ELSEIF SKEY%=256
    POSITION POS-2
    CLS
  ENDIF
NEXT
UNTIL EOF
CLOSE
ENDP
```

```
PROC proct:
LOCAL D$(128),C%
FONT 9,16
D$=DIR$("ROM::")
WHILE D$<>""
```

```

DO
PRINT D$
D$=DIR$("")
C%=C%+1
UNTIL C%=17
GET
C%=0
ENDWH
ENDP

PROC LST:
LOCAL LINE$(20),SKEY%,NM$(3),NNN$(20)
CLS
PAR%=64
DATA:
IF STR$=""
RETURN
ENDIF
OPEN STR$,C,NM$,LINE$
FIRST
DO
CLS
FIRST
DO
PRINT C.NM$,C.LINE$
SKEY%=GET
IF SKEY%=256
CLS
BACK
ELSEIF SKEY%=%Q OR SKEY%=27
CLOSE
RETURN
ELSE

```

Appendix 3 - Two-pass Assembler/Disassembler Listing

```
NEXT
ENDIF
UNTIL EOF
UNTIL SKEY%=%Q
CLOSE
ENDP
```

```
PROC procu:
LOADM"LOC::M:\OPO\ADISS.OPO"
STATUSWIN OFF
FONT 10,16
DISS:
UNLOADM"LOC::M:\OPO\ADISS.OPO"
ENDP
```

```
PROC proca:
LOCAL A%,B%,A&,OFFSET%,HASH$(1)
LOCAL MNUS$(20),OP$(4),CODE$(20)
GLOBAL SYNTAX%,C%,COUNT%,A$(20),ADD%,CODE1%
GLOBAL ADDR$(4),BS(1),TOTAL3%,NUM3%,FF%
GLOBAL TOTAL4%,ECOUNT%,C1%,SCAN%,OFFSET&
GLOBAL TABLE$(100,10),SYMCNT%,SYMCNT2%,ADR&(100)
GLOBAL JFLAG%,T%,START%,SERCH$(4),POKEADD&
GLOBAL ADDRESS&,BYTE%,BYTES(2),PS%,NUM%(2),WORD%,AA%
GLOBAL NNN$(127),BUFFC%,MEMCNT%,MEMTABS(100,14)
LOCAL MODE2%,LE2%,RET2%,HANDLE2%
TC%=0
X%=0
OPEN "LOC::M:\DAT\MNEMONIC.DBF",A,TEXT,COD$,LENS
CLS
STATUSWIN OFF
FONT 10,16
CLS
```

```

PAR%=64
DATA:
IF STR$=""
RETURN
ENDIF
OPEN STR$,B,LINES$,SOURCE$
COUNT%=COUNT :JFLAG%=1
BUFFC%=COUNT%*3
AGAIN::
USE A :FIRST
USE B :FIRST
SYMCNT2%=1
T%=1
SYMCNT%=1
TOTAL3%=0
TOTAL4%=0 :ECOUNT%=0 :CLS :C%=1 :C1%=1
IF JFLAG%=1
CLS
AT 1,1
PRINT "Pass 2"
PAUSE 10 :CLS
ELSE
AT 1,1
PRINT "Pass 1"
PAUSE 10 :CLS
ENDIF
DO
OP$=""
USE B
POSITION C1%
A$=B.SOURCE$
AT 1,C%
PRINT C1%

```

```

AT 5,C%
PRINT AS
HASH$=MID$(A$,1,1)
IF C1%=1
IF MID$(A$,1,3)<>"ORG"
SYNTAX%=1
ERROR:
ELSE
IF MID$(A$,4,1)="D"
TOTAL3%=VAL(MID$(A$,5,4))
FF%=1
HEX3:
GOTO CON::
ENDIF
IF MID$(A$,5,1)="?" OR MID$(A$,4,1)="?"
X%=1
TOTAL3%=ORG&
AT 20,C%
PRINT "Start address: $";HEX$(TOTAL3%);" (";TOTAL3%;" dec)"
IF JFLAG%=1
START%=TOTAL3%
ENDIF
ELSE
ADDR$=MID$(A$,5,4)
HEX3:
ORG&=TOTAL3%
ENDIF
GOTO CON::
ENDIF
ENDIF
IF HASH$="#"
IF MID$(A$,7,1)="X" OR MID$(A$,7,1)="I"
OP$=MID$(A$,9,4)

```

```
A$=MID$(A$,2,7)+"FFFF"
ELSE OPS=MID$(A$,9,2)
A$=MID$(A$,2,7)+"FF"
ENDIF
ENDIF
IF HASH$="."
OPS=MID$(A$,6,2)
A$=MID$(A$,2,4)+"FF"
ENDIF
IF HASH$="J"
IF JFLAG%=1
IF MID$(A$,5,1)="X"
SYNTAX%=7
ELSE
A$=MID$(A$,1,3)+"FF"
SYNTAX%=5
ERROR:
ENDIF
ELSEIF JFLAG%=-1
IF MID$(A$,5,1)="X"
SYNTAX%=7
ELSEIF JFLAG%=-1
SYMCNT2%=0
SERCH$=MID$(A$,5,4)
DO
SYMCNT2%=SYMCNT2%+1
IF SYMCNT2%>99
SYNTAX%=9
ERROR:
GOTO CON::
ENDIF
UNTIL SERCH$=TABLE$(SYMCNT2%)
OFFSET&=ADR&-(SYMCNT2%)-START%-TOTAL4%-2
```

```
IF OFFSET&>126
SYNTAX%=8
ERROR:
OFFSET&=0
ENDIF
IF OFFSET&<0
IF OFFSET&<-126
SYNTAX%=8
ERROR:
OFFSET&=0
ELSE
OFFSET&=256+OFFSET&
ENDIF
ENDIF
OP$=HEX$(OFFSET&)
IF LEN(OP$)<2
OP$="0"+OP$
ENDIF
A$=MID$(A$,1,3)+"FF"
ENDIF
ENDIF
ENDIF

IF HASH$="<"
IF MID$(A$,9,3)="MEM"
OP$=HEX$(ORG&+BUFFC%+(VAL(MID$(A$,12,1)))^2)
MEMCNT%=MEMCNT%+1
MEMTAB$(MEMCNT%)=MID$(A$,9,4)+"..." +OP$
ELSE
OP$=MID$(A$,9,4)
ENDIF
A$=MID$(A$,2,6)+"[FFFF]"
ENDIF
```

```
IF HASH$=">"
  IF MID$(A$,9,3)="MEM"
    OP$=HEX$(ORG&+BUFFC%+(VAL(MID$(A$,12,1))) *2)
    MEMCNT%=MEMCNT%+1
    MEMTAB$(MEMCNT%)=MID$(A$,9,4)+"..." +OPS
  ELSE
    OP$=MID$(A$,9,4)
  ENDIF
  A$="MOV[FFFF],"+MID$(A$,5,2)
ENDIF
IF HASH$="*"
  IF LEN(A$)>5 OR LEN(A$)<5
    SYNTAX%=6
    ERROR:
    GOTO CON::
  ENDIF
  SYNTAX%=5
  SYMTAB:
  GOTO CON::
ENDIF
IF HASH$=" "
  SYNTAX%=2
  ERROR:
  GOTO CON::
ENDIF
IF MID$(A$,1,3)="END"
  SYNTAX%=3
  ERROR:
  GOTO CON::
ENDIF
USE A
FINDFIELD (A$,1,1,3)
IF OP$<>""
```

```

CODE$=MID$(A.CODE$,1,2)+MID$(OP$,3,2)+MID$(OP$,1,2)
ELSE CODE$=A.CODE$
ENDIF
IF CODE$=""
SYNTAX%=4
ERROR:
GOTO CON::
ENDIF
ONERR F::
TOTAL4%=TOTAL4%+VAL(A.LEN$)
ADDRESS&=(TOTAL4%+TOTAL3%-VAL(A.LEN$))
F::
AT 20,C%
PRINT CODE$
AT 46,C%
PRINT HEX$(ADDRESS&)
IF JFLAG%=1
BYTE%=0 :AA%=0
DO
PS%=1
DO
BYTE$=MID$(CODE$,BYTE%+PS%,1)
NUM%(PS%)=ASC(BYTE$)
IF NUM%(PS%)>64
NUM%(PS%)=NUM%(PS%)-55
ELSE NUM%(PS%)=NUM%(PS%)-48
ENDIF
PS%=PS%+1
UNTIL PS%=3
WORD%=NUM%(1)*16+NUM%(2)
POKEB ADDRESS&+AA%,WORD%
BYTE%=BYTE%+2
AA%=AA%+1

```

```

UNTIL BYTE%=LEN(CODE$)
ENDIF
CON::
C1%=C1%+1
C%=C%+1
IF C%>13
  C%=1
  GET
  CLS
ENDIF
UNTIL C1%>COUNT%
AT 1,C%
PRINT "Code length:",TOTAL4%,".", "End
address:",HEX$(TOTAL3%+TOTAL4%-1);"(hex)"
AT 1,C%+2
PRINT ECOUNT%,"Errors"
GET
CLS
C%=3
AT 1,1
PRINT"Symbol table:"
SYMCNT%=1
DO
  AT 1,C%
  PRINT TABLE$(SYMCNT%);"...";HEX$(ADR&(SYMCNT%))
  SYMCNT%=SYMCNT%+1
  C%=C%+1
  IF C%>13
    CLS
    C%=1
  ENDIF
UNTIL ADR&(SYMCNT%)=0
C%=3

```

```
AT 20,1
PRINT "Memory table:"
MEMCNT%=1
DO
  AT 20,C%
  PRINT MEMTAB$(MEMCNT%)
  MEMCNT%=MEMCNT%+1
  C%=C%+1
  TC%=TC%+1
  IF C%>13
    GET
    CLS
    C%=1
  ENDIF
UNTIL MEMTAB$(MEMCNT%)=""
IF JFLAG%=1
  GET
  CLS
  JFLAG%=JFLAG%*-1
  GOTO AGAIN::
ELSE
  JFLAG%=1
  SCAN%=GET
ENDIF
CLOSE
CLOSE
ENDP
```

PROC ERROR:

LOCAL MESSAGE\$(255),T%

MESSAGE\$="missing ORG remove tab remove END unknown codeJump label
label lengthJump inst. jmp distancemissing lbl.missing file"

```
SYNTAX%=SYNTAX%*12-11
```

```
T%=30
```

```
AT T%,C%
```

```
PRINT MID$(MESSAGE$,SYNTAX%,12)
```

```
ECOUNT%=ECOUNT%+1
```

```
ENDP
```

```
PROC SYMTAB:
```

```
TABLE$(SYMCNT%)=MID$(A$,2,4)
```

```
IF A.LEN$=""
```

```
A.LEN$="0"
```

```
ENDIF
```

```
ADR&$(SYMCNT%)=(TOTAL4%+TOTAL3%)
```

```
SYMCNT%=SYMCNT%+1
```

```
TC%=TC%+1
```

```
ENDP
```

```
PROC HEX3:
```

```
LOCAL CO2%
```

```
IF FF%=1
```

```
GOTO E::
```

```
ENDIF
```

```
START::
```

```
CO2%=1 :TOTAL3%=0
```

```
DO
```

```
B$=MID$(ADDR$,CO2%,1)
```

```
IF B$=""
```

```
RETURN
```

```
ENDIF
```

```
NUM3%=ASC(B$)
```

```
IF NUM3%<48
```

```
GOTO START::
```

Appendix 3 - Two-pass Assembler/Disassembler Listing

```
ENDIF
IF NUM3%>57
IF NUM3%<65
  GOTO START::
ENDIF
ENDIF
NUM3%=NUM3%-48
IF NUM3%>9
  NUM3%=NUM3%-7
  IF NUM3%>15
    GOTO START::
  ENDIF
ENDIF
TOTAL3%=TOTAL3%*16+NUM3%
CO2%=CO2%+1
UNTIL CO2%=5
E::
FF%=0
AT 20,C%
PRINT "Start address:",TOTAL3%,"(dec)"
IF JFLAG%=1
  START%=TOTAL3%
ENDIF
ENDP

PROC JUMP:
ENDP

PROC GOMENU:
LOCAL H$(24),A$(5)
H$="lsdmghfaznvrpiqcxbytu"
WHILE 1
  IF k%=0
```

MINIT

```

MCARD"Code","Load",%l,"Save",%s,"Dump",%d,"Modify/Run",%m,"IP="+H
EX$(ORG&),%h,"IP=Default",%z,"Enter string",%y
MCARD"Source","Edit",%e,"List",%f,"Assemble",%a,"Unassemble",%u
MCARD"Special","Registers",%r,"Help",%p,"Mnemonics",%i,"Dir",%b,"ROM
Dir",%t,"Exit menu",%x,"Quit",%q
K%=MENU
IF K% AND (LOC(H$,CHR$(K%))<0)
A$="PROC"+CHR$(K%)
@(A$):
RETURN
ENDIF
ELSE
IF LOC(H$,CHR$(K%))
a$="proc"+CHR$(K%)
@(A$):
ENDIF
RETURN
ENDIF
ENDWH
ENDP

```

```

PROC procq:
STOP
ENDP

```

```

PROC procx:
RETURN
ENDP

```

```
PROC procb:
LOCAL D$(128),P%
FONT 9,16
SETPATH"LOC::M:\DAT\"
D$=DIR$("M:\DAT\*.DBF")
WHILE D$<>""
PRINT MID$(D$,13,12)
P%=P%+1
IF P%>14
GET
P%=0
ENDIF
D$=DIR$("")
ENDWH
GET
ENDP
```

```
PROC DATA:
LOCAL P$(4)
SETPATH"LOC::M:\DAT\"
P$="File"
DINIT
DFILE STR$,P$,PAR%
DIALOG
ENDP
```

```
PROC GOHELP:
LOCAL S%
DINIT "General help"
DTEXT "",CHR$(254)+"Debug", $500
DTEXT "",CHR$(254)+"Machine code", $500
DTEXT "",CHR$(254)+"Editor", $500
```

```
DTEXT "",CHR$(254)+"Assembler", $500
DTEXT "",CHR$(254)+"Disk utilities", $500
DTEXT "",CHR$(254)+"Mnemonics", $500
DTEXT "",CHR$(254)+"Source syntax", $500
S%=DIALOG
IF S%=2
  DINIT "Debug"
  DTEXT "", "Debug starts at free address. Move around", 0
  DTEXT "", "+/- 40 bytes using cursor keys. "+CHR$(02)+" and cursor", 0
  DTEXT "", "keys for single byte scrolling.", 0
  DTEXT "", CHR$(2)+"D dumps from entered hex address.", 0
  DTEXT "", CHR$(2)+"H dump from previous address.", 0
  DTEXT "", "Enter text using the string option (" +CHR$(02)+"Y)", 0
  DIALOG
ELSEIF S%=3
  DINIT "Machine code"
  DTEXT "", "Enter code in free areas of memory.", 0
  DTEXT "", "Ensure code ends in RETF before running.", 0
  DIALOG
ELSEIF S%=4
  DINIT "Editor"
  DTEXT "", "If new name is not entered, Editor will load", 0
  DTEXT "", "existing source file. Use up/down keys to", 0
  DTEXT "", "move around text. Type ""I"" to insert, ""A"" to", 0
  DTEXT "", "append, ""D"" to delete line.", 0
  DTEXT "", "Type END, followed by ""Q"" to save and exit.", 0
  DIALOG
ELSEIF S%=5
  DINIT "Assembler"
  DTEXT "", "Simply type source name, followed by", 0
  DTEXT "", "ENTER. Any key to move to next screen.", 0
  DTEXT "", "Code will be assembled at ORG address.", 0
  DTEXT "", "Correct any syntax errors with Editor.", 0
```

```

DTEXT "";"MEM1 to MEM9 available.",0
DIALOG
ELSEIF S%=6
DINIT "Disk utilities"
DTEXT "";"LIST file allows you to quickly examine",0
DTEXT "";"source file; cursor keys scroll through.",0
DTEXT "";"LOAD & SAVE act on machine code only.",0
DTEXT "";"Source is loaded and saved automatically",0
DTEXT "";"during editing. From menu, mnemonic file",0
DTEXT "";"can be loaded and examined-""Q"" to exit.",0
DIALOG
ELSEIF S%=7
DINIT "Mnemonics"
DTEXT "";"8086 Mnemonic.dbf should be in LOC::M:\DAT",0
DTEXT "";"New codes can be developed by using the assemble",0
DTEXT "";"(A) and unassemble option (U) in DEBUG on an",0
DTEXT "";"MSDOS PC computer. Compress file before copying",0
DTEXT "";"to a ROM pack-(see manual for details).",0
DIALOG
ELSEIF S%=8
DINIT "Source syntax"
DTEXT "";">MOVAL,[MEM1] >MOVAX,[MEM2] <MOVAX,[MEM3]",0
DTEXT "";"AL to MEM1 AX to MEM2 MEM3 to AX",0
DTEXT "";"#MOV AX,FFFF; #MOV BX,FFFF; *FRED; JNZ*FRED; JZ
*FRED.etc",0
DTEXT "";"AH=d7 d6 d5 d4 d3 d2 d1 d0 ; Use ""."" before INT: eg .INT 21;
",0
DTEXT "";" SF ZF X AF X PF X CF - LAHF(flag to AH),SAHF",0
DIALOG
ELSEIF S%=0
RETURN
ENDIF
ENDP

```

```

PROC DISS:
GLOBAL XA&,XORG&,XTOTAL&,XAS(12),XBUFFER%(100),XFL%,NO%
GLOBAL XST&,XP%,XSE$(12),XCO%,XY$(12),XSC%,XOP$(10),XB%
GLOBAL XT%(2),XCS%,XL%,XCS$(4),XCSS$(4),XZEROS$(4)
GLOBAL FQ%,R$(2),R2$(2),XSE2%,XFF%,XD%
CONT2::
XORG&=ADDR(XBUFFER%())
XZEROS$="0"
XT%(1)=$C88C
XT%(2)=$CB
XCS%=USR(ADDR(XT%(1)),0,0,0,0)
XCS$=HEX$(XCS%)
XL%=4-LEN(XCS$)
XCSS$=REPT$(XZEROS$,XL%)+XCS$
PROCD1:
XST&=XORG&
OPEN "LOC::M:\DAT\MNEMONIC.DBF",A,TEXT$,CODE$,LENS
CLS
DO
  XFL%=0
  IF PEEKW(XST&)= $FFFF
    XCO%=2
    PRINT HEX$(XCS%),": ";
    NO%=4-LEN(HEX$(XST&))
    WHILE NO%>0
      PRINT"0";
      NO%=NO%-1
    ENDWH
    PRINT HEX$(XST&);
    PRINT " FFFF  ????"
    GOTO CONT::
  ENDIF
  XSES=(HEX$(PEEKB(XST&))+ "*" )

```

```

IF LEN(XSE$)=2
  XSE$="0"+XSE$
ENDIF
XSE2%=PEEK(XST&)
IF MID$(XSE$,1,1)="7"
  XB%=PEEK(XST&+1)
  IF XB%>128
    XB%=256-XB%
    XOP$="-"+GEN$(XB%,3)
  ELSE
    XOP$="+"+GEN$(XB%,3)
  ENDIF
  XFL%=1
ENDIF

IF MID$(XSE$,1,1)="B"
  IF XSE2%<184
    XB%=PEEK(XST&+1)
    XOP$=HEX$(XB%)
    XFL%=1
    XFF%=1
  ENDIF
ENDIF
IF MID$(XSE$,1,2)="CD"
  XB%=PEEK(XST&+1)
  XOP$=HEX$(XB%)
  XFL%=1
ENDIF
PRINT XCSS$,".";
NO%=4-LEN(HEX$(XST&))
WHILE NO%<>0
  PRINT"0";
  NO%=NO%-1

```

ENDWH

PRINT HEX\$(XST&);
XP%=FINDFIELD(XSE\$,2,1,3)

```
IF A.LEN$=""
  A.LEN$="1"
ENDIF
XY$=A.LEN$
XCO%=VAL(XY$)
XD%=0
PRINT " ";
DO
  IF PEEKB(XST&+XD%)<16
    PRINT"0";
  ENDIF
  PRINT HEX$(PEEKB(XST&+XD%));
  XD%=XD%+1
UNTIL XD%=XCO%

XD%=3-XCO%
PRINT " ";
PRINT REPT$(" ",(XD%*2));
IF XCO%=2
  XSE$="*"+(HEX$(PEEKB(XST&+1)))
  IF LEN(XSE$)=2
    XSE$=MID$(XSE$,1,1)+"0"+MID$(XSE$,2,1)
  ENDIF
  IF XFL%=1
    XSE$="*FF"
  ENDIF
  XP%=FINDFIELD(XSE$,2,1,1)
```

```

ELSEIF XCO%=3
R2$=HEX$(PEEK(B(XST&+2))
R$=HEX$(PEEK(B(XST&+1))
IF LEN(R$)=1
R$="0"+R$
ENDIF
IF LEN(R2$)=1
R2$="0"+R2$
ENDIF
XOPS=R2$+R$
XCO%=3
PRINT " ";A.TEXT$;" (";XOPS;"h)"

GOTO CONT::
ENDIF
IF A.TEXT$=""
PRINT " ????"
ELSE
IF XOPS=""
PRINT " ";A.TEXT$
ELSE
IF XFF%=1
XFF%=0
PRINT " ";MID$(A.TEXT$,1,6);" ";XOPS
ELSE
PRINT " ";MID$(A.TEXT$,1,3);" ";XOPS
ENDIF
XOPS=""
ENDIF
ENDIF
CONT::
XOPS=""

```

```

XSC%=GET
XST&=XST&+XCO%
UNTIL XSC%=%Q OR XSC%=%D OR XSC%=%U
CLOSE
IF XSC%=%D OR XSC%=%U
GOTO CONT2::
ENDIF
ENDP

```

```

PROC PROCD1:
FQ%=0
HEX5:
IF XA$<"Q"
PAUSE 5
XA&=XTOTAL&
XORG&=XTOTAL&
ENDIF
ENDP

```

```

PROC HEX5:
LOCAL NUM%,XCO%
START::
CLS
PRINT"Disassemble from (HEX): ";
XCO%=0 :XTOTAL&=0
DO
XA$=GET$
IF XA$="Q"
RETURN
ELSEIF XA$="X"
XTOTAL&=A&
GOTO ENDI::
ENDIF

```

```
PRINT XA$;  
NUM%=ASC(XA$)  
IF NUM%<48  
  GOTO START::  
ENDIF  
IF NUM%>57  
  IF NUM%<65  
    GOTO START::  
  ENDIF  
ENDIF  
NUM%=NUM%-48  
IF NUM%>9  
  NUM%=NUM%-7  
  IF NUM%>15  
    GOTO START::  
  ENDIF  
ENDIF  
XTOTAL&=XTOTAL&+16+NUM%  
XCO%=XCO%+1  
UNTIL XCO%=4  
ENDI::  
ENDP
```

```
PROC ED:
LOCAL LINE$(20),SKEY%,NM$(3),NNN$(20),FL%
LOCAL POSS%,INPOS%,ENDPOS%,COUNT%,BUFFERS$(20),SET%
LOCAL INPOS2%,INSERT$(20)
START::
CLS
PRINT"Edit:", :INPUT NNN$
IF EXIST("LOC::M:\DAT\"+NNN$+".DBF")
OPEN "LOC::M:\DAT\"+NNN$+".DBF",C,NM$,LINES
FL%=1
ELSE
CLS
PRINT "New file" :PAUSE 20
CREATE "LOC::M:\DAT\"+NNN$+".DBF",C,NM$,LINES
FL%=0
ENDIF
FIRST
DO
CLS
FIRST
DO
PRINT C.NM$,C.LINE$
IF FL%=1
SKEY%=GET
ENDIF
IF SKEY%=256
CLS
BACK
ELSEIF SKEY%=%N
CLOSE
SKEY%=%A
GOTO START::
ELSEIF SKEY%=%Q
```

Appendix 4 - Source Editor Listing

```
CLOSE
RETURN
ELSEIF SKEY%=290 OR SKEY%=291
CLS
AT 4,4
PRINT "A-Append I-Insert D-Delete N-New Q-Quit"
GET :CLS
ELSEIF SKEY%=%I
INPOS%=POS
INPOS2%=POS+1
LAST
ENDPOS%=POS
COUNT%=ENDPOS%-INPOS%
SET%=0
POSITION INPOS%
BUFFER$=C.LINE$
PRINT " ";INPUT INSERT$
C.LINE$=INSERT$
CLS
DO
APPEND
INPOS%=INPOS%+1
POSITION INPOS%
BUFFER$=C.LINE$
SET%=SET%+1
UNTIL SET%=COUNT%+1
SET%=0
REM UPDATE
DO
POSITION INPOS2%
ERASE
SET%=SET%+1
UNTIL SET%=COUNT%
```

```
ELSEIF SKEY%=%A OR FL%=0
```

```
DO
```

```
LAST
```

```
PRINT POS+FL%
```

```
C.NM$=GEN$(POS+FL%,3)
```

```
FL%=1
```

```
PRINT"Enter source:";
```

```
INPUT C.LINE$
```

```
APPEND
```

```
CLS
```

```
BACK
```

```
IF C.LINE$="END"
```

```
ERASE
```

```
FIRST
```

```
CLS
```

```
ENDIF
```

```
NEXT
```

```
UNTIL C.LINE$="END"
```

```
ELSEIF SKEY%=%D
```

```
ERASE
```

```
CLS
```

```
ELSE
```

```
NEXT
```

```
ENDIF
```

```
UNTIL EOF
```

```
UNTIL SKEY%=%Q
```

```
CLOSE
```

```
ENDP
```

Appendix 5 - 8086 Mnemonic File

Enter as example below, and save file as LOC::M:MNEMONIC.DBF

Example:

TEXT: AAA

CODE: 37

LENGTH: 1

AAA 37 1
AAD D50A 2
AAM D40A 2
AAS 3F 1
ADC AL,AH 10E0 2
ADD AX,AX 01C0 2
ADD AX,BX 01D8 2
ADD AX,CX 01C8 2
ADD AX,DX 01D0 2
ADD AX,FFFF 05FFFF 3
ADD BX,AX 01C3 2
AND AX,BX 21D8 2
AND AX,FFFF 25FFFF 3
CALL AX FFD0 2
CALL BX FFD3 2
CALL CX FFD1 2
CALL DX FFD2 2
CBW 98 1
CLC F8 1
CLD FC 1
CLI FA 1
CMC F5 1
CMP AX,BX 39D8 2

CMPSB A6 1
CMPSW A7 1
CMPSW A6 1
CMPSW A7 1
CS: 2E 1
CWD 99 1
DAA 27 1
DAS 2F 1
DEC AX 48 1
DEC BX 4B 1
DEC CX 49 1
DEC DX 4A 1
DIV AX F7F0 2
DIV BX F7F3 2
DIV CX F7F1 2
DIV DX F7F2 2
DS: 3E 1
ES: 26 1
IN AX,DX ED 1
INC AX 40 1
INC BX 43 1
INC CX 41 1
INC DX 42 1
INT FF CDFF 2
INTO CE 1
IRET CF 1
JA FF 77FF 2
JB FF 72FF 2
JBEFF 76FF 2
JG FF 7FFF 2
JGEFF 7DFF 2
JL FF 7CFF 2
JLEFF 7EFF 2

Appendix 5 - 8086 Mnemonic File

JMPAX FFE0 2
JMPBX FFE3 2
JMPCX FFE1 2
JMPDX FFE2 2
JNBFF 73FF 2
JNCFF 73FF 2
JNOFF 71FF 2
JNSFF 79FF 2
JNZFF 75FF 2
JO FF 70FF 2
JPEFF 7AFF 2
JPOFF 7BFF 2
JS FF 78FF 2
JZ FF 74FF 2
LAHF 9F 1
LOCK F0 1
LODSB AC 1
LODSW AD 1
MOV AH,AL 88C4 2
MOV AH,BH 88FC 2
MOV AH,BL 88DC 2
MOV AH,CH 88EC 2
MOV AH,CL 88CC 2
MOV AH,DH 88F4 2
MOV AH,DL 88D4 2
MOV AH,FF B4FF 2
MOV AL,AH 88E0 2
MOV AL,BH 88F8 2
MOV AL,BL 88D8 2
MOV AL,CH 88E8 2
MOV AL,CL 88C8 2
MOV AL,DH 88F0 2
MOV AL,DL 88D0 2

MOV AL,FF B0FF 2
MOV AX,BX 89D8 2
MOV AX,CS 8CC8 2
MOV AX,CX 89C8 2
MOV AX,DI 89F8 2
MOV AX,DS 8CD8 2
MOV AX,DX 89D0 2
MOV AX,ES 8CC0 2
MOV AX,FFFF B8FFFF 3
MOV AX,SI 89F0 2
MOV AX,SP 89E0 2
MOV AX,SS 8CD0 2
MOV BH,CH 88EF 2
MOV BH,CL 88CF 2
MOV BH,DH 88F7 2
MOV BH,DL 88D7 2
MOV BH,FF B7FF 2
MOV BL,CH 88EB 2
MOV BL,CL 88CB 2
MOV BL,DH 88F3 2
MOV BL,DL 88D3 2
MOV BL,FF B3FF 2
MOV BX,AX 89C3 2
MOV BX,CS 8CCB 2
MOV BX,CX 89CB 2
MOV BX,DS 8CDB 2
MOV BX,DX 89D3 2
MOV BX,ES 8CC3 2
MOV BX,FFFF BBFFFF 3
MOV BX,SS 8CD3 2
MOV CH,DH 88F5 2
MOV CH,DL 88D5 2
MOV CH,FF B5FF 2

```

MOV CL,DH 88F1 2
MOV CL,DL 88D1 2
MOV CL,FF B1FF 2
MOV CS,AX 8EC8 2
MOV CS,BX 8ECB 2
MOV CS,CX 8EC9 2
MOV CS,DX 8ECA 2
MOV CX,AX 89C1 2
MOV CX,BX 89D9 2
MOV CX,CS 8CC9 2
MOV CX,DS 8CD9 2
MOV CX,DX 89D1 2
MOV CX,ES 8CC1 2
MOV CX,FFFF B9FFFF 3
MOV CX,SS 8CD1 2
MOV DH,FF B6FF 2
MOV DI,AX 89C7 2
MOV DI,FFFF BFFFFF 3
MOV DL,FF B2FF 2
MOV DS,AX 8ED8 2
MOV DS,BX 8EDB 2
MOV DS,CX 8ED9 2
MOV DS,DX 8EDA 2
MOV DX,AX 89C2 2
MOV DX,BX 89DA 2
MOV DX,CS 8CCA 2
MOV DX,CX 89CA 2
MOV DX,DS 8CDA 2
MOV DX,ES 8CC2 2
MOV DX,FFFF BAFFFF 3
MOV DX,SS 8CD2 2
MOV ES,AX 8EC0 2
MOV ES,BX 8EC3 2

```

MOV ES,CX 8EC1 2
MOV ES,DX 8EC2 2
MOV SI,AX 89C6 2
MOV SI,FFFF BEFFFF 3
MOV SS,AX 8ED0 2
MOV SS,BX 8ED3 2
MOV SS,CX 8ED1 2
MOV SS,DX 8ED2 2
MOVAL,[FFFF] A0FFFF 3
MOVAX,[FFFF] A1FFFF 3
MOVSB A4 1
MOVSW A5 1
MOVSB A5 1
MOV[FFFF],AL A2FFFF 3
MOV[FFFF],AX A3FFFF 3
MOV[FFFF],BP 892EFFFF 4
MOV[FFFF],BX 891EFFFF 4
MOV[FFFF],CS 8C0EFFFF 4
MOV[FFFF],CX 890EFFFF 4
MOV[FFFF],DI 893EFFFF 4
MOV[FFFF],DS 8C1EFFFF 4
MOV[FFFF],DX 8916FFFF 4
MOV[FFFF],ES 8C06FFFF 4
MOV[FFFF],SI 8936FFFF 4
MOV[FFFF],SP 8926FFFF 4
MOV[FFFF],SS 8C16FFFF 4
MUL AX F7E0 2
MUL BX F7E3 2
MUL CX F7E1 2
MUL DX F7E2 2
NEG AH F6DC 2
NEG AL F6D8 2
NEG AX F7D8 2

Appendix 5 - 8086 Mnemonic File

NEG BX F7DB 2
NEG CX F7D9 2
NEG DX F7DA 2
NOP 90 1
NOT AH F6D4 2
NOT AL F6D0 2
NOT AX F7D0 2
NOT BX F7D3 2
NOT CX F7D1 2
NOT DX F7D2 2
OR AX,BX 09D8 2
OUT DX,AX EF 1
POP AX 58 1
POP BX 5B 1
POP CS 0F 1
POP CX 59 1
POP DS 1F 1
POP DX 5A 1
POP ES 07 1
POP SS 17 1
POPF 9D 1
PUSH AX 50 1
PUSH BX 53 1
PUSH CS 0E 1
PUSH CX 51 1
PUSH DS 1E 1
PUSH DX 52 1
PUSH ES 06 1
PUSH SS 16 1
PUSHF 9C 1
REP F3 1
REPE F3 1
REPNE F2 1

REPNZ F2 1
REPZ F3 1
RET C3 1
RETF CB 1
ROL AH,1 D0C4 2
ROL AH,CL D2C4 2
ROL AL,1 D0C0 2
ROL AL,CL D2C0 2
ROL AX,1 D1C0 2
ROL AX,CL D3C0 2
ROR AH,1 D0CC 2
ROR AH,CL D2CC 2
ROR AL,1 D0C8 2
ROR AL,CL D2C8 2
ROR AX,1 D1C8 2
ROR AX,CL D3C8 2
SAHF 9E 1
SCASB AE 1
SCASB AE 1
SCASW AF 1
SCASW AF 1
SHL AH,1 D0E4 2
SHL AH,CL D2E4 2
SHL AL,1 D0E0 2
SHL AL,CL D2E0 2
SHL AX,1 D1E0 2
SHL AX,CL D3E0 2
SHR AH,1 D0EC 2
SHR AH,CL D2EC 2
SHR AL,1 D0E8 2
SHR AL,CL D2E8 2
SHR AX,1 D1E8 2
SHR AX,CL D3E8 2

Appendix 5 - 8086 Mnemonic File

SS: 36 1
STC F9 1
STD FD 1
STI FB 1
STOSB AA 1
STOSW AB 1
SUB AL,AH 28E0 2
SUB SI,DX 2BF2 2
TEST AH,BH 84FC 2
TEST AL,BL 84D8 2
TEST AX,BX 85D8 2
TEST AX,CX 85C8 2
TEST AX,DX 85D0 2
WAIT 9B 1
XCHG AX,BX 93 1
XLAT D7 1
XOR AX,AX 31C0 2
XOR AX,BX 31D8 2
XOR AX,FFFF 35FFFF 3
ADD AH,AH 00E4 2
ADD AH,AL 00C4 2
ADD AL,AH 00E0 2
ADD AL,AL 00C0 2

Appendix 6 - System Error Messages

System Error Messages

Any attempt to carry out an illegal operation can result in an error message of the type shown below:

SYSSPRGO.&07

**Process exited
Exit number 60**

Continue

Esc

Pision calls these exit conditions "System panic numbers". These numbers, which range from 0 to 255, arise when the system detects a condition which can occur when there is a "bug" in the application. Panics are always fatal and will cause the program or application to terminate immediately. They ensure that the system is protected from badly constructed applications.

Panic Number	Meaning
00	Used by test code when test fails
01	Invalid function number
02	Invalid semaphore handling
03	Semaphore not allowed
04	Initial semaphore count is invalid
05	Signal count is negative
06	Invalid function number for process manager
07	Invalid process ID
08	Task tried to create a task
09	Invalid function number for time manager

Appendix 6 - System Error Messages

10	Invalid function number for segment manager
11	Segment size was negative
12	Segment type was incorrect
13	Invalid segment handle
14	Segment copy is out of range
15	Invalid function number for heap number
16	Heap not initialised
17	A heap cell is being reduced by more than its size
18	Attempt to set heap granularity greater than allowed
19	A heap cell address is outside the boundaries of the heap
20	Invalid function number for inter-process manager
21	Inter-process messaging has already been initialised
22	Inter-process messaging has not been initialised
23	Cannot initialise with zero messages in the queue
24	Invalid function number for I/O manager
25	Invalid I/O channel
26	Device request panic
27	Invalid wait handler handle
28	Key and pointing device already hooked
29	Key and pointing device requesting process is not a task
30	Invalid function number for device manager
31	Invalid device handle
32	Invalid function number for file manager
33	Process already connected to file server
34	Reserved
35	Invalid function number for library manager
36	Invalid library handle
37	Invalid function number for library
38	Invalid LIB file channel
39	Invalid DYL index number
40	Invalid message to file server
41	Process has not connected to file server
42	Invalid function number for conversion manager
43	Invalid function for general manager
44	Attempt to unhook from notify when not already hooked

45	Invalid revector address
46	Invalid function number for conversion manager
47	Leave called before a call to enter
48	No method available to handle message
49	Invalid reclass attempt
50	Unknown category in LibHandle
51	Unknown class in LibCreate
52	Supersend called from outside a method
53	Attempt to get a handle before being linked
54	Missing external categories in LibLink
55	Object does not point to valid class
56	Invalid link layer completion code
57	Invalid function for window server
58	Invalid function number for hardware manager
59	Unexpected interrupt
60	Attempted to write outside of process data segment
61	Interrupts have been disabled for too long
62	Reserved
63	Attempt to divide by zero error
64	Overflow interrupt
65	Invalid function number for Dbf manager
66	Invalid DBF I/O channel
67	Invalid parameter for DBF function
68	Address zero overwrite
69	The system detected less than 0x100 bytes of stack remaining
70	Environment name size > EnvMaxNameSize
71	Single step interrupt (INT 1)
72	Break point interrupt (INT 3)
73	A request was made while an asynchronous request of the same type and on the same channel was already pending
74	Invalid function number for serial I/O manager
75	Call a Psion 3 function on a Psion 3A machine
76	Attempt to find a DYL not in a visible bank
77	Floating point emulator exception

Appendix 6 - System Error Messages

78	Semaphore count exceeds 0x7FFF
80	Library fatal error
255	Corrupted heap detected

Appendix 7 - Interrupts and Function Calls

SINGLE SERVICE

Character Management

INTERRUPT NUMBER: 93

PURPOSE: Character is a digit
ENTRY: AL The character to be tested
RETURN: Z flag = 0 If character is a digit
Z flag = 1 if character is not a digit

INTERRUPT NUMBER: 94

PURPOSE: Character is a hex digit
ENTRY: AL The character to be tested
RETURN: Z flag = 0 If character is a hex digit
Z flag = 1 if character is not a hexdigit

INTERRUPT NUMBER: 95

PURPOSE: Character is printable
ENTRY: AL The character to be tested
RETURN: Z flag = 0 If character is a printable
Z flag = 1 if character is not a printable

INTERRUPT NUMBER: 96

PURPOSE: Character is alphabetic
ENTRY: AL The character to be tested
RETURN: Z flag = 0 If character is a alphabetic
Z flag = 1 if character is not a alphabetic

INTERRUPT NUMBER: 97

PURPOSE: Character is alphabetic a or digit
ENTRY: AL The character to be tested
RETURN: Z flag = 0 If character is alphabetic
Z flag = 1 if character is not alphabetic

INTERRUPT NUMBER: 98

PURPOSE: Character is a upper case
ENTRY: AL The character to be tested
RETURN: Z flag = 0 If character is upper case
Z flag = 1 if character is not upper case

INTERRUPT NUMBER: 99

PURPOSE: Character is lower case
ENTRY: AL The character to be tested
RETURN: Z flag = 0 If character is lower case
Z flag = 1 if character is not lower case

INTERRUPT NUMBER: 9A

PURPOSE: Character is a space
ENTRY: AL The character to be tested
RETURN: Z flag = 0 If character is space
Z flag = 1 if character is not space

INTERRUPT NUMBER: 9B

PURPOSE: Character is punctuation
ENTRY: AL The character to be tested
RETURN: Z flag = 0 If character is punctuation
Z flag = 1 if character is not punctuation

INTERRUPT NUMBER: 9C

PURPOSE: Character is graphic
ENTRY: AL The character to be tested
RETURN: Z flag = 0 If character is graphic
Z flag = 1 if character is not graphic

INTERRUPT NUMBER: 9D

PURPOSE: Character is control
ENTRY: AL The character to be tested
RETURN: Z flag = 0 If character is control
Z flag = 1 if character is not control

INTERRUPT NUMBER: 9E

PURPOSE: Character to upper case
ENTRY: AL A character to be converted
AH A character to be converted
ENTRY: AL Converted to upper case
AH Converted to upper case

INTERRUPT NUMBER: 9F

PURPOSE: Character to lower case
ENTRY: AL A character to be converted
AH A character to be converted
ENTRY: AL Converted to lower case
AH Converted to lower case

INTERRUPT NUMBER: A0

PURPOSE: Character to folded characters
ENTRY: AL A character to be folded
AH A character to be folded
ENTRY: AL Folded
AH Folded

Buffer Management

INTERRUPT NUMBER: A1

PURPOSE: Copy one buffer to another
ENTRY: DS:SI Pointer to source buffer
ES:DI Pointer to destination buffer
CX Number of bytes to copy
RETURN: None

INTERRUPT NUMBER: A2

PURPOSE: Swap the contents of two buffers
ENTRY: DS:SI Pointer to one of the buffers
ES:DI Pointer to the other buffer
CX Number of bytes to swap
RETURN: None

INTERRUPT NUMBER: A3

PURPOSE: Compare one buffer with another
ENTRY: DS:SI Pointer to left operand buffer
ES:DI Pointer to right operand buffer
CX Number of bytes in left operand
BX Number of bytes in right operand
RETURN: Flags < 0 if [SI] < [DI]
Flags = 0 if [SI] = [DI]
Flags > 0 if [SI] > [DI]

INTERRUPT NUMBER: A4

PURPOSE: Compare one buffer with another folded
ENTRY: DS:SI Pointer to left operand buffer
ES:DI Pointer to right operand buffer
CX Number of bytes in left operand
BX Number of bytes in right operand
RETURN: Flags < 0 if [SI] < [DI]
Flags = 0 if [SI] = [DI]
Flags > 0 if [SI] > [DI]

INTERRUPT NUMBER: A5

PURPOSE: Match a wild card buffer
ENTRY: DS:SI Pointer to buffer to be searched
ES:DI Pointer to wild card match buffer
CX Length of buffer to be searched
DX Length of match buffer
RETURN: Carry clear

INTERRUPT NUMBER: A6

PURPOSE: Match a wild card buffer folded
ENTRY: DS:SI Pointer to buffer to be searched
ES:DI Pointer to wild card match buffer
CX Length of buffer to be searched
DX Length of match buffer
RETURN: Carry clear

INTERRUPT NUMBER: A7

PURPOSE:	Locate a character in a buffer
ENTRY:	DS:SI Pointer to buffer to be searched
	AH The character to be located
	CX Length of buffer to be searched
RETURN:	AX Index of the character in the string

INTERRUPT NUMBER: A8

PURPOSE:	Locate a character in a buffer folded
ENTRY:	DS:SI Pointer to buffer to be searched
	AH The character to be located
	CX Length of buffer to be searched
RETURN:	AX Index of the character in the string

INTERRUPT NUMBER: A9

PURPOSE:	Find a sub buffer in a buffer
ENTRY:	DS:SI Pointer to buffer to be searched
	ES:DI Pointer to the buffer to be located
	CX Length of buffer to be searched
	BX The number of bytes in the sub buffer
RETURN:	AX Offset of the sub buffer within the buffer

Appendix 7 - Interrupts and Function Calls

INTERRUPT NUMBER: AA

PURPOSE:	Find a sub buffer in a buffer folded
ENTRY:	DS:SI Pointer to buffer to be searched
	ES:DI Pointer to the buffer to be located
	CX Length of buffer to be searched
	BX The number of bytes in the sub buffer
RETURN:	AX Offset of the sub buffer within the buffer

INTERRUPT NUMBER: AB

PURPOSE:	Justify a buffer
ENTRY:	DS:SI Pointer to buffer to be justified
	ES:DI Pointer to the target buffer
	CX Length of the buffer to be justified
	BX Length of the target buffer
	DL Justify, left, centre, right
	DH The fill character
RETURN:	AX Points to the character after the last byte copied to the target buffer

String Management

INTERRUPT NUMBER: AC

PURPOSE:	Copy one string to another
ENTRY:	DS:SI Pointer to source string
	ES:DI Pointer to target string
RETURN:	None

INTERRUPT NUMBER: AD

PURPOSE: Copy one string to another folded
 ENTRY: DS:SI Pointer to source string
 ES:DI Pointer to target string
 RETURN: None

INTERRUPT NUMBER: AE

PURPOSE: Convert a string to folded
 ENTRY: DS:SI Pointer to string to be folded
 RETURN: None

INTERRUPT NUMBER: AF

PURPOSE: Compare one string with another
 ENTRY: DS:SI Pointer to left operand string
 ES:DI Pointer to right operand string
 RETURN: Flags < 0 If[SI] < [DI]
 Flags = 0 If[SI] = [DI]
 Flags > 0 If[SI] > [DI]

INTERRUPT NUMBER: B0

PURPOSE: Compare one string with another folded
 ENTRY: DS:SI Pointer to left operand string
 ES:DI Pointer to right operand string
 RETURN: Flags < 0 If[SI] < [DI]
 Flags = 0 If[SI] = [DI]
 Flags > 0 If[SI] > [DI]

INTERRUPT NUMBER: B1

PURPOSE: Match a wild card string
ENTRY: DS:SI Pointer to the string to be searched
ES:DI Pointer to the wild card match string
RETURN: Carry clear = match

INTERRUPT NUMBER: B2

PURPOSE: Match a wild card string folded
ENTRY: DS:SI Pointer to the string to be searched
ES:DI Pointer to the wild card match string
RETURN: Carry clear = match

INTERRUPT NUMBER: B3

PURPOSE: Locate a character within string
ENTRY: DS:SI Pointer to the string to be searched
AH The character to be located
RETURN: AX Index of the character in the string

INTERRUPT NUMBER: B4

PURPOSE: Locate a character within string folded
ENTRY: DS:SI Pointer to the string to be searched
AH The character to be located
RETURN: AX Index of the character in the string

INTERRUPT NUMBER: B5

PURPOSE:	Locate a character in reverse	
ENTRY:	DS:SI	Pointer to the string to be searched
	AH	The character to be located
RETURN:	AX	Index of the character in the string

INTERRUPT NUMBER: B6

PURPOSE:	Locate a character in reverse folded	
ENTRY:	DS:SI	Pointer to the string to be searched
	AH	The character to be located
RETURN:	AX	Index of the character in the string

INTERRUPT NUMBER: B7

PURPOSE:	Find a substring in a string	
ENTRY:	DS:SI	Pointer to the string to be searched
	ES:DI	Pointer to the string to be located
RETURN:	AX	Offset of the substring in the string

INTERRUPT NUMBER: B8

PURPOSE:	Find a substring in a string folded	
ENTRY:	DS:SI	Pointer to the string to be searched
	ES:DI	Pointer to the string to be located
RETURN:	AX	Offset of the substring in the string

INTERRUPT NUMBER: B9

PURPOSE:	Length of a string
ENTRY:	ES:DI Pointer to the string whose length is to be found
RETURN:	AX Length of the string

INTERRUPT NUMBER: BA

PURPOSE:	Validate a system name
ENTRY:	AL Max number of characters in the name allowed in the name
	AH Non 0 = An extension is valid 0 = An extension is invalid
	ES:DI Pointer to the string to be validated
	ES:DI Pointer to the string to be located
RETURN:	AX Offset of the substring in the string

Long Integer Management

INTERRUPT NUMBER: BB

PURPOSE: Compare two long integers
ENTRY: AX:BX The left operand long integer
CX:DX The right operand long integer
RETURN: Flags are as 8086 compare, CMP AX,BX etc

INTERRUPT NUMBER: BC

PURPOSE: Long integer multiplication
ENTRY: AX:BX The left operand long integer
CX:DX The right operand long integer
RETURN: AX:BX Product

INTERRUPT NUMBER: BD

PURPOSE: Long integer division
ENTRY: AX:BX The left operand long integer (dividend)
CX:DX The right operand long integer (divisor)
RETURN: AX:BX Quotient
CX:DX Remainder

INTERRUPT NUMBER: BE

PURPOSE: Compare two unsigned long integers
ENTRY: AX:BX The left operand unsigned long integer
CX:DX The right operand unsigned long integer
RETURN: Flags are as 8086 compare, CMP AX,BX etc

INTERRUPT NUMBER: BF

PURPOSE: Unsigned long integer multiplication
ENTRY: AX:BX The left operand unsigned long integer
CX:DX The right operand unsigned long integer
RETURN: AX:BX Product

INTERRUPT NUMBER: C0

PURPOSE: Unsigned Long integer division
ENTRY: AX:BX The left operand unsigned long integer
(dividend)
CX:DX The right operand unsigned long integer
(divisor)
RETURN: AX:BX Quotient
CX:DX Remainder

Floating Point Number Management

INTERRUPT NUMBER: C2

PURPOSE:	Subtract floats
ENTRY:	DI Pointer to left hand (and destination) floating point operand
	SI Pointer to right hand floating point operand

INTERRUPT NUMBER: C3

PURPOSE:	Multiply two floats
ENTRY:	DI Pointer to left hand (and destination) floating point operand
	SI Pointer to right hand floating point operand

INTERRUPT NUMBER: C4

PURPOSE:	Divide two floats
ENTRY:	DI Pointer to left hand (and destination) floating point operand
	SI Pointer to right hand floating point operand

INTERRUPT NUMBER: C5

PURPOSE:	Compare two floats
ENTRY:	DI Pointer to left hand point operand
	SI Pointer to right hand floating point operand

INTERRUPT NUMBER: C6

PURPOSE: Negate a float
ENTRY: DI Pointer to a floating point operand

INTERRUPT NUMBER: C7

PURPOSE: Convert float to signed integer
ENTRY: SI Pointer float operand to be converted
RETURN: AX Unsigned integer result

INTERRUPT NUMBER: C8

PURPOSE: Convert float to unsigned integer
ENTRY: SI Pointer float operand to be converted
RETURN: AX Integer result

INTERRUPT NUMBER: C9

PURPOSE: Convert float to signed long
ENTRY: SI Pointer float operand to be converted
RETURN: AX:BX Long integer result

INTERRUPT NUMBER: CA

PURPOSE: Convert float to unsigned long
ENTRY: SI Pointer float operand to be converted
RETURN: AX:BX Unsigned long integer result

INTERRUPT NUMBER: CB

PURPOSE: Convert signed integer to float
ENTRY: AX Signed integer to convert
DI Pointer to destination float
RETURN: [DI]

INTERRUPT NUMBER: CC

PURPOSE: Convert unsigned integer to float
ENTRY: AX Unsigned integer to convert
DI Pointer to destination float
RETURN: [DI]

INTERRUPT NUMBER: CD

PURPOSE: Convert signed long to float
ENTRY: AX:BX Signed long to convert
DI Pointer to destination float
RETURN: [DI]

INTERRUPT: CF

TITLE: LibSend
PURPOSE: Send a message
ENTRY: BX The handle of the object to receive the message
CL The message number to be sent
DX Argument 1
SI Argument 2
DI Argument 3
RETURN: AX Whatever the method returns
COMMENTS: This service will send a message to an object.

INTERRUPT: D0

TITLE: LibSendSuper
PURPOSE: Send a superclass message
ENTRY: BX The handle of the object to receive the message
CL The message number to be sent
DX Argument 1
SI Argument 2
DI Argument 3
RETURN: AX Whatever the method returns
COMMENTS: This service will send a message to an object.

INTERRUPT NUMBER: D5

TITLE: GenIntByNumber
PURPOSE: Interrupt by number service
ENTRY: AL The interrupt number
DS:SI Pointer to the input register values
DS:DI Pointer to the output register values
RETURN: AX The flags register after the call

COMMENTS: This service can be used to call any software interrupt from machine code. The register values are stored sequentially as 6 words (2 bytes) and represent the values for AX, BX, CX, DX, SI, and DI. BP is not needed by the operating. DS:SI and DS:DI can point to the same memory location. The value returned is the flags register because although the carry flag is the most important, some of the operating system routines also set the arithmetic flags. The carry flag is in bit 0 of the returned results in AX.

INTERRUPT: D9

TITLE: LibEnterSend
PURPOSE: Send a message under protection of error
ENTRY: BX The handle of the object to receive the message
CL The message number to be sent
DX Argument 1
SI Argument 2
DI Argument 3
RETURN: AX Whatever the method returns

COMMENTS: This service will send a message to an object.

INTERRUPT NUMBER: DB

PURPOSE:	Capitalise a string
ENTRY:	DS:SI Pointer to string to be capitalised
RETURN:	None

MULTI SERVICE

Heap Memory Management - INT 81

FUNCTION NUMBER: 00

TITLE: HeapAllocate Cell
PURPOSE: Allocate heap memory
ENTRY: CX Size of cell to be allocated in bytes
RETURN: AX Base of cell

COMMENTS: Allocates a cell in the process' heap memory

FUNCTION NUMBER: 01

TITLE: HeapReAllocateCell
PURPOSE: Re-allocate heap memory
ENTRY: BX Base of cell to be re-allocated or zero.
CX The new size of the re-allocated cell in bytes.
RETURN: AX Base of the re-allocated cell

COMMENTS: The re-allocated cell can be made smaller or larger.

FUNCTION NUMBER: 02

TITLE: HeapAdjustCellSize
PURPOSE: Adjust the size of the heap
ENTRY: BX Base of cell to be adjusted
CX Adjustment to the size of the cell in bytes
DX Offset in the cell to make the adjustment
RETURN: AX Base of cell after adjusting

COMMENTS: Adjust the size of the cell at BX, at an offset DX in the cell, by CX bytes.

FUNCTION NUMBER: 03

TITLE: HeapFreeCell
PURPOSE: Free heap memory
ENTRY: BX Base of cell
RETURN: None

COMMENTS: Free the memory cell whose base is in the BX register.

FUNCTION NUMBER: 04

TITLE: HeapCellSize
PURPOSE: Size of cell heap
ENTRY: BX Pointer to the base of the cell
RETURN: AX Size of cell in bytes

COMMENTS: Returns the cell size in memory.

FUNCTION NUMBER: 05

TITLE: HeapSetGranularity
PURPOSE: Set the heap granularity
ENTRY: BX The new granularity in paragraphs (16 bytes)
RETURN: None

COMMENTS: Set the heap granularity parameter to the value in BX

FUNCTION NUMBER: 06

TITLE: HeapFreeMemory
PURPOSE: Size of available heap memory
ENTRY: None
RETURN: AX Number of bytes available in heap
BX Address of the base of heap memory

COMMENTS: Returns the size of potentially available heap memory and the address of the heap memory. Due to the multi-tasking system, the returned value should be treated with some caution.

Library Management - INT 84

FUNCTION NUMBER: 00

TITLE: LibLoad
PURPOSE: Load a dynamic Library
ENTRY: ES:BX Pointer to the Dynamic Library file name
CL Zero -Don't link the library automatically
Non-Zero - Link the library after loading
RETURN: AX The Dynamic Library handle
COMMENTS: Loads a Dynamic Library into memory for access by
other LIB services

FUNCTION NUMBER: 01

TITLE: LibUnLoad
PURPOSE: Unload a Dynamic Library
ENTRY: BX Handle for the Dynamic Library
RETURN: Carry Set: OK, Carry Cleared: Library not opened.
COMMENTS: Unloads a Dynamic Library from memory

FUNCTION NUMBER: 02

TITLE: LibLink
PURPOSE: Link a Dynamic Library
ENTRY: BX Handle of the Dynamic Library
RETURN: None

COMMENTS: Links a Dynamic Library which is already in memory.

FUNCTION NUMBER: 03

TITLE: LibFind
PURPOSE: Get a Dynamic Library
ENTRY: ES:BX Pointer to the Dynamic library internal
name.
RETURN: AX The Dynamic Library handle

COMMENTS: The library must exist on the ROM.

FUNCTION NUMBER: 04

TITLE: LibHandle
PURPOSE: Get a DYL handle by number
ENTRY: BX The number of the required external category
RETURN: AX The category handle

COMMENTS: Get a DYL handle by number

FUNCTION NUMBER: 05

TITLE: LibCreate
PURPOSE: Create an object by Category Number
ENTRY: BX The number of the category
CX The class number within the category
RETURN: AX The object handle

COMMENTS: This service converts the category number passed in BX to a handle.

FUNCTION NUMBER: 06

TITLE: LibCreate
PURPOSE: Create an object by Category Name
ENTRY: BX The number of the category
CX The class number within the category
RETURN: AX The object handle

COMMENTS: This service converts the category name passed in BX to a handle.

FUNCTION NUMBER: 07

TITLE: LibDestroy
PURPOSE: Destroy an object
ENTRY: BX The handle of the object to be destroyed
RETURN: None

COMMENTS: De-allocate all the memory associated with an object including any compound components.

FUNCTION NUMBER: 08

TITLE: LibCopy
PURPOSE: Copy data from a category
ENTRY: BX The category number
CX Number of bytes to copy
DI Pointer to a buffer to receive the data
SI Offset of the source in the category segment.
RETURN: None

COMMENTS: Copy data from a category segment into the data segment of the current process.

FUNCTION NUMBER: 09

TITLE: LibOpen
PURPOSE: Open a multi library file
ENTRY: ES:BX Pointer to the file name
RETURN: AX The open multi library file handle

COMMENTS: This service opens a file which contains multiple libraries

FUNCTION NUMBER: 0A

TITLE: LibLoadFile
PURPOSE: Load a multiple dynamic library
ENTRY: BX Multiple library file handle
CH The index of the library to be loaded
CL Zero -don't link the library automatically
Non zero - Link the library automatically

RETURN: Carry

COMMENTS: The process must be connected to the file server before calling this function.

FUNCTION NUMBER: 0B

TITLE: LibReClass
PURPOSE: Reclass an object by number
ENTRY: BX The number of the category
CX The number of the class within the category
DI Pointer to the object to be reclassified
RETURN: None
COMMENTS: Converts the category number passed in BX to a handle.

FUNCTION NUMBER: 0C

TITLE: LibReClassByHandle
PURPOSE: Reclass an object by handle
ENTRY: BX The handle of the category
CX The number of the class within the category
DI Pointer to the object to be reclassified
RETURN: None
COMMENTS: Changes the class to which the object belongs to that specified by BX and CX.

Input Output Management - INT 86

FUNCTION NUMBER: 00

TITLE: IoAsynchronous
PURPOSE: Asynchronous I/O
ENTRY: AL I/O number
BX I/O handle
DS: CX Pointer to argument 1
DS: DX Pointer to argument 2
DS: DI Pointer to argument 3
RETURN: AX Result from the device driver
COMMENTS: Requests I/O services from device drivers

FUNCTION NUMBER: 01

TITLE: IoAsynchronousNoError
PURPOSE: Asynchronous I/O - no error reporting
ENTRY: AL I/O number
BX I/O handle
DS: CX Pointer to argument 1
DS: DX Pointer to argument 2
DS: DI Pointer to argument 3
RETURN: None
COMMENTS: Requests I/O services from device drivers

FUNCTION NUMBER: 02

TITLE: IoWithWait
PURPOSE: Synchronous I/O
ENTRY: AL I/O number
BX I/O handle
DS: CX Pointer to argument 1
DS: DX Pointer to argument 2
RETURN: AX Result from the device driver

COMMENTS: Requests I/O services from device drivers
synchronously

FUNCTION NUMBER: 03

TITLE: IoRoot
PURPOSE: Chain to root device
ENTRY: DS: BX I/O handle
DS: SI Pointer to the request package
RETURN: AX Result from the device driver

COMMENTS:

FUNCTION NUMBER: 04

TITLE: IoSuper
PURPOSE: Chain to superclass device

ENTRY: DS: BX I/O handle
DS: SI Pointer to the request package

Appendix 7 - Interrupts and Function Calls

RETURN: AX Result from the device driver

COMMENTS:

FUNCTION NUMBER: 05

TITLE: IoWaitForSignal
PURPOSE: Wait for I/O completion
ENTRY: None
RETURN: None

COMMENTS: Wait for I/O completion

FUNCTION NUMBER: 06

TITLE: IoWaitForStatus
PURPOSE: Wait for specific request to complete
ENTRY: DS:DI Pointer to status word of I/O request
RETURN: None

COMMENTS: Wait for specific I/O request to be completed

FUNCTION NUMBER: 07

TITLE: IoYield
PURPOSE: Poll for completion
ENTRY: None
RETURN: None

COMMENTS: Update the status word of any I/O requests which may have been completed

FUNCTION NUMBER: 08

TITLE: IoSignal
PURPOSE: Signal completion
ENTRY: None
RETURN: None

COMMENTS: Signals the completion of an outstanding I/O request to the current process

FUNCTION NUMBER: 09

TITLE: IoSignalByPid
PURPOSE: Signal completion by process ID
ENTRY: BX The process ID to be signalled
RETURN: Carry clear

COMMENTS: Signals the completion of an asynchronous outstanding I/O request to the current process whose Id is in the BX register

FUNCTION NUMBER: 0A

TITLE: IoSignalByPidNoReSched
PURPOSE: Signal completion, no rescheduling
ENTRY: BX The process ID to be signalled
RETURN: Carry clear

COMMENTS: Signals the completion of an asynchronous outstanding I/O request to the current process whose Id is in the BX register

FUNCTION NUMBER: 0B

TITLE: IoAddHandler
PURPOSE: Add a handle
ENTRY: AL The vector number of the handle
BX Identifying data to be passed to the handler
DS,ES Must be pointing at the user process data space
RETURN: AX Handler handle

COMMENTS:

FUNCTION NUMBER: 0C

TITLE: IoRemoveHandler
PURPOSE: Remove a handler
ENTRY: BX The handler handle

DS,ES Must be pointing at the user process data
space
RETURN: None

COMMENTS:

FUNCTION NUMBER: 0D

TITLE: IoEnableHandler
PURPOSE: Enable/Disable a handler
ENTRY: BX The handler handle
CL 0 - To disable the handler
1 - To enable the handler
DS,ES Must be pointing at the user process data
space
RETURN: None

COMMENTS:

FUNCTION NUMBER: 0E

TITLE: IoRequestReset
PURPOSE: Request a reset
ENTRY: BX Device handle
CX Identifying data
RETURN: None

COMMENTS:

FUNCTION NUMBER: 0F

TITLE: IoRequestResetCancel
PURPOSE: Cancel a requested reset
ENTRY: BX Device handle
CX Identifying data
RETURN: None

COMMENTS:

FUNCTION NUMBER: 00

TITLE: IoOpen
PURPOSE: Open a device
ENTRY: ES:BX Pointer to the device name
CX Open mode
DX If an attached driver is being opened, the handle of the device
RETURN: AX I/O handle

COMMENTS: The name of the device is zero terminated string

FUNCTION NUMBER: 10

TITLE: IoClose
PURPOSE: Close a device
ENTRY: BX I/O handle
RETURN: AX Result from the device driver
COMMENTS: Close an open channel

FUNCTION NUMBER: 11

TITLE: IoRead
PURPOSE: Read from a device
ENTRY: BX I/O handle
DS: CX Pointer to a buffer to receive the data
DX Number of bytes to read
RETURN: AX Number of bytes actually read
COMMENTS: Read data from the device

FUNCTION NUMBER: 12

TITLE: IoWrite
PURPOSE: Write to a device
ENTRY: BX I/O handle
DS: CX Pointer to a buffer to be written
DX Number of bytes to write
RETURN: AX Result from device driver
COMMENTS: Write data to the device

FUNCTION NUMBER: 13

TITLE: IoSeek
PURPOSE: Seek on a device
ENTRY: BX I/O handle
CX Varies options (see dev manual)
DX Pointer to the double word containing the new position
RETURN: AX Result from device driver
COMMENTS: Performs a seek on a device

FUNCTION NUMBER: 14

TITLE: IoKeyWithWait
PURPOSE: Keyboard
ENTRY: DS:BX Pointer to a KeyEnt structure
RETURN: None
COMMENTS: System maintains a queue of upto 16 keyboard events in a buffer. Event data copied to structure at BX

FUNCTION NUMBER: 15

TITLE: IoAddApplicationHandler
PURPOSE: Add an application handler
ENTRY: BX Identifying data to be passed to the handler
CX Address of the application handler
DX Address of the application handler dispatcher
DS,ES Must point to the user process data space
RETURN: Carry set

FUNCTION NUMBER: 16

TITLE: IoRemoveApplicationHandler
PURPOSE: Remove an application handler
ENTRY: BX The handler handle
DS,ES Must point to the user process data space
RETURN: None

COMMENTS:

FUNCTION NUMBER: 17

TITLE: IoEnableApplicationHandler
PURPOSE: Enable/Disable application handler
ENTRY: BX The handler handle
CL 0 - Disable the handler
CL 1 - Enable the handler
DS,ES Must point to the user process data space
RETURN: None

COMMENTS:

FUNCTION NUMBER: 18

TITLE: IoShiftStates
PURPOSE: Get the shift states
ENTRY: None
RETURN: AX The current shift states

COMMENTS:

Appendix 7 - Interrupts and Function Calls

FUNCTION NUMBER: 19

TITLE: IoWaitForSignalNoHandler
PURPOSE: Wait for I/O completion no handlers
ENTRY: None
RETURN: None

COMMENTS:

FUNCTION NUMBER: 1A

TITLE: IoSignalKillAsynchronous
PURPOSE: Request signal from supervisor
ENTRY: BX The process ID
DI The status word to be cleared on completion
RETURN: Carry clear on success

COMMENTS:

FUNCTION NUMBER: 1B

TITLE: IoSignalKillCancel
PURPOSE: Cancel requested signal from supervisor
ENTRY: BX The process ID
RETURN: Carry clear on success

COMMENTS:

FUNCTION NUMBER: 1D

TITLE: IoNextHalfSecond
PURPOSE: Request signal on next half second
ENTRY: None
RETURN: None

COMMENTS:

FUNCTION NUMBER: 1F

TITLE: IoPlaySoundW
PURPOSE: Play back a sound file synchronously
ENTRY: BX Pointer to the sound file name
CX Duration in ticks
DX Sound volume

RETURN:

COMMENTS:

FUNCTION NUMBER: 1E

TITLE: IoPlaySoundA
PURPOSE: Play back a sound file synchronously
ENTRY: BX Pointer to the sound file name
CX Duration in ticks
DI Pointer to the status word
DX Sound volume

RETURN: None

COMMENTS:

FUNCTION NUMBER: 20

TITLE: IoPlaySoundCancel
PURPOSE: Cancel play back
ENTRY: None
RETURN: None

COMMENTS:

FUNCTION NUMBER: 22

TITLE: IoRecordSoundW
PURPOSE: Record sound to a file synchronously
ENTRY: BX Pointer to the sound file name
CX Sound file length in 2048-byte units
DX Sound volume
RETURN:

COMMENTS: Not to SSD. CX number of bytes recorded excluding
32 byte header

FUNCTION NUMBER: 21

TITLE: IoRecordSoundA
PURPOSE: Record sound to a file asynchronously
ENTRY: BX Pointer to the sound file name
CX Sound file length in 2048-byte units
DI Pointer to the statues word
RETURN: None

COMMENTS: Not to SSD. CX number of bytes recorded excluding
32 byte header

FUNCTION NUMBER: 23

TITLE: IoRecordSoundCancel
PURPOSE: Cancel recording sound to a file
ENTRY: None
RETURN: None

COMMENTS:

Date and Time Management - INT 89

FUNCTION NUMBER: 00

TITLE: TimeSleepForTenths
PURPOSE: Sleep for tenths of second
ENTRY: CX:DX The time to sleep in tenths of second
RETURN: Carry clear on success

COMMENTS: CX:DX = 32 bit integer

FUNCTION NUMBER: 01

TITLE: TimSleepForTicks
PURPOSE: Sleep for system clock ticks
ENTRY: CX:DX The time to sleep in system ticks
RETURN: Carry set on success

COMMENTS: CX:DX is a 32 bit integer

FUNCTION NUMBER: 02

TITLE: TimGetSystemTime
PURPOSE: Get the system time
ENTRY: None
RETURN: AX:BX System time in seconds

COMMENTS: Time since 1/1/70 at 00:00:00. BX=least, AX=most

FUNCTION NUMBER: 03

TITLE: TimSetSystemTime
PURPOSE: Set the system time
ENTRY: CX:DX The new system time
RETURN: None

COMMENTS: Enter number of seconds since 1/1/70 at 00:00:00

FUNCTION NUMBER: 08

TITLE: TimDaysInMonth
PURPOSE: Number of days in month
ENTRY: CH The month number, 0 - 11 (0 = Jan)
CL The year number (0 = 1900)
RETURN: AX Number of days in month

COMMENTS:

FUNCTION NUMBER: 09

TITLE: TimDayOfWeek
PURPOSE: Week day number
ENTRY: CX:DX The number of days since 1900
RETURN: AX The week day number, 0 = Monday

COMMENTS:

FUNCTION NUMBER: 0A

TITLE: TimNameOfDay
PURPOSE: Name of the day
ENTRY: AL The day of week number, 0 to 6
DS:BX Pointer to a buffer to receive the name of
the day
RETURN: Carry clear on success

COMMENTS:

FUNCTION NUMBER: 0B

TITLE: TimNameOfMonth
PURPOSE: Name of the month
ENTRY: AL The month number, 0 to 11
DS:BX Pointer to a buffer to receive the name of
the month
RETURN: Carry clear on success

COMMENTS:

FUNCTION NUMBER: 0D

TITLE: TimWeekNumber
PURPOSE: Week number
ENTRY: CX:DX The number of days since 1/1/1900
RETURN: AX The week number in the range 1 - 53

COMMENTS:

FUNCTION NUMBER: 0E

TITLE: TimNameOfDayAbb
PURPOSE: Abbreviated name of the day
ENTRY: AL The day of the week number, 0 to 6
BX Pointer to a buffer to receive the name of the day
RETURN: Carry clear on success

COMMENTS:

FUNCTION NUMBER: 0F

TITLE: TimNameOfMonthAbb
PURPOSE: Abbreviated name of month
ENTRY: AL The month number, 0 to 11
BX Pointer to a buffer to receive the name of the month
RETURN: Carry clear on success

COMMENTS:

Conversion Management - INT 8A

FUNCTION NUMBER: 00

TITLE: CovUnsignedIntToBuffer
PURPOSE: Unsigned integer to buffer
ENTRY: BX Number to be converted
CX The conversion radix
ES:DI Pointer to a buffer to receive the converted number
RETURN: AX The number of characters written to DI
COMMENTS: CX=2 for binary, CX=8 for Octal, CX=0A for decimal and CX=10 for hexadecimal

FUNCTION NUMBER: 01

TITLE: CovUnsignedLongIntToBuffer
PURPOSE: Unsigned long integer to buffer
ENTRY: BX Number to be converted
CX The conversion radix
ES:DI Pointer to a buffer to receive the converted number
RETURN: AX The number of characters written to DI
COMMENTS: CX=2 for binary, CX=8 for Octal, CX=0A for decimal and CX=10 for hexadecimal

FUNCTION NUMBER: 02

TITLE: ConvIntToBuffer
PURPOSE: Integer to buffer
ENTRY: BX The number to be converted
ES:DI Pointer to buffer to receive the converted number
RETURN: AX The number of characters written to DI
COMMENTS: If BX is negative, then the first character in the buffer will be a minus sign

FUNCTION NUMBER: 03

TITLE: ConvLongIntToBuffer
PURPOSE: Long Integer to Buffer
ENTRY: DX:BX Number to be converted
ES:DI Pointer to buffer to receive the converted number
RETURN: AX The number of characters written to DI
COMMENTS: Converts a long integer in DX:BX to its corresponding digits in the buffer pointed to by DI

FUNCTION NUMBER: 04

TITLE: ConvArgumentsToBuffer
PURPOSE: Convert arguments to buffer
ENTRY: SS:BX Pointer to the argument list
ES:DI Pointer to the buffer to receive the converted arguments
DS:SI Pointer to format control string
RETURN: AX The number of characters written to ES:DI
COMMENTS: Converts the values in the list pointed to by BX into a buffer at DI

FUNCTION NUMBER: 05

TITLE: ConvStringToUnsignedInt
PURPOSE: String to unsigned integer
ENTRY: CX Radix for conversion
DS:SI Pointer to zero terminated string to be converted
RETURN: AX Resultant unsigned integer
SI Pointer to the first unused character in the string
COMMENTS: Convert a zero terminated string to an unsigned integer using the radix specified in CX: 2=binary, 8=octal etc

FUNCTION NUMBER: 06

TITLE: ConvStringToUnsignedLongInt
PURPOSE: String to unsigned long integer
ENTRY: CX Radix for conversion
DS:SI Pointer to zero terminated string to be converted
RETURN: AX:BX Resultant unsigned integer
SI Pointer to the first unused character in the string

COMMENTS: Convert a zero terminated string to an unsigned integer using the radix specified in CX: 2=binary, 8=octal etc. BX = least significant word, AX = most significant word.

FUNCTION NUMBER: 07

TITLE: ConvStringToInt
PURPOSE: String to integer
ENTRY: DS:SI Pointer to zero terminated string to be converted
RETURN: AX Resultant integer
SI Pointer to the first unused character in the string

COMMENTS: Uses radix 10. The string can begin with a minus character "-", in which case the integer will be a negative or a plus character "+", in which case the integer will be positive. The maximum positive value is 32,767 and the maximum negative value is -32,768.

FUNCTION NUMBER: 08

TITLE: ConvStringToLongInt
PURPOSE: String to long integer
ENTRY: DS:SI Pointer to zero terminated string to be converted
RETURN: AX:BX Resultant long integer
SI Pointer to the first unused character in the string

COMMENTS: Uses radix 10. The string can begin with a minus character "-", in which case the long integer will be a negative or a plus character "+", in which case the long integer will be positive. The maximum positive value is 4,294,967,295 and the maximum negative value is -4,294,967,295.

FUNCTION NUMBER: 09

TITLE: ConvFloatToBuffer
PURPOSE: Floating point number to buffer
ENTRY: SI Pointer to the float to be converted
DX Pointer to the format structure
DI Pointer to the buffer to receive the converted float
RETURN: AX Length of the converted string

COMMENTS: Converts the double floating point number to a printable ASCII zero terminated string pointed to by DI using the OS specification.

FUNCTION NUMBER: 0A

TITLE: ConvStringToFloat

PURPOSE: String to float

ENTRY: SI Address of pointer to text to be converted

DX Decimal point character

DI Pointer to double destination

RETURN: Carry clear on success

COMMENTS: Scans the string for a number and writes the value as a double floating point to [DI]. Format = $[+|-]<int>.<frac>[E|e][+|-]<exp>$
(1e-99 to 1e+99)

General Management - INT 8B

FUNCTION NUMBER: 00

TITLE: GenVersion
PURPOSE: Get the operating system version number
ENTRY: None
RETURN: AX The operating system version number

COMMENTS: 16 bit integer returned

FUNCTION NUMBER: 01

TITLE: GenLCDType
PURPOSE: Get the system LCD type
ENTRY: None
RETURN: AL The LCD type

COMMENTS:

FUNCTION NUMBER: 02

TITLE: GenStartReason
PURPOSE: Get the system cold start reason
ENTRY: None
RETURN: AL The cold start reason

COMMENTS:

Appendix 7 - Interrupts and Function Calls

FUNCTION NUMBER: 03

TITLE: GenParse
PURPOSE: Generic file name parser
ENTRY: BX pointer to a GenParseEnt structure
RETURN:

COMMENTS:

FUNCTION NUMBER: 04

TITLE: LongUnsignedIntRandom
PURPOSE: Unsigned long integer random number
ENTRY: DS:BX Pointer to unsigned long integer seed
RETURN: AX:BX The unsigned long integer random number

COMMENTS:

FUNCTION NUMBER: 05

TITLE: GenGetCountryData
PURPOSE: Get the country data
ENTRY: BX Pointer to a CDataEnt structure
RETURN: None

COMMENTS:

FUNCTION NUMBER: 06

TITLE: GenGetErrorText
PURPOSE: Get error text
ENTRY: AL The error number

BX Pointer to the buffer to receive the error text
RETURN: None

FUNCTION NUMBER: 07

TITLE: GenGetOSData
PURPOSE: Get the OS data
ENTRY: DI Pointer to a buffer to receive the data
SI Offset in the operating system data space
CX The number of bytes to be copied

RETURN: None

COMMENTS:

FUNCTION NUMBER: 08

TITLE: GenDeferredMode
PURPOSE: Set deferred mode
ENTRY: AL Deferred mode
RETURN: None

COMMENTS:

FUNCTION NUMBER: 09

TITLE: GenNotify
PURPOSE: Notify by text
ENTRY: BX Pointer to the first message
CX Pointer to the second message or zero
DX Pointer to the first option or zero
DI Pointer to the second option or zero
SI Pointer to the third option or zero
RETURN: AL 0, 1 or 2 depending on option
COMMENTS:

FUNCTION NUMBER: 0A

TITLE: GenNotifyError
PURPOSE: Notify by error number
ENTRY: AL The error number to be notified
BX Pointer to the first message
DX Pointer to the first option or zero
DI Pointer to the second option or zero
SI Pointer to the third option or zero
RETURN: AL 0, 1 or 2 depending on option
COMMENTS:

FUNCTION NUMBER: 0B

TITLE: GenNotifyHook
PURPOSE: Hook the notify interface
ENTRY: BX The message number
RETURN: Carry clear on success

COMMENTS:

FUNCTION NUMBER: 0C

TITLE: GenNotifyUnHook
PURPOSE: Unhook the notify interface
ENTRY: None
RETURN:

COMMENTS:

FUNCTION NUMBER: 0D

TITLE: GenGetRAMSizeInParas
PURPOSE: Get the system RAM size
ENTRY: None
RETURN: Size of system RAM in paras (16 bytes)

COMMENTS:

FUNCTION NUMBER: 0E

TITLE: GenGetCommandLine
PURPOSE: Get the command line
ENTRY: None
RETURN: AX Pointer to the command line or 0

COMMENTS: The command line is an entry in the heap.

FUNCTION NUMBER: 0F

TITLE: GenGetSoundFlags
PURPOSE: Get the sound flags
ENTRY: None
RETURN: AL the sound flags

COMMENTS: Sound flag bits are as follows:

D0 set = Enable keyboard clicks
D1 set = Enable the piezo sound system
D2 set = Enable the SND: device driver
D3 set = Make piezo sound louder
D4 set = Disable all system sound

FUNCTION NUMBER: 10

TITLE: GenSetSoundFlags
PURPOSE: Set the sound flags
ENTRY: BX The sound flags
RETURN: None

COMMENTS: Sound flag bits in BX are as follows:

D0 set = Enable keyboard clicks
D1 set = Enable the piezo sound system
D2 set = Enable the SND device driver
D3 set = Make piezo sound louder
D4 set = Disable all system sound

For example, BX = 12 will produce a loud piezo after calling function 11

FUNCTION NUMBER: 11

TITLE: GenSound
PURPOSE: Make a sound with the piezo
ENTRY: BX The duration of the sound in ticks
CX The pitch of the sound
RETURN: None

COMMENTS: Call function 10 before using this function. The pitch is (512/CX) kHz. Try BX = 000F and CX = 01FF.

FUNCTION NUMBER: 12

TITLE: GenMarkActive
PURPOSE: Mark a process as active
ENTRY: None
RETURN: None

FUNCTION NUMBER: 13

TITLE: GenMarkNonActive
PURPOSE: None
ENTRY: None
RETURN:

COMMENTS:

FUNCTION NUMBER: 14

TITLE: GenGetText
PURPOSE: Get the operating system text
ENTRY: AL The number of the text message to be retrieved
ES:BX Pointer to the buffer to receive the text
RETURN: Carry clear on success

COMMENTS:

FUNCTION NUMBER: 15

TITLE: GenGetNotifyState
PURPOSE: Get notify state
ENTRY: None
RETURN: AL The notify state

COMMENTS:

FUNCTION NUMBER: 16

TITLE: GenSetNotifyState
PURPOSE: Set the notify state

ENTRY: AL The notify state
RETURN: None

COMMENTS:

FUNCTION NUMBER: 17

TITLE: GenGetAutoSwitchOffValue
PURPOSE: Get the auto switch off time
ENTRY: None
RETURN: AX The auto switch off time in seconds

COMMENTS: Default time is 300 seconds

FUNCTION NUMBER: 18

TITLE: GenSetAutoSwitchOffValue
PURPOSE: Set the auto switch off time
ENTRY: BX The auto switch off time in seconds
RETURN: None

COMMENTS: -1 = disable

FUNCTION NUMBER: 19

TITLE: GenSetRevector
PURPOSE: Capture an interrupt
ENTRY: AL The vector number
CX:BX The segment and offset on the interrupt routine
RETURN: None

COMMENTS:

FUNCTION NUMBER: 1A

TITLE: GenResetRevector
PURPOSE: Release an interrupt
ENTRY: AL the vector number
RETURN: None

FUNCTION NUMBER: 1B

TITLE: GenGetLanguageCode
PURPOSE: Get the Language Code
ENTRY: None
RETURN: AX the language code

COMMENTS: 1 = Eng, 2 = Fr, 3 = Ger, 4 = Sp, 5 = It, 6 = Sw, 7 = Danish, 8 = Nor, 9 = Fin, 10 = USA, 11 = Swiss Fr, 12 = Swis Ger, 13 = Port, 14 = Turk, 15 = Ice, 16 = Russian, 17 = Hung, 18 = Dutch, 19 = Belgian Fl, 20 = Aust, 21 = New Z, 22 = Austrian, 23 Belgian Fr.

FUNCTION NUMBER: 1C

TITLE: GenGetSuffixes
PURPOSE: Get suffix text
ENTRY: ES:BX Pointer to buffer to receive the suffix text
RETURN: None

COMMENTS:

FUNCTION NUMBER: 1D

TITLE: GenGetAmPmText
PURPOSE: Get the AM and PM text
ENTRY: AL Zero = get AM text
Non zero = get PM text
RETURN: ES:BX Pointer to buffer to receive the AM and PM text

COMMENTS: Buffer must be ≥ 6 bytes

FUNCTION NUMBER: 20

TITLE: GenSetBatteryType
PURPOSE: Set the battery type
ENTRY: AL The battery type
RETURN: None

COMMENTS:

FUNCTION NUMBER: 29

TITLE: GenCRC
PURPOSE: Generate a CRC
ENTRY: CX The number of bytes in the buffer
DX The current CRC
DS:SI Pointer to the buffer to be CRC checked
RETURN: AX The updated CRC checksum

COMMENTS:

FUNCTION NUMBER: 21

TITLE: GenEnvBufferGet
PURPOSE: Get environment variable
ENTRY: ES:DI Pointer to the environment variable name
DL Length of the environment variable name

ES:SI Pointer to the buffer to receive the variable's name

RETURN: AX The length of the data returned in ES:SI

COMMENTS:

FUNCTION NUMBER: 22

TITLE: GenEnvBufferSet
PURPOSE: Set environment variable
ENTRY: ES:DI Pointer to the environment variable name
 DL Length of the environment variable name
 ES:SI Pointer to the buffer containing the variable's
 data
 CL The length of the data as ES:SI
RETURN: Carry clear on success

FUNCTION NUMBER: 23

TITLE: GenEnvBufferDelete
PURPOSE: Delete environment variable
ENTRY: ES:DI Pointer to the environment variable name
 DL Length of the environment variable name
RETURN: Carry clear on success

COMMENTS:

FUNCTION NUMBER: 24

TITLE: GenEnvBufferFind
PURPOSE: Find environment variable
ENTRY: BX The find handle

Appendix 7 - Interrupts and Function Calls

ES:DI Pointer to the environment variable name

DL Length of the environment variable name

ES:SI Pointer to the buffer to receive the variable's data

RETURN: AX The next find handle

COMMENTS:

FUNCTION NUMBER: 25

TITLE: GenEnvStringGet

PURPOSE: Get string environment variable

ENTRY: ES:DI Pointer to the environment name string

ES:SI Pointer to the buffer to receive the variable's

name

RETURN: Carry clear on success

COMMENTS:

FUNCTION NUMBER: 26

TITLE: GenEnvStringSet

PURPOSE: Set string environment variable

ENTRY: ES:DI Pointer to the environment variable name

ES:SI Pointer to the string containing the variable's data

RETURN: Carry clear

COMMENTS:

FUNCTION NUMBER: 27

TITLE: GenEnvStringDelete
PURPOSE: Delete the string environment variable
ENTRY: ES:DI Pointer to the environment variable name
RETURN: Carry clear

COMMENTS:

FUNCTION NUMBER: 28

TITLE: GenEnvStringFind
PURPOSE: Find string environment variable
ENTRY: BX The find handle
ES:DI Pointer to the environment variable name
ES:SI Pointer to the buffer to receive the variable's
name
ES:CX Pointer to the buffer to receive the
variable's data
RETURN: AX The next find handle

COMMENTS:

FUNCTION NUMBER: 2B

TITLE: GenAlarmHook
PURPOSE: Hook the alarm interface
ENTRY: BX The message number

Appendix 7 - Interrupts and Function Calls

RETURN: Carry clear
COMMENTS: ALM: is the alarm device

FUNCTION NUMBER: 2C

TITLE: GenAlarmUnHook
PURPOSE: Unhook the alarm interface
ENTRY: None
RETURN: None

COMMENTS:

FUNCTION NUMBER: 2D

TITLE: GenAlarmID
PURPOSE: Get the PID of the alarm server
ENTRY: None
RETURN: AX The alarm server ID

COMMENTS:

Trig Functions

FUNCTION NUMBER: 00

PURPOSE: Sine of a float
ENTRY: DI Pointer to destination floating point number
SI Pointer to floating point function argument

FUNCTION NUMBER: 01

PURPOSE: Cosine of a float
ENTRY: DI Pointer to destination floating point number
SI Pointer to floating point function argument

FUNCTION NUMBER: 02

PURPOSE: Tangent of a float
ENTRY: DI Pointer to destination floating point number
SI Pointer to floating point function argument

FUNCTION NUMBER: 03

PURPOSE: Arcsine of a float
ENTRY: DI Pointer to destination floating point number
SI Pointer to floating point function argument

FUNCTION NUMBER: 05

PURPOSE: Arctangent of a float
ENTRY: DI Pointer to destination floating point number
 SI Pointer to floating point function argument

FUNCTION NUMBER: 06

PURPOSE: Exponential of a float
ENTRY: DI Pointer to destination floating point number
 SI Pointer to floating point function argument

FUNCTION NUMBER: 07

PURPOSE: Natural logarithm of a float
ENTRY: DI Pointer to destination floating point number
 SI Pointer to floating point function argument

FUNCTION NUMBER: 08

PURPOSE: Logarithm of a float
ENTRY: DI Pointer to destination floating point number
 SI Pointer to floating point function argument

FUNCTION NUMBER: 09

PURPOSE: Square root of a float
ENTRY: DI Pointer to destination floating point number
 SI Pointer to floating point function argument

FUNCTION NUMBER: 0A

PURPOSE: Power of 2 floats
ENTRY: DI Pointer to destination floating point operand
SI Pointer to floating point base
DX Pointer to floating point power

FUNCTION NUMBER: 0B

PURPOSE: Float random number
ENTRY: DI Pointer to destination floating point number
SI Pointer to unsigned long integer seed

FUNCTION NUMBER: 0C

PURPOSE: Modulo of a float
ENTRY: DI Pointer to destination floating point operand
SI Pointer to floating point function argument
DX Pointer to floating point modulo value

FUNCTION NUMBER: 0D

PURPOSE: Zero fractional part of a float
ENTRY: DI Pointer to destination floating point operand
SI Pointer to floating point function argument

T

M

A

C

F

The assembler which runs on the Psion Series 3A is available on PC disk ready for downloading. Instructions are on a ReadMe file.

Please return this form if you would like to receive your **FREE** copy.

Please allow 28 days for delivery. Overseas orders please allow £2.00 carriage charges by Visa or Mastercharge credit cards (please give details below).

NAME:

ADDRESS:

CREDIT CARD NO:

EXPIRY:

For those 3A owners who do not have a PC link for downloading the assembler, this is available on 128K Rom cartridge at a cost of £39.95 inc VAT and carriage.

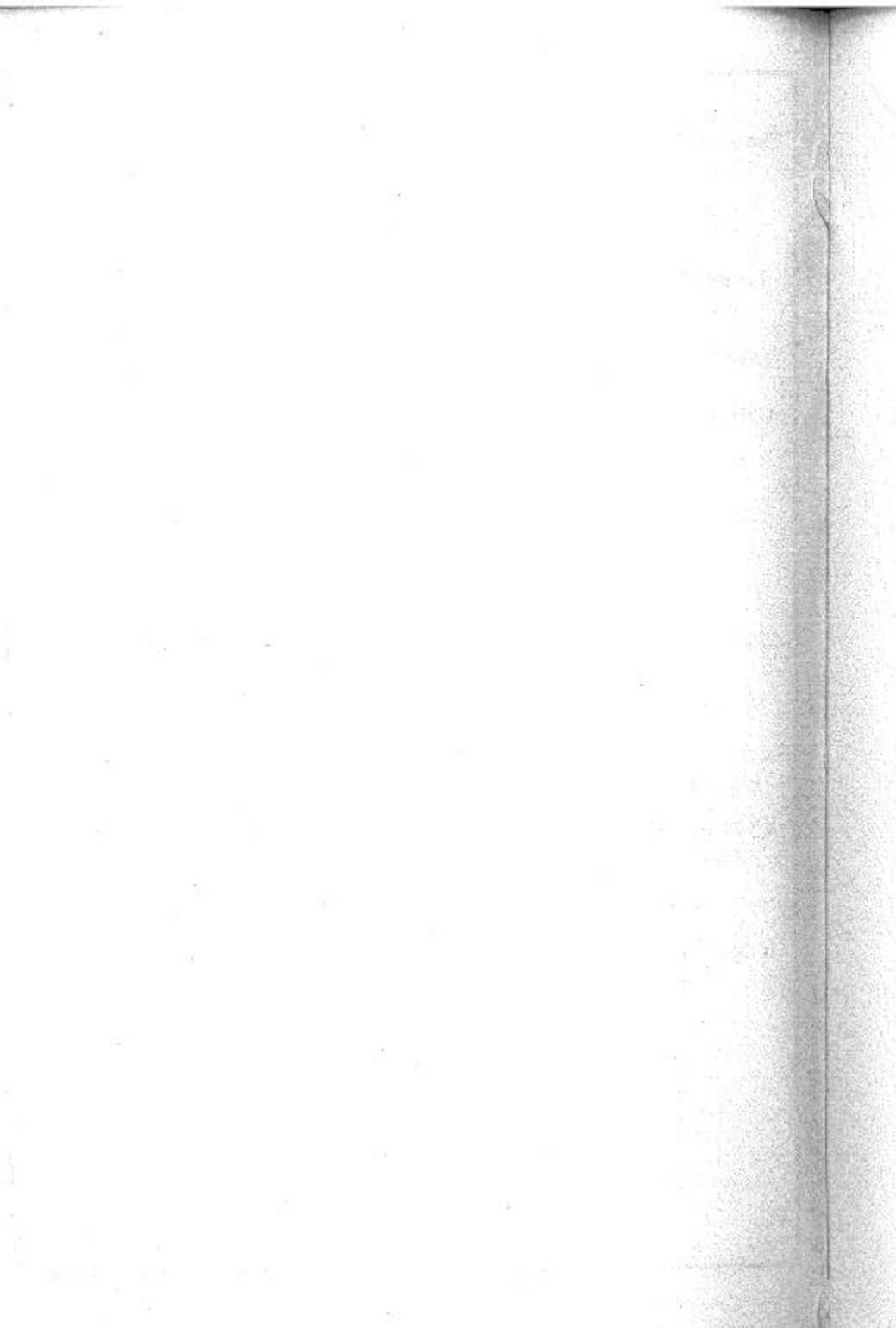
If you would like to order this please tick here and give details of your credit card above, or alternatively send a cheque made payable to Kuma Books Ltd.

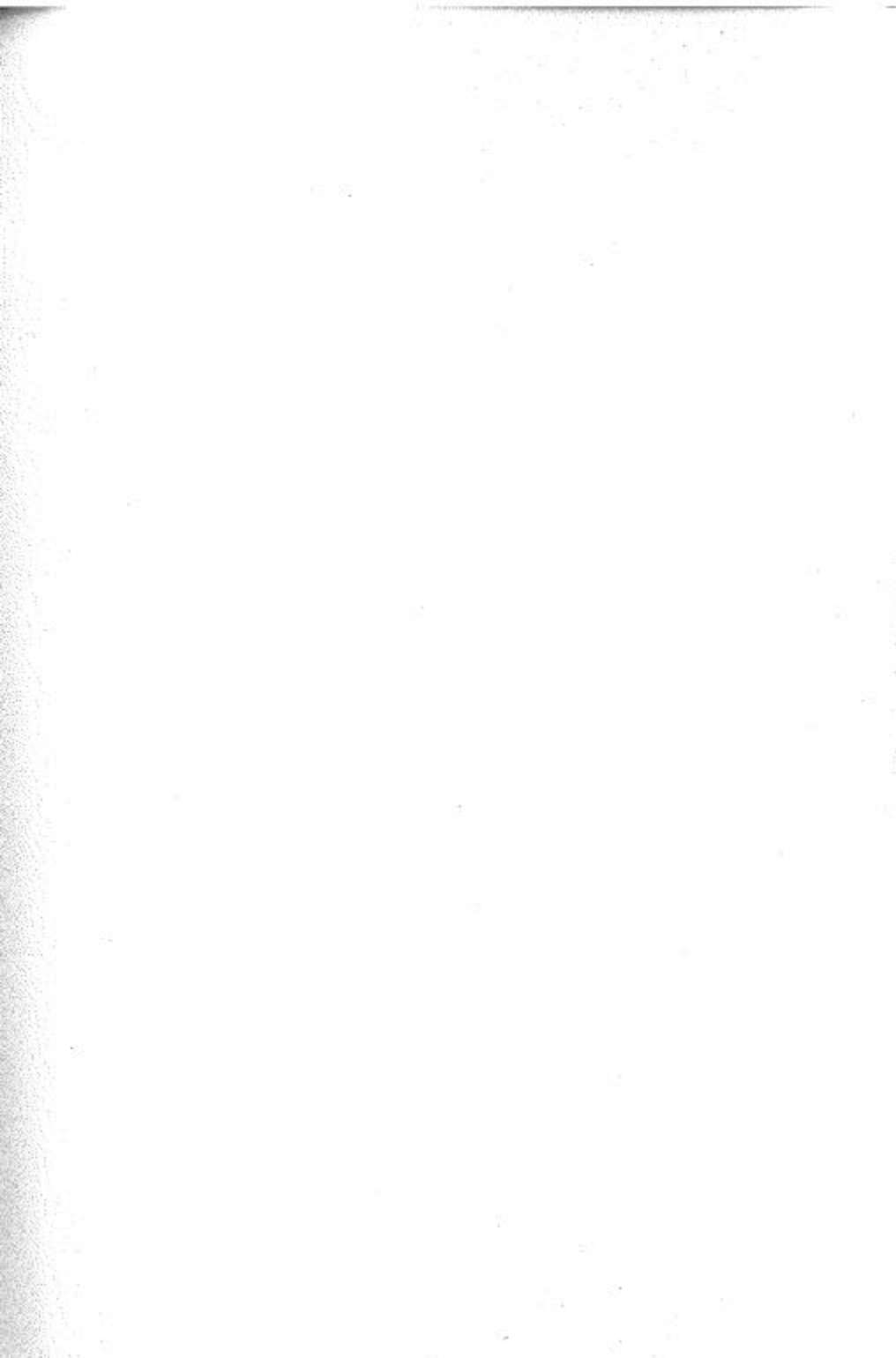
☐

If there are any areas of interest concerning your Psion 3A that you would like to see covered in a book, please contact us at the address below.

Please return this form to:

**KUMA BOOKS LTD
12 HORSESHOE PARK
PANGBOURNE
BERKSHIRE
RG8 7JW**





MACHINE CODE PROGRAMMING ON THE PSION SERIES 3A & 3

by John Farrant

Starting with a general introduction to the Psion 3 family, this book attempts to solve the problem of writing machine code to run on this hand-held computer. The hardware of the Psion 3A/3 is fully described, together with an insight to how the operating system works.

Providing a functional description of the 8086 processor, this book moves on to constructing programs, writing in 8086 machine code and assembly language using a fully functional two-pass assembler with its own text editor and debugger. Along the way, you will gain a better understanding of the Psion 3's hardware design and operation.

You will discover how to write assembly language programs using the most suitable commands, and then how to debug them if they don't work.

All the programs listed in this book are on the supplied disc.

The contents of this book apply to the models 3A and 3.

The author of this book was a Senior Lecturer in microcomputers and electronics at Newham College of Further Education until he decided to become a full-time writer and computer consultant.

Published by
Kuma

Kuma Books Ltd.
Pangbourne, Berkshire, England
Telephone: 0734 - 844335
Fax: 0734 - 844339

£16.95 net

ISBN C-7457-0299-6



9 780745 702995



Machine Code Programming on the PSION SERIES 3A & 3

John Farrant

KUMA