

BUSINESS COMPUTER SYSTEMS PLC.

MOLECULAR 18
SOFTWARE MANUAL
MK V

Released Nov. 1984

Re Order no. 660799

Issue 2.1 - 1:11.84

no. 1000
to 1000
of 1000
of 1000
of 1000
of 1000

This software and all its associated documentation (including all rights herein) is the property of BUSINESS COMPUTER SYSTEMS PLC and may only be reproduced and used in accordance with the terms under which it was supplied or otherwise with the prior written permission of BUSINESS COMPUTER SYSTEMS PLC.

(C) BUSINESS COMPUTER SYSTEMS PLC - 1984

THE UNIVERSITY OF CHICAGO
DEPARTMENT OF CHEMISTRY
530 SOUTH EAST ASIAN AVENUE
CHICAGO, ILLINOIS 60607
TEL: 773-936-3700

RECEIVED
JAN 15 1984
CHEMISTRY DEPARTMENT

CONTENTS

| | |
|------------|--------------------------------------|
| SECTION 1 | GENERAL |
| SECTION 2 | MACHINE CODE PROGRAMMING |
| SECTION 3 | GENERAL LIBRARY SUBROUTINES |
| SECTION 4 | DATA RETRIEVAL SYSTEM |
| SECTION 5 | SPOOLING & POSTING |
| SECTION 6 | I/O STATION & PRINTER PROGRAMMING |
| SECTION 7 | OPERATING SYSTEM UTILITIES |
| SECTION 8 | OPERATING SYSTEM COMMANDS |
| SECTION 9 | MISCELLANEOUS PROGRAMMER INFORMATION |
| SECTION 10 | APPENDICES |

SECTION 1

GENERAL

CONTENTS

PAGE

| | |
|-------------------------------|-------|
| SOFTWARE CONCEPTS | 1. 1 |
| The Operating System | 1. 1 |
| Program and Tasks | 1. 1 |
| Spooling and Print Queues | 1. 2 |
| The Configuration Table | 1. 3 |
| The System Catalogue | 1. 4 |
| The System Tables File | 1. 5 |
| HARDWARE ORGANISATION | 1. 6 |
| Overview | 1. 6 |
| Primary Storage | 1. 6 |
| Central Processing Unit | 1. 9 |
| Peripheral interface | 1. 9 |
| MEMORY ORGANISATION | 1. 10 |
| Addressing Techniques | 1. 10 |
| Operating System Requirements | 1. 12 |
| Task Requirements | 1. 12 |
| System Workspace Requirements | 1. 13 |

11 11

11 11

11 11

11 11

11 11

11 11

11 11
11 11
11 11
11 11

11 11 11 11
11 11 11 11
11 11 11 11
11 11 11 11

SOFTWARE CONCEPTS

The Operating System

All software functions are controlled by a memory-based Control Program or Operating System (OS). The term 'Memory-based' denotes that after loading (or Bootstrapping) the machine, the OS is memory resident and not utilising disk backing storage. This design gives efficiency in terms of programming simplicity and operating response times.

The basic OS occupies 12K words of memory. A 1K block of memory is 1024 words addressed 0 to 1023 and is often referred to by the term 'page'.

Some of the features available within the OS are as follows:

- a) input and output procedures may be carried out independently and concurrently.
- b) each conversational terminal or I/O station may act as an independent control device for output functions whilst still performing its normal task as an integral part of the system.
- c) a comprehensive library of application - orientated subroutines which significantly reduce program development time.
- d) system control functions.
- e) a comprehensive set of utilities which allow on-line loading, debugging, amendment and listing (in annotated Assembler language) of application program as well as facilitating system initiation and enhancement.
- f) easy recovery procedures from security disks when required.
- g) facilities to regenerate the Operating System to meet varying configuration requirements and availabilities.
- h) mains failure protection facilities.

Currently available versions of the Operating System cater for various peripherals. See COMA Utility for details.

Programs and Tasks

The series of instructions together with static data constitute what is referred to as a 'Program' sometimes called a 'Program module' or an 'overlay module'. The program is held on disk within a library of programs (referred to as the Overlay library) and is called into memory when required i.e. it overlays an area of memory. A program may be called by entering its name at an I/O Station e.g. VDU. Since there may be several I/O stations, there may be several programs running at once or the same program may be running more than once - in such circumstances a multi-programming environment exists.

Each I/O station is referred to as a 'Task' and is allocated its own task number, from one upwards. The terminology is simply derived from the fact that an I/O station running any program is performing a task. Other tasks may also be running in the background - one for each active printer or tasks not requiring any I/O functions.

Each task is allocated a memory partition for its exclusive use. This partition is fixed both in size and position; all task partitions are 2K words or 2 pages long and must begin on a page boundary. Obviously certain programs will be small so that the 2K task partition will not be fully utilised. Such 'spare memory' however is emphatically not available for use by any other task. Where the program is sufficiently large enough to require more than 2K words then the technique of overlaying must be employed.

Spooling and Print Queues

As mentioned previously, each I/O station and each printer is defined as a separate task. It follows therefore that the I/O station and printer programs are logically separate units.

If, for example, an application required that some information is input, via an I/O station, processed in some way and then output to a printer, this would require two programs to be written. The 'input' program would, for example, validate the information whilst the 'print' program would, process the information and output the results. The I/O Station task does not output directly to a printer; the information to be printed has to be transferred from the I/O station task ie. the 'input' program to the printer task ie. the 'print' program. The mechanism used to achieve this transfer is referred to as 'spooling'.

Within any task partition a particular area is reserved for holding data which is to be either transferred to or from a disk-held spool file. This area is referred to as the 'Spool Buffer'. The 'input' program will set up any data required to control printing within its spool buffer and subroutines available to the programmer will be used to transfer the contents of the spool buffer to the spool file. When data is transferred to the spool file it is said to have been 'posted'.

Once an I/O Station task has posted its printing requirements to the spool file it is then free to commence any other job without waiting for the slower print operation to complete. Any number of I/O Station tasks may be posting information to the spool file simultaneously and it is essential therefore that some mechanism exists whereby:

- a) Related print requirements may be linked together so that subsequent printing is not garbled.
- b) Printouts required to appear on pre-printed stationery may be initiated when the relevant stationery is available and has been properly loaded onto an available printer.
- c) The sequence of printing various reports may be controlled to suit individual user requirements.

The means by which this is accomplished is to link in a controlled manner all postings to the spool file. A number of 'starting points' are available within the spool file and as information is posted it may be linked to any of these starting points (controlled by the program or by the operator). Subsequent postings may be linked to previous postings already linked to one of the starting points. Within the spool file therefore will exist a number of 'chains' of print requirements which are waiting or 'queuing' to be printed. The chains of postings are therefore referred to as 'Print Queues' or 'Spool Queues'.

It is important to realise that the spool file is not partitioned; partitions would have to be of some defined size which may individually become full of postings thereby causing bottlenecks in processing. By organising the spool file into print queues the manner of its sub-division becomes totally flexible e.g. one print queue may occupy for example 95% of the spool file whilst another twenty print queues may occupy the remaining 5%.

The method of actually retrieving the postings from the print queues within the spool file is fully discussed in the section - SPOOLING AND POSTING.

The Configuration Table

The initial bootstrap procedure (described later in this section) will simply load a basic Operating System (OS) into memory. Contained within the Operating System is a routine known as the Initiator which is called as soon as the OS is loaded, and this reads a Configuration Table of 3 sectors from disk into memory at location 00/0400. The Configuration Table will contain the parameters defining the System Catalogue, System Tables, memory and peripheral requirements (including Disk backing storage) for the installation. By reference to this, the Initiator will mould the basic Operating System into one which is applicable to the requirements of the installation.

The moulding process is achieved simply by transferring information from the Configuration Table into appropriate areas of the Operating System.

At any installation it will be possible to perform a specific number of jobs or tasks simultaneously. There will be a memory requirement for any task which may run on the installation and the OS is geared to handling tasks which are held within a 2K 'partition'.

For the purpose of normal running the Operating System will require to accommodate service routines for each of the peripherals in the system, File Control Blocks containing all the parameters relating to a file, outer print buffers, disk labels, disk queue vectors for any disk accesses and various tables, lists and queues. Therefore, sufficient free memory should be available for this purpose.

Various applications may be available on any installation. The definitions of these 'Application Systems', comprising of a list of all Files available to that System, number of Print queues, system passwords, date etc., are held in the System Tables File.

To summarise, therefore, for any installation, the Operating System will require to know:

- a) Number of tasks which may run simultaneously
- b) Peripheral type allocated to each task.
- c) Address of a 2K memory partition to be reserved for each task.
- d) Disk Backing Storage information.
- e) A description of the location of both the Catalogue and the System Tables.
- f) Location of free memory areas available within the system.

The System Catalogue

The System Catalogue is used to map out the available backing storage, thus allowing easy maintenance of disk-based data, by means of defining the basic layout and usage of the 'Data-Sets'. These may be used to define Files (see later).

Associated with each Data Set Definition and included on the appropriate catalogue record will be a 12 character name.

The System Tables File

File accessing is always done via an octal file identifier; that identifier is then used to access a pointer to the details for that file, known as the File Control Block (FCB). All FCB pointers are in fact held as a 'File Table' within the System Table file, and each pointer is simply the Catalogue Record Number which holds the FCB details for that file identifier. The initiator will, upon bootstrapping, obtain the FCB details of all files from the Catalogue and establish them in free memory areas.

File identifiers are associated with the correct FCB details by use of a Programmer Utility known as MAPS i.e. Maintain Application System(s). Within this utility it is possible to set up/amend the file table so that any slot reserved for a file identifier may have inserted into it the catalogue record number containing the FCB details. This is done by simply entering the file identifier and the Data-Set name with which it is to be associated. A search of the System Catalogue will then take place until a catalogue record is found containing the entered name. The number of that record within the catalogue is then inserted into the slot reserved for that file identifier within the file table.

When the system is bootstrapped, one of the functions performed by the Initiator when 'moulding' the Operating System is the transfer of the file table of FCB pointers from the System table file to a location in free memory. At this point all the catalogue records specified within the file table are read into various free memory areas. The memory address of each Catalogue Record (or FCB) is then inserted into the file table replacing the initial catalogue record number. Reference to any file via a file identifier will then enable the address of the FCB details for that file to be accessed. With the FCB details located, any part of the file may be accessed.

As part of any task it will often be necessary to enable input of parameters etc. as a result of which some form of report or output document will require to be printed; this will require the use of one of the printers available to the installation and so any I/O Station must be given the ability to drive a particular printer when necessary.

Facilities are available, via Program MAPS, to associate I/O stations with certain printers and/or preclude their use by other I/O stations.

HARDWARE ORGANISATION

Overview

CENTRAL PROCESSING UNIT

A REGISTER
 B REGISTER
 MA MEMORY ADDRESS REG
 PC PROGRAM COUNTER
 C CARRY FLAG
 GT GREATER-THAN FLAG

INPUT/OUTPUT BUS

SLOW SPEED PERIPHERALS

VISUAL DISPLAY UNIT
 TELETYPE
 MODEM
 PRINTER

PRIMARY STORAGE

MEMORY 128K WORDS
 --- BUS --- (FERRITE CORE/MOS
 MAIN MEMORY)
 EACH WORD
 17 BITS DATA
 1 BIT PARITY

DATA CHANNEL

HIGH SPEED PERIPHERALS

DISC DRIVES

Primary Storage

Primary storage is a fast random access storage medium used to hold program instructions and immediate program data. It is supplied in 16K or 32K word stacks or units (1K = 1024 words). Each word consists of 18 bits where a bit is a binary digit which may have only 1 of 2 possible values ie 0 or 1. The physical medium may comprise of magnetic core or MOS chip memory, though the physical attributes of these media do not concern us here.

Only 17 of the 18 bits in each word are accessible to the program; the inaccessible 'parity' bit may be disregarded. The accessible bits are numbered 17 to 1 from left to right as follows:

17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

As stated above each bit has the binary value of either zero or one according as it is 'off' or 'on' respectively. It is possible to represent the setting of each bit within a word by means of a 17 digit binary number.

eg. 00 100 010 110 100 001

indicates that bits 15, 11, 9, 8, 6 and 1 only are 'on'. However, it is more convenient to group the bits into threes as above and to represent the setting of each group by means of an Octal Number.

| BINARY | OCTAL |
|--------|-------|
| 000 | 0 |
| 001 | 1 |
| 010 | 2 |
| 011 | 3 |
| 100 | 4 |
| 101 | 5 |
| 110 | 6 |
| 111 | 7 |

Thus, a word with bits 15, 11, 9, 8, 6 and 1 only 'on' is represented in octal by the six digit number 042641. All program instructions, addresses, parameters and constant factors have a binary representation but by effectively summarizing it into a 6 digit octal number the memorizing and handling of binary representations becomes much more convenient. As indicated, any word of memory may be interpreted as one of the following:

- a) instruction
- b) address
- c) parameter
- d) literal or text
- e) constant factor

and depending upon the interpretation different rules will apply for determining the format of the binary representation of that word. The rules for determining the binary representation of an instruction and an address are fully discussed in Section 2 - MACHINE CODE PROGRAMMING although for convenience some discussion on addresses and addressing will take place later in this Section under MEMORY ORGANISATION. The binary representation of a parameter will be determined by either available subroutine requirements or by the particular requirements of the program, and that of a literal by the requirements of ASCII.

If the word is interpreted as a constant factor then the sum of the individual bits set 'on' will make up the value of the constant factor.

It can be seen therefore that a word with bits 16-1 inclusive set 'on' can have a maximum positive value of 65535 and a maximum negative value of -65536 if bit 17 only is set. For this reason bit 17 is described as the sign bit in words interpreted as constant factors; if it is set then the number must have a negative value, and if it is not set then the number is positive.

| BIT | VALUE IF ON | CUMULATIVE VALUE |
|-----|-------------|------------------|
| 1 | 1 | 1 |
| 2 | 2 | 3 |
| 3 | 4 | 7 |
| 4 | 8 | 15 |
| 5 | 16 | 31 |
| 6 | 32 | 63 |
| 7 | 64 | 127 |
| 8 | 128 | 255 |
| 9 | 256 | 511 |
| 10 | 512 | 1023 |
| 11 | 1024 | 2047 |
| 12 | 2048 | 4095 |
| 13 | 4096 | 8191 |
| 14 | 8192 | 16383 |
| 15 | 16384 | 32767 |
| 16 | 32768 | 65535 |
| 17 | -65536 | -1 |

Again, it is more convenient to consider the binary representations of constant factors in terms of octal digits as follows.

| <u>CONSTANT</u> | <u>BINARY REPRESENTATION</u> | <u>OCTAL</u> |
|-----------------|------------------------------|--------------|
| 8 | 00 000 000 000 001 000 | 10 |
| 10000 | 00 010 011 100 010 000 | 023420 |
| 1023 | 00 000 001 111 111 111 | 1777 |
| -1 | 11 111 111 111 111 111 | 377777 |
| 50000 | 01 100 001 101 010 000 | 141520 |

It is useful at this stage to introduce the concepts of 'one's complement' and 'two's complement'; given that we have the octal representation of a number and wish to determine the representation should the sign be reversed then application of the above concepts becomes most useful eg. to find the octal representation of -5

| | <u>OCTAL</u> | <u>DEC</u> |
|------------------|------------------------|------------|
| | 00 000 000 000 000 101 | 000005 +5 |
| One's complement | 11 111 111 111 111 010 | |
| Plus 1 | | 1+ |
| Two's Complement | 11 111 111 111 111 011 | 377773 -5 |

Central Processing Unit

This unit handles all machine input and output, arithmetic or logic operations, memory updates, program flow and execution.

The processor retrieves instructions from Primary Storage as referenced by the Program Counter; during execution of a program the Program Counter automatically points to the next instruction to be executed. Data manipulation is performed via Memory and two other registers - the A and B registers or Accumulators. Data movements to and from memory and/or peripherals are achieved by instructions which reference these accumulators. Two hardware indicators are provided for use in conjunction with the A and B registers; namely the CARRY and the GREATER THAN flags. These allow for more sophisticated arithmetic and boolean operations eg, double word arithmetic is accomplished by use of the CARRY flag.

Peripheral Interface

Most peripherals are very much slower than the CPU itself and, therefore, mechanisms must exist to allow the processor to 'start' an input/output (I/O) operation and then return to some other duty while waiting for the I/O device to complete. This gives the impression that the CPU is processing more than one task simultaneously.

The two main facilities to achieve this are Program Interrupts and Direct Memory Access (DMA) or Data Channelling. The interrupt facility allows peripheral devices to interrupt the normal flow of the CPU when a certain function is complete. High speed devices, such as disk drives, 'steal' machine cycles from the processor to directly input to or extract from memory. This action proceeds automatically, the only effect on the processor being to 'slow' its execution. These features are discussed more fully at the end of Section 2 - MACHINE CODE PROGRAMMING.

MEMORY ORGANISATION

As the Molecular 18 uses a 16 bit Memory Address register, it follows that a maximum of 64K words of memory are directly addressable by the hardware at any one time. This is ample for most situations but for some of the larger and more sophisticated applications there is a requirement for more memory than this.

To overcome this limitation a technique known as 'bank switching' has been implemented. The memory is thought of as 32K 'banks', the first bank of which contains the OS and any tables, queues, buffers etc. This is known as bank zero, addressed as pages 0 to 37 octal. Bank one is used to hold task partitions and is addressed as pages 40 to 77 octal. This constitutes the standard 64K machine. Extra 32K banks may now be added, numbered 2 to 8, each of which can contain task partitions but are also addressed as pages 40 to 77.

The OS changes or 'switches' memory bank depending on which task partition requires service next. This 'switching' is invisible to the application programs running at the time and therefore is invisible to the application programmer. The difference is apparent only when configuring the task partitions as a bank number is required as well as a partition base address.

Addressing Techniques

The term 'random access' means that any location can be accessed at any time and in any order. It follows therefore that for memory to be random access, every distinct unit or word must be uniquely addressable. Reference has already been made to Pages (or 1K blocks) of memory and within any page each word is addressed from 0 to 1023 decimal or 0 to 1777 octal.

Each page of memory if further sub-divided in what are referred to as 'columns' of 64 words each; it then follows that there are 16 columns per page of memory. For programming purposes each word within a column is referred to as a 'step'. To summarise therefore:

| | | |
|----------|--------------|----------------------|
| 1 page | = 16 columns | addressed 0-17 octal |
| 1 column | = 64 steps | addressed 0-77 octal |

The column is a most important concept. It is equivalent to a complete M18 Program Coding Sheet; the contents of the coding sheet are therefore a 'window' into the memory itself.

Operating System Requirements

Assuming that on any configuration at least one I/O station and one printer would be required and each would require a task partition of 2K words, the minimum practical configuration would require 16K words.

The OS will probably require extra areas of memory, depending upon the size of the installation and the complexity of the applications; this may be for file definitions, actual device service routines, disk file buffer areas etc. The information as to where these areas are and also, for example, the number, location and type of peripherals to be used is transmitted to the OS by means of the Configuration Table discussed earlier in this section under SOFTWARE CONCEPTS.

Task Requirements

Each task has its own memory partition. All partitions are of the same size and format. Because the partition maps are pre-defined and known throughout the system, there is a considerable saving in the number of parameters required by subroutines. Experience has shown a 2K partition size to be the optimum choice for Commercial Applications, and all offsets quoted in this Manual will refer only to a 2K partition with a base offset of zero ie first word of the partition is offset 0000- and the last word of the partition is offset 3777-

Each task partition will comprise the following:

- a) a PROGRAM AREA occupying 0000- to 3177-
- b) a MASTER BUFFER occupying 3200- to 3377-
- c) a TASK SPOOL BUFFER 3400- to 3577-
- d) an INPUT BUFFER or PRINT BUFFER depending upon whether this is an I/O STATION or a print program and occupying 3600- to 3677- for an I/O STATION task, or 3600- to 3703- for a PRINT task
- e) a CONTROL AREA occupying 3704- to 3777-

The PROGRAM AREA will contain program code, data and work areas. Program and Overlay Modules are loaded into this area by FETCH subroutines.

The MASTER BUFFER is used by the FETCH, and OVERWRITE subroutines and is large enough to hold 128 words (equivalent to 1 sector of disk storage) of information.

The TASK SPOOL BUFFER is used by the SPOOL, UNSPOOL, POST, SPOOL AND POST subroutines. The input task creates spool records here; the print task receives spool records here. This buffer may be used as workspace (or as an extension to the MASTER BUFFER) when spooling is not required.

The INPUT BUFFER is used by the GET and SPLIT subroutines and contains the most recent input from the I/O STATION (ASCII format terminated by a Nul byte).

The PRINT BUFFER is used by the PRINTER OUTPUT subroutine which always space fills the buffer. The buffer holds one line of ASCII print-out; a map of print positions against offsets is provided in the Appendices.

The CONTROL AREA is reserved for System use.

System Workspace Requirements

The amount of system workspace required for different sites varies considerably depending on the particular configuration. See Utility "COMA" for workspace requirements calculation.

SECTION 2

MACHINE CODE PROGRAMMING

| <u>CONTENTS</u> | <u>PAGE</u> |
|--------------------------------------|-------------|
| PROGRAM INSTRUCTION FORMATS | 2.1 |
| MEMORY REFERENCE INSTRUCTIONS | 2.3 |
| Jump - unconditional | 2.4 |
| Jump - Sub-routine | 2.4 |
| Increment and Skip if zero | 2.5 |
| Decrement and skip if zero | 2.6 |
| 'And' to A | 2.7 |
| 'Inclusive or' to A | 2.7 |
| 'Exclusive or' to A | 2.8 |
| Add to A | 2.8 |
| Add to B | 2.8 |
| Subtract from A | 2.9 |
| Subtract from B | 2.9 |
| Add to A with Carry | 2.9 |
| Add to B with Carry | 2.10 |
| Subtract from A with Carry | 2.10 |
| Subtract from B with Carry | 2.11 |
| Load into A | 2.11 |
| Load into B | 2.11 |
| Compare Store with A | 2.11 |
| Compare Store with B | 2.11 |
| Store A | 2.12 |
| Store B | 2.12 |
| ADDRESSING | 2.13 |
| REGISTER INSTRUCTIONS | 2.15 |
| Introduction | 2.15 |
| Mode 01 - Shift/Rotate Group | 2.16 |
| Mode 10 - Clear and Complement Group | 2.19 |
| Mode 11 - Alter/Skip Group | 2.20 |
| Mode 00 - Odd Instructions Group | 2.22 |
| INPUT/OUTPUT INSTRUCTIONS | 2.26 |
| Introduction | 2.26 |
| Instruction format | 2.26 |
| Input/Output | 2.27 |
| Program Interrupts | 2.28 |
| Data Channel | 2.29 |

PROGRAM INSTRUCTION FORMATS

There are three types of basic instructions which are grouped according to the bit format of the instruction word. These types are:-

- a) Memory Reference Instructions which deal mainly with the transfer of information to/from Accumulators A and B to/from the memory stores.
- b) Register Instructions which deal with shifts, clears, rotates, negative tests, zero tests, etc on the Accumulators.
- c) Input/Output Instructions which deal with the input and output of data to/from the peripherals, and include the Interrupt system.

A comparison of the three formats is given in Figure 2.1 below, and more detailed coding is included with the instruction descriptions following:-

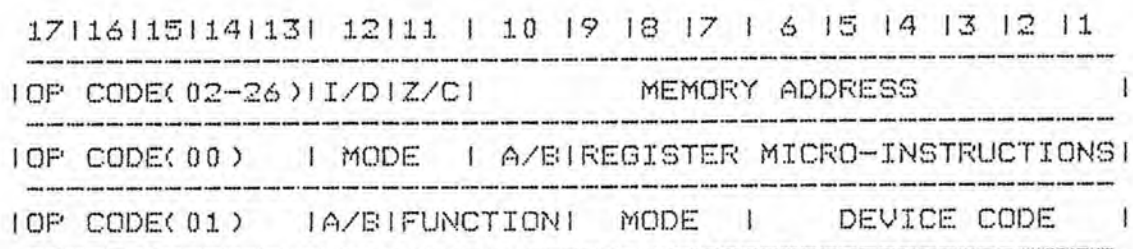


Figure 2.1

The first type comprises the Memory Reference instructions, using 10 bits (10 to 1) for a memory address, Bit 11 to specify Zero or Current page and Bit 12 for direct or indirect addressing. This leaves five bits (bits 17-13) to encode the 21 instruction commands in this group.

The other two types use these same five bits (17-13) to distinguish the Register and Input/Output Instructions, being all zeros for Register instructions and Bit 13 only being set for Input/Output.

The register type uses Bits 9 to 1 to combine in 'micro-instructions', with the resulting multiple instruction operating on the A or B accumulators as a single-word instruction. The Input/Output type uses Bits 12 to 7 for a variety of input/output instructions, and Bits 6 to 1 to make the instruction apply directly to one of the 63 possible input/output devices.

The following paragraphs describe in detail each of the instructions in the three groups. Functions of bits appearing in the form A/B, Z/C, I/D, L/R or T/F throughout these specifications are invariably obtained by coding a 1 or 0 respectively (1/0). Thus, for example, A is specified by a one-bit, and B by a zero bit. The following defines the abbreviations used:

| | |
|-----|-----------------------------|
| A/B | Accumulator A/Accumulator B |
| Z/C | Zero Page/Current Page |
| I/D | Indirect/Direct |
| L/R | Left/Right |
| T/F | True/False |

MEMORY REFERENCE INSTRUCTIONS

There are 21 Memory Reference Instructions which carry out some operation involving memory locations, such as transferring information in or out of a memory location or checking the memory location contents. The address referenced (ie. the absolute address) is determined by a combination of the task partition address, the ten memory address bits in the instruction word (10 to 1) plus two other bits (Bits 12 and 11). Bit 11 is reserved to specify either Page Zero or Current Page, and Bit 12 to specify direct or indirect addressing. The manner in which locations are specified for the Molecular 18 is discussed in detail under 'Addressing' later in this section.

All Memory Reference Instructions take a minimum of two machine cycles ('Fetch', to read the instruction word and 'Execute' to read the referenced memory location), except for JUMP, which takes a minimum of one machine cycle.

Note that since Accumulators A and B can be addressed, any Memory Reference instruction can apply to either of these registers, both directly and indirectly, as well as the Memory Stores. (Page zero must be specified for these operations, since the A and B register addresses, 0000 and 0001 are on Page Zero).

To summarise, a memory reference instruction uses a 10-bit address value to refer to a memory location, and then operates on the 17-bit binary number stored in the referenced memory location.

| <u>Mnemonic</u> | <u>Octal OP Code</u> |
|-----------------|----------------------|
| JUMP | 02 |
| JSER | 03 |
| INSZ | 04 |
| DESZ | 05 |
| ANDA | 06 |
| IORA | 07 |
| XORA | 10 |
| ADA | 11 |
| ADE | 12 |
| SFA | 13 |
| SFB | 14 |
| ADAC | 15 |
| ADEC | 16 |
| SFAC | 17 |
| SFBC | 20 |
| LDA | 21 |
| LDB | 22 |
| CMPA | 23 |
| CMPE | 24 |
| STA | 25 |
| STB | 26 |

Figure 2.2

JUMP = Unconditional Jump

The JUMP instruction loads the effective address to the instruction into the Program Counter (PC), thereby changing the program sequence since the PC specifies the next instruction to be performed. It then takes the next instruction from that location and continues operation from there. The JUMP instruction does not affect the contents of the Accumulators.

In the following example, execution of the instruction in column 02 step 70 (JUMP to 0270) causes the program to jump over the instructions in Steps 21 through 67 and immediately transfer control to the instruction in column 02 step 70.

| <u>Location</u> | <u>Content</u> |
|--------------------|--|
| Current page, 0220 | 020270 (this instruction transfers program control to location Column 02, step 70) |
| Current Page, 0270 | 250300 |

JSER = Jump to Subroutine

On this instruction the program will jump to the word of memory specified, which is assumed to have an initial value of zero. The value of the Program Counter (which is the address of the JSER instruction + 1), in other words the return address, is always stored in the first location of the subroutine, replacing the original contents (overwriting).

After the subroutine is executed, the pointer address identifies the next instruction to be executed. Thus, programmers have at their disposal a simple means of exiting from the normal flow of the program to perform an intermediate task and a means of return to the correct location upon completion of the task. (This return is accomplished using indirect addressing, which is discussed later in this section). This also facilitates nesting of routines.

One can also alter the program within the program by changing the PC + 1 which is placed at the step at the beginning of the subroutine.

INSZ - Increment, and skip if zero

This instruction adds 1 to the contents of the word of memory specified, all 17 bits, and then examines the result of the addition. If the result is zero, the instruction following the INSZ is skipped and the Carry Flag will be set. If the result is not zero the program will proceed normally to be instruction immediately following the INSZ (the next word of program in sequence).

The following points should be kept in mind when using the INSZ instruction:-

1. The contents of the A and B accumulators are not disturbed, unless the INSZ instruction is used to increment either of the hardware working registers 0000 and 0001.
2. The original data word in the reference memory location is replaced by the incremented value.
3. The INSZ performs the increment first and then checks for a zero result.
4. When using the INSZ for looping a specified number of times, the tally must be set to the negative (twos complement) of the desired number.
5. The Carry Flag is set whenever a transfer occurs between Bits 16 and 17 in any store as a result of the INSZ instruction.

DESZ = Decrement, and Skip if Zero

This instruction subtracts 1 from the contents of the word of memory specified, all 17 bits, and then examines the result of this subtraction. If the result is zero, the instruction following the DESZ is skipped. If the result is not zero, the program will proceed normally to the instruction immediately following DESZ (the next word of program in sequence). Should the specified store overflow as a result of this instruction, the Carry Flag will be set.

The following points should be kept in mind when using the DESZ instruction:-

1. The Contents of the A and B Accumulators are not disturbed, unless the DESZ instruction is used to decrement either of the two hardware working registers 0000 and 0001.
2. The original data in the referenced memory location is replaced by the decremented value.
3. The DESZ performs the decrement first and then checks for a zero result.
4. The Carry Flag is set whenever a transfer occurs between Bits 17 and 16 in any store, as a result of this transaction.

(eg. If a word contains Bit 17 only and is decremented, it will afterwards contain Bits 16 to 1 inclusive, and the Carry Flag will be set.)

ANDA = 'And' to A

The ANDA instruction causes a bit-by-bit Boolean AND operation between the Contents of Accumulator A and the contents of the word of memory specified by the ANDA instruction. The result is left in Accumulator A, replacing its original contents, but the word of memory specified is not altered.

The following points sum up the ANDA instruction:

1. A '1' is left in Accumulator A only when a '1' is present in the corresponding position of both Accumulator A and the specified word of memory (mask)
2. The Carry Flag is not affected, neither is the Greater Than flag as the operation is performed on a bit-for-bit basis.
3. The specified word of memory remains unaltered.

| Acc A. | Store | Result in Acc. A |
|--------|-------|---------------------|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

IORA = 'inclusive OR' to A

The IORA instruction causes a bit-by-bit Boolean OR (or Inclusive OR) operation between the contents of Accumulator A and the contents of the word of memory specified by the IORA instruction. The result is left in A, replacing its original contents, but the word of memory specified is not altered.

The following points sum up the IORA instruction

1. A '1' is inserted in Accumulator A if a 1 is present in the corresponding position of either Accumulator A or the operand.
2. The Carry Flag is not affected, neither is the Greater Than flag.
3. The specified word of memory remains unchanged.

| Acc A | Store | Result in A |
|-------|-------|-------------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

XORA = 'Exclusive OR' to A

The XORA instruction causes a bit-by-bit Boolean 'Exclusive OR' operation between the contents of Accumulator A and the contents of the word of memory specified by the XORA instruction. The result is left in Accumulator A, replacing its original contents, but the word of memory specified is not altered.

The following points sum up the XORA instruction

1. A '1' is inserted in Accumulator A only if the two corresponding bits of the Accumulator and the operand differ
2. The Carry Flag is not affected, neither is the Greater Than flag
3. The specified word of memory remains unaltered.

| Acc. A | Store | Result in A |
|--------|-------|-------------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

ADA = Add to A

The ADA instruction performs a binary addition between the specified data word and the contents of Accumulator A, all 17 bits, leaving the result of the addition in Accumulator A. The specified word of memory will remain unchanged, but the result of the addition could set the Carry Flag.

ADE = Add to E

The ADE instruction performs a binary addition between the specified word and the contents of Accumulator E, all 17 bits, leaving the result of the addition in Accumulator E. The specified word of memory will remain unchanged, but the result of the addition could set the Carry Flag.

SFA = Subtract from A

The SFA instruction performs a binary subtraction, subtracting the contents of the specified word of memory from the contents of Accumulator A, all 17 bits, leaving the difference in Accumulator A. The specified word of memory will remain unchanged, but the result of the subtraction could set the Carry Flag.

Arithmetically, binary numbers may be directly subtracted in a manner similar to decimal subtraction. The essential difference is that if a 'borrow' is required, it is equal to the base of the system of 2.

ie,

| | | | |
|-----|---|---|-----------|
| 110 | = | 6 | (decimal) |
| 101 | = | 5 | |
| 001 | = | 1 | |

To subtract 1 from 0 in the first column, a borrow of 1 was made from the second column, which effectively added 2 to the first column. After the borrow, $2 - 1 = 1$ in the first column; in the second column $0 - 0 = 0$; and in the third column $1 - 1 = 0$.

SFB = Subtract from B

The SFB instruction performs a binary subtraction, subtracting the contents of the specified word of memory from the contents of Accumulator B, all 17 bits, leaving the difference in Accumulator B. The specified word of memory remains unchanged, but the result of the subtraction could set the Carry Flag.

ADAC = Add with Carry (to Accumulator A)

Multiple register addition is possible using the ADAC instruction. This type of arithmetic is needed when a number that is too large to be contained in one word (ie. more than 65,535 in decimal) has to be added to another similar number, or when the result of an addition may be too large for a single register.

eg. Two numbers 65,535 and 65,537 are to be added together

| | | | | | | | | |
|----------|-------|------|------|------|-------|------|------|------|
| 65,535 = | 00000 | 0000 | 0000 | 0000 | 01111 | 1111 | 1111 | 1111 |
| 65,537 = | 00000 | 0000 | 0000 | 0001 | 00000 | 0000 | 0000 | 0001 |

Let us assume that the 65,535 is double stored in stores 0235 and 0236, the 65,537 is double stored in Stores 0251 and 0252, and that the answer is to be stored in 0276 and 0277. The first bit of program would read.

| | | | | | |
|----------|--------------------|-------|------|------|------|
| CLC | (Clear Carry) | | | | |
| LDA 0236 | (load A with 0236) | 01111 | 1111 | 1111 | 1111 |
| ADA 0252 | (Add to A 0252) | 00000 | 0000 | 0000 | 0001 |
| | (result in A) | 10000 | 0000 | 0000 | 0000 |

As Bit 17 is left on, the presence of this bit denotes a negative number, therefore it must be cleared before being stored in the answer store:

```
CLSA      (Clear sign of A)
STA       (Store A in 0277)   0000  0000  0000  0000
```

The first register has been added, and because a carry occurred between Bits 16 and 17 the Carry Flag will be set. Using ADAC for the addition of the second pair of stores, the 'with carry' will cause the Carry Flag to be added to the A register before the addition is started. The addition is then completed normally, and on completion the Carry Flag will no longer be set.

```
LDA 0235      00000  0000  0000  0000
                                     1 - Carry Flag
ADAC 0251     00000  0000  0000  0001
STA 0276      00000  0000  0000  0010
```

The total left in Stores 0276 and 0277 now equals:

```
00000  0000  0000  0010  00000  0000  0000  0000 = 131,072
```

To sum up, on an ADAC instruction, the contents of the Carry Flag (if any) will be added to the contents of Accumulator A, the Carry Flag will be cleared, and the contents of the word of memory specified will then be added to Accumulator A. The store specified will remain unchanged. This instruction is used for double or multiple store arithmetic.

NB: It should be pointed out that the Carry Flag could be set again by the addition (or subtraction).

ADEC = Add with Carry (To accumulator E)

Exactly as for ADAC, except that the contents of the Carry Flag (if any) will be added to the contents of Accumulator E, the Carry Flag will be cleared, and the contents of the word of memory specified will then be added to Accumulator E. The store specified will remain unchanged. This instruction is used for double or multiple store arithmetic.

SFAC = Subtract Store From A with Carry

The contents of the Carry Flag (if any) will be subtracted from the contents of Accumulator A, the Carry Flag will be cleared, and the contents of the word specified will then be subtracted from Accumulator A. The Store specified will remain unchanged. This instruction is used for double or multiple store arithmetic.

SFBC = Subtract Store from B with Carry

The contents of the Carry Flag (if any) will be subtracted from the contents of Accumulator B, the Carry Flag will be cleared, and the contents of the word specified will then be subtracted from Accumulator B. The store specified will remain unchanged. This instruction is used for double or multiple store arithmetic.

LDA = Load into A

LDA stores the contents of the referenced location in Accumulator A, over-writing the original contents of Accumulator A. The specified word of memory remains unaltered.

LDB = Load into B

LDB stores the contents of the referenced location in Accumulator B, over-writing the original contents of Accumulator B. The specified word of memory remains unaltered.

CMFA = Compare Store with A (skip if unequal)

This instruction compares the contents of the word of memory specified with the contents of Accumulator A, all 17 bits. If the two words are different the next instruction will be skipped. (ie. the PC is advanced by two instead of one). If both words are identical, the program will proceed normally to the next instruction in sequence. The contents of both the specified word of memory and Accumulator A remain unaltered.

Should the contents of Bits 16 to 1 of Accumulator A be greater than the contents of Bits 16 to 1 of the word of memory addressed, the Greater Than Flag will be set (but NOT the Carry Flag).

Should the contents of bits 16 to 1 of Accumulator A be less than or equal to the contents of bits 16 to 1 of the word of memory addressed, the Greater Than Flag will be cleared (but not the Carry Flag).

CMFB = Compare Store With B (skip if unequal)

This instruction compares the contents of the word of memory specified with the contents of Accumulator B, all 17 bits. If the two words are different the next instruction will be skipped. (ie. the PC is advanced by two instead of one). If both words are identical, the program will proceed normally to the next instruction in sequence. The contents of both the specified word of memory and Accumulator B remain unaltered.

Should the contents of Bits 16 to 1 of Accumulator B be greater than the contents of Bits 16 to 1 of the word of memory addressed, the Greater Than Flag will be set (but NOT the Carry flag).

Should the contents of Bits 16 to 1 of Accumulator B be less than or equal to the contents of Bits 16 to 1 of the word of memory addressed, the Greater Than flag will be cleared (but NOT the Carry Flag).

STA = Store A

The STA instruction stores the contents of Accumulator A in the word of memory specified, over-writing the original contents of the referenced location. Accumulator A remains unaltered.

It is not possible to STA indirectly into Accumulator B through another store.

STB = Store B

The STB instruction stores the contents of Accumulator B in the word of memory specified, over-writing the original contents of the referenced location. Accumulator B remains unaltered.

It is not possible to STB indirectly into Accumulator A through another store containing zero.

ADDRESSING

When the Memory Reference instructions were introduced, it was stated that the format was five bits for the operation code and the remaining twelve bits are allocated to specify the operand (the address referenced by the instruction). However, a full 16 bits are needed to uniquely address the 65536 locations that are contained in a 64K Molecular 18. To make the best use of the available twelve bits, the following formula has been adopted.

Bit 10 to 1 - Address Within Page

This is the address to which the rest of the instruction refers. The Memory Reference instructions can directly address any word in the Current or Zero Page (from 0 to 1023 in Binary).

e.g. Column 13 Step 24 is shown in the 10 address bits as follows:

| | | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|
| 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |

which, when read off in octal is 1324. Similarly, column 01 step 77 would be entered as Octal 0177 from Bit 10 downwards.

Bit 11 - Zero or Current Page Indicator

All Memory reference instructions include a bit (bit 11) reserved to specify either Page zero (the base page) or the Current Page (the page in which the instruction itself is located). These page references for addressing are specified by Bit 11 as follows:

1 = Zero Page (Z)
0 = Current Page (C)

To address locations in any other page, indirect addressing is used, via the current or zero page, as explained below.

Bit 12 - Indirect/Direct Address Indicator

All memory reference instructions include a bit (bit 12) reserved to specify direct or indirect addressing. Direct addressing combines the instruction code and the effective address into one word, permitting a Memory Reference instruction to be executed in two machine cycles (Fetch and Execute). Indirect addressing uses the address part of the Page which is taken as a new memory reference for the same instruction. This new address is a full 17 bits long, 16 bits of address plus another bit, Bit 17 which is used as a further Indirect/Direct bit. The 16 bit length of the address permits access to any location in memory, using the following method to work out the address

eg. for page 51, column 13, step 24 insert the column and step as described above. Then dividing the rest of the bits into groups of three from bit 11 upwards, insert the page in octal as below:

| | | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|
| 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |

Thus the address in octal is 123324.

Direct or indirect addressing is specified by Bit 12 (or Bit 17) as follows:

1 = Indirect
0 = Direct

Owing to the multi-programming aspects of the machine it is not practical to set up absolute addresses within a program. For complete flexibility it is recommended the offset address, as discussed in Section 1, be set up within any program. Such addresses, when resolved to absolute by the OS will have the format shown above.

REGISTER INSTRUCTIONSIntroduction

The register instructions in general manipulate bits in the A and B Accumulators. There is no reference to memory, thus these instructions are executed in only one machine cycle. They are combinable to form a one word multiple instruction that can operate in various ways on the contents of Accumulators A and B. These 'micro-instructions' are divided into four sub-groups, the Shift-Rotate Group, the Alter-Skip Group, the Clear and Complement Group and the Odd Instruction Group.

As shown previously the bit structure is made up as follows:

| | |
|-----------------|--|
| Bits 17 to 13 | specify the operation code (all zeros) |
| Bits 12 and 11 | specify the Mode |
| Bit 10 | specifies the Accumulator |
| Bit 9 through 1 | are the Micro-Instruction bits. |

Micro-instructions may be combined under the following general rules.

1. No instruction may be used which combine micro-instructions from different Modes.
2. Reference to both A and B registers cannot be mixed in the same micro-instruction.
3. If more than one of the micro-instruction bits of a particular Mode is set at any one time, the sequence of execution is left to right (Bit 9 to Bit 1).
4. Use zeros to exclude unwanted micro-instruction bits.
5. If two (or more) skip functions are combined, the skip will only occur if all conditions are fulfilled. One exception exists - In Mode 01 only, the skip will occur if either or both conditions are met.
6. Shift and Rotate or Increment and Decrement must not be mixed in the same micro-instruction - the effect of doing so is undefined.

There are four modes - the mode alters the meaning of the micro-instruction bits.

| | |
|---------|------------------------------|
| Mode 01 | (Shift/Rotate Group) |
| Mode 10 | (Clear and Complement Group) |
| Mode 11 | (Alter-skip Group) |
| Mode 00 | (Odd Instruction Group) |

Register instructions for modes 01, 10, 11

| MODE | | MICRO INSTRUCTIONS | | | | | | | | | | | | | |
|-------|---|--------------------|-------|-------|-------|--------|-------|------|-------|-------|-------|-------|------|---|--|
| Bits: | | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0 | 1 | A/B | Clear | Left | Shift | Rotate | With | Dec. | Inc. | Skip | Skip | | | | |
| 1 | 1 | 1/0 | Carry | Right | | | Carry | | | | | b16=0 | b1=0 | | |
| 1 | 0 | A/B | Clear | One's | Clear | Comp. | Skip | Swap | Clear | Comp. | Enter | | | | |
| 1 | 1 | 1/0 | Accum | Comp. | Carry | Carry | | | Sign | Sign | S/reg | | | | |
| 1 | 1 | A/B | True | Skip | Skip | Skip | Clear | Skip | Clear | Clear | One's | | | | |
| 1 | 1 | 1/0 | False | Neg. | Not 0 | Carry | Carry | If > | >Flag | Accum | Comp. | | | | |

Mode 01 - Shift/Rotate Group

There are 21 basic instructions in this group which can manipulate the contents of Accumulators A and B and the Carry Flag; these instructions can be combined with other instructions also in Mode 01.

Mode 01 Instructions

| <u>Mnemonics</u> | <u>Octal Code</u> |
|------------------|-------------------|
| CLC | 002400 |
| LSA | 003300 |
| LSB | 002300 |
| RSA | 003100 |
| RSB | 002100 |
| LRA | 003240 |
| RRA | 003040 |
| LRAC | 003260 |
| RRAC | 003060 |
| LRB | 002240 |
| RRB | 002040 |
| LRBC | 002260 |
| RRBC | 002060 |
| DECA | 003010 |
| DECB | 002010 |
| INCA | 003004 |
| INCB | 002004 |
| AMSB | 003002 |
| EMSB | 002002 |
| ALSB | 003001 |
| ELSB | 002001 |

If, for instance, it is desired to clear carry and left shift accumulator A, to perform this task the program could include the following instructions (given in both mnemonic and octal form).

```
CLC 002400
LSA 003300
```

Since the CLC and LSA instructions occupy separate bit positions, they may be used in the same instruction, thus combining the two operations into one instruction. This instruction would be annotated as CLC,LSA which is 003700 in octal. In this manner, many more Register instructions, separated by a comma, can be combined to make the execution of the program more efficient.

The operation of each individual instruction specified by Bits 10 to 1 is described below.

CLC = Clear Carry - causes the Carry Flag to be cleared

LSA or LSB = Left Shift A or E - Bits 16 to 1 of the specified accumulator will be shifted one place to the left. Bit 17 (ie. the sign bit) of the accumulator to be shifted remains stationary.

If there is a '1' bit in position 16 of the accumulator, the Carry Flag will be set after a left shift.

If there is a '0' bit in position 16 of the accumulator, the Carry Flag should it be set already, will not be overwritten after a Left Shift.

RSA or RSE = Right Shift A or E - Bits 16 to 1 of the specified accumulator will be shifted one place to the right. Bit 17 (ie. the sign bit) of the accumulator to be shifted remains stationary. If there is a '1' bit in Position 1 of the Accumulator, the Carry Flag will be set after a Right Shift. If there is a '0' bit in position 1 of the accumulator, the Carry Flag should it be set already, will not be overwritten after a Right Shift.

LRA or LRE = Left Rotate A or E - This instruction rotates Bits 16 to 1 of the specified accumulator one place to the left.

One left rotate will perform as for one left shift, except that the figure which was in Bit 16 will re-enter the Accumulator at Bit 1. The Carry Flag is not affected. In other words, it treats Bits 16 to 1 of the specified accumulator as a closed loop, and performs what is commonly called a circular shift, meaning that any bit rotated off the left end will re-appear at the right end.

| | | | | | |
|-------|--------------------|------|------|------|------|
| e. g. | One left rotate of | 0110 | 1100 | 0110 | 0101 |
| | gives | 1101 | 1000 | 1100 | 1010 |

RRA or RRE = Right Rotate A or E - This instruction rotates Bits 16 to 1 of the specified accumulator one place to the right.

One right rotate will perform as for one right shift, except that the figure which was in Bit 1 will re-enter the Accumulator at Bit 16. The Carry Flag is not affected. In other words, it treats Bits 16 to 1 of the specified accumulator as a closed loop, and performs what is commonly called a circular shift, meaning that any bit rotated off the right end will re-appear at the left end.

e. g. one right rotate of 0110 1100 0110 0101
gives 1011 0110 0011 0010

LRAC or LRBC = Left Rotate A or B with Carry - This instruction causes any previous Carry Flag to be introduced into the gap left by the rotate, and the bit rotated off the left end will replace the Carry Flag. In other words it treats Bits 16 to 1 PLUS THE CARRY FLAG as a closed loop, and performs a circular shift as previously described.

eg. 0110 1100 0110 0100 Carry flag is 1
after one left rotate with Carry would read:
11001 1000 1100 1001 Carry Flag is 0

Rotate with Carry is used for multiple store shifting.

RRAC or RREC = Right Rotate A or B with Carry - This instruction causes any previous Carry Flag to be introduced into the gap left by the rotate, and the bit rotated off the right end will replace the Carry Flag. In other words it treats Bits 16 to 1 PLUS THE CARRY FLAG as a closed loop, and performs a circular shift as previously described.

eg. 0110 1100 0110 0100 Carry Flag is 1
after one right rotate with Carry would read:
1011 0110 0011 0010 Carry Flag is 0.

Rotate with Carry is used for multiple store shifting.

DECA or DECB = Decrement A or B - This instruction will decrement the contents of the specified accumulator by 1.

INCA or INCB = Increment A or B - This instruction will increment the contents of the specified accumulator by 1.

AMSB or EMSB = Skip if bit 16 of A or B equals zero - This instruction causes the program to skip the next step if Bit 16 of the specified accumulator is zero.

ALSB or ELSB = Skip if bit 01 of A or B equals zero - This instruction causes the program to skip the next step if Bit 01 of the specified accumulator is zero, or, in other words, if there is an even number in the accumulator.

Mode 10 - Clear and Complement Group

There are 15 basic instructions in this group which can clear and complement the contents of Accumulators A and B, and the Carry Flag; as before, these instructions can be combined with others also in Mode 10.

| <u>Mnemonics</u> | <u>Octal Code</u> |
|------------------|-------------------|
| CLA | 005400 |
| CLE | 004400 |
| CPLA | 005200 |
| CPLB | 004200 |
| CLC | 004100 |
| CMFC | 004040 |
| SKIP | 004020 |
| SWFA | 005010 |
| SWFB | 004010 |
| CLSA | 005004 |
| CLSB | 004004 |
| CFSA | 005002 |
| CFSB | 004002 |
| ESRA | 005001 |
| ESRB | 004001 |

The operation of each individual instruction specified by Bits 10 to 1 is described below:

CLA or CLE = Clear A or B - set the specified accumulator (all 17 bits) to zeros.

CPLA or CPLB = Complement A or B - causes the specified accumulator (all 17 bits) to be set to the one's complement of its original value; that is, all ones become zeros, and all zeros become ones.

e.g before one's complement 01000 1100 1110 1111
 after one's complement 10111 0011 0001 0000

CLC = Clear Carry causes the Carry Flag to be cleared.

CMFC = Complement Carry causes the state of the Carry Flag to be complemented (ie reversed).

SKIP = Unconditional Skip - causes the program to skip the next step, unconditionally.

SWFA or SWFB = Swap A or B - causes the contents of the top half (bits 16 to 9) of the specified accumulator to be swapped with the contents of the bottom half (Bits 8 to 1) of the said accumulator. The sign bit is not affected.

CLSA or CLSB = Clear Sign of A or B - causes the sign bit (bit 17) of the specified accumulator to be cleared (set to zero).

CPSA or CPEB = Complement Sign of A or E - causes the state of the sign bit (Bit 17) of the specified accumulator to be complemented (ie reversed).

ESRA or ESRE = Enter Switch Register into A or E - causes whatever is set on the Data Switches of the Control Panel to be loaded into the specified accumulator.

If it is desired to set the sign of an accumulator or to set Carry, obviously the Mode 10 instructions may be used.

CLSA, CPSA (005006 Octal) is equivalent to 'Set the Sign of A'.

CLC, CMFC (004140 Octal) will cause the Carry Flag to be set.

Mode 11 = Alter/Skip Group

There are 18 basic instructions in this group which perform tests on Accumulators A and E, Carry Flag, and the Greater Than Flag; the next instruction is skipped or not depending upon the results of the test. The group is subdivided into 2 sections. Within each section the instruction may be combined but between sections they may not.

Mode 11 Instructions - Section 1

| <u>Mnemonics</u> | <u>Octal Code</u> |
|------------------|-------------------|
| ANEG | 007600 |
| ENEG | 006600 |
| AN0 | 007500 |
| EN0 | 006500 |
| SK=C | 006440 |
| CLC | 006020 |
| S=GT | 006410 |
| CLGT | 006004 |
| CLA | 007002 |
| CLB | 006002 |
| CPLA | 007001 |
| CPLB | 006001 |

ANEG or ENEG = Skip if A or E is negative - causes the next instruction to be skipped if the contents of the specified accumulator are negative (ie. if the sign bit is set).

AN0 or EN0 = Skip if A or E is not zero - causes the next instruction to be skipped if the contents of the specified accumulator are not zero (ie. if any of Bits 17-1 is set).

SK=C = Skip if Carry - causes the next instruction to be skipped if the Carry Flag is set.

CLC = Clear Carry - causes the Carry Flag to be reset (cleared).

S=GT = Skip if Greater Than - the next instruction will be skipped if the Greater Than Flag is set.

CLGT = Clear Greater Than - causes the Greater Than flag to be cleared.

CLA or CLE = Clear A or E - sets the specified accumulator (all 17 bits) to zeros.

CPLA or CPLB = Complement A or E - causes the specified accumulator (all 17 bits) to be set to the one's complement of its original value; all ones become zeros and all zeros become ones.

Mode 11 Instructions - Section 2

| <u>Mnemonics</u> | <u>Octal Code</u> |
|------------------|-------------------|
| APOS | 007200 |
| EPOS | 006200 |
| A=0 | 007100 |
| E=0 | 006100 |
| SKNC | 006040 |
| CLC | 006020 |
| SNGT | 006010 |
| CLGT | 006004 |
| CLA | 007002 |
| CLE | 006002 |
| CPLA | 007001 |
| CPLB | 006001 |

APOS or EPOS = Skip if A or E is positive - causes the next instruction to be skipped if the contents of the specified accumulator are positive (ie. if the sign bit is not set).

A=0 or E=0 = Skip if A or E is zero - causes the next instruction to be skipped if the contents of the specified accumulator (all 17 bits) are zero.

SKNC = Skip if Not Carry - causes the next instruction to be skipped if the Carry Flag is not set.

CLC = Clear Carry - causes the Carry Flag to be reset (cleared).

SNGT = Skip if not Greater Than - the next instruction will be skipped if the Greater Than Flag is not set.

CLGT = Clear Greater Than - causes the Greater Than Flag to be cleared.

CLA or CLE = Clear A or E - sets the specified accumulator (all 17 bits) to zeros.

CPLA or CPLE = Complement A or E - causes the specified accumulator (all 17 bits) to be set to the one's complement of its original value; that is, all ones become zeros, and all zeros become ones.

Mode 00 = Odd Instruction Group

There are 25 basic instructions in this group; with the exception of multiple shifts and rotates and set Greater Than very few will be used by the applications programmer as this mode covers processor control instructions. These instructions are not combinable.

Mode 00 Instructions

| <u>Mnemonics</u> | <u>Octal Code</u> |
|--------------------|-------------------|
| NOP | 000000 |
| HALT | 000001 |
| RSTN (I/O RESET) | 000017 |
| MASK | 000002 |
| ACKI | 000003 |
| ION | 000004 |
| IOFF | 000005 |
| SION | 000006 |
| SIOF | 000007 |
| SMOF | 000010 |
| SMON | 000011 |
| PRTY | 000012 |
| PTCT | 000013 |
| BNDY | 000014 |
| MASW | 000015 |
| CONT | 000016 |
| STGT | 001001 |
| SLLAnn | 0017nn |
| SLLBnn | 0007nn |
| RLMAnn | 0016nn |
| RLMBnn | 0006nn |
| SRLAnn | 0015nn |
| SRLBnn | 0005nn |
| RRMAnn | 0014nn |
| RRMBnn | 0004nn |

NOP = No Operation - If all bits are zero then no operation is performed and program control is transferred to the next instruction in sequence. A subroutine to be entered by a JSBR instruction should have a NOP as the first step. The return address is then stored in the location occupied by the NOP during execution of the program. Also any program should be liberally sprinkled with NOPs to allow for flexibility in inserting subsequent amendments.

HALT = Halt - If Bit 1 is set and all other bits are zero, the computer will stop at the conclusion of the current machine cycle.

RSTN = I/O Reset - It must be emphasised that this instruction must not be used in program by application programmers. It is a useful instruction only on switches during debugging, etc., as it clears the Flags of all Input/Output devices connected to the computer. To perform an I/O Reset via switches, the following steps should be taken.

- 1) Load 000017 in octal on the data switches
- 2) Load into A (via switches), then set switches to zero
- 3) Depress Instruction Step Switch once.

MASK = Mask Out - This instructions sets up the Interrupt Disable Flags of each device, according to a pattern or mask set up in Accumulator A - each device's Interrupt Disable Flag is set or cleared as the corresponding bit in the Mask is 1 or 0.

The Mask Bit numbers refer to Data Channel Bits 1 to 17 numeric with ascending Binary order. Every device is wired to a particular data line on the in-out bus and hence to a particular bit of the mask. Although slower devices are assigned to the higher numbered bits in the mask, there is no established priority as the program can use any mask configuration.

ACKI = Acknowledge Interrupt - This instruction determines which is the highest priority device awaiting service by reading its device code into Accumulator B assuming an Interrupt has been called. If Accumulator B is zero in this case, it means an internal interrupt has been called. It can read the code of only one device at a time, whichever of those waiting has the highest priority. This is normally determined by the positions of the I/O Boards in the chassis, ie. the further from the processor, the lower the priority.

IQN = Interrupt On - This instruction sets the Interrupt On Flag to allow the processor to respond to interrupt requests.

I_{OFF} = Interrupt Off - This instruction clears the Interrupt On Flag to prevent the processor from responding to interrupt requests, and also prevents internal processor interrupts, such as:

- Memory Boundary Interrupt
- Mains Failure Interrupt
- Mains On Interrupt
- Memory Parity Interrupt
- Memory Protect Interrupt
- Memory Address = Switch Register Interrupt
- Continuous Interrupt Switch Interrupt

S_{ION} = Skip if Interrupt On - This will skip the next instruction if the interrupt is on, enabling the processor to respond to interrupt requests.

S_{IOF} = Skip if Interrupt Off - This will skip the next instruction if the interrupt is off, preventing the processor from responding to interrupt requests.

S_{MOF} = Skip if Mains Failure Interrupt - This will skip the next instruction in sequence when an internal interrupt is called by a power failure. If the skip is taken, the Interrupt will be reset.

S_{MON} = Skip if Mains on Interrupt - This will skip the next instruction in sequence when an internal interrupt is called when power is restored. If the skip is taken, the interrupt will be reset.

P_{RTY} = Skip if Memory Parity Interrupt - This will skip the next instruction in sequence when an internal interrupt is called in the case of a memory parity failure. If the skip is taken, then interrupt will be reset.

P_{RCT} = Skip if Memory Protect Interrupt - This will skip the next instruction in sequence when an internal interrupt is called by the program selecting an address in a protected section of memory. If the skip is taken, the interrupt will be reset.

E_{NDY} = Skip if Memory Boundary Interrupt - This will skip the next instruction in sequence when an internal interrupt is called by the program specifying an address which is outside the memory. If the skip is taken, the interrupt will be reset.

M_{ASW} = Skip if MA=Switch Register - This will skip the next instruction in sequence when an internal interrupt is called when the memory address equals the address previously set on the Data Switches. If the skip is taken, the interrupt will be reset.

C_{ONT} = Skip if Continuous Interrupt Switch Interrupt - This will skip the next instruction in sequence when an internal interrupt is called after each step.

STGT - Set Greater Than Flag - causes the Greater Than Flag to be set.

SLLAnn/SLLEnn - Multiple left shift of Accumulators - causes Bits 16-1 to be left shifted n times where n is an octal number 1-17 (i.e. decimal 1-15). Has same effect on the Carry Flag as single shift instruction.

SRLAnn/SRLEnn - Multiple right shift of Accumulators - causes bits 16-1 to be right shifted n times where n is an octal number 1-17 (i.e. decimal 1-15). Has same effect on the Carry Flag as single shift.

RLMAnn/RLMEnn - Multiple left rotate of Accumulators - causes bits 16-1 to be left rotated n times where n is an octal number 1-17 (i.e. decimal 1-15). The Carry Flag or bit 17 of the register is not affected.

RRMAnn/RRMEnn - Multiple right rotate of Accumulators - causes bits 16-1 to be right rotated n times where n is an octal number 1-17 (i.e. decimal 1-15). The Carry Flag or bit 17 of the register is not affected.

INPUT/OUTPUT INSTRUCTIONSIntroduction

These are a set of program instructions which, like the Mode 00 Register Instructions, are not used in general programming, as all programmed transfers of information are accomplished by certain set subroutines which are supplied to the programmer, as described later in this manual. However, there now follows a description of the Input/Output instructions which control all transfers of data to and from the peripherals and also perform various operations within the processor. They provide the following general capabilities:

- a) Fix the state of the Busy and Done Flags.
- b) Test the State of the Busy and Done Flags.
- c) Enter data from a specified device into the A or B registers.
- d) Output data to a specified device from the A or B registers.

Input/output instructions are recognised by the computer when the four most significant bits of the instruction word are 0000 and Bit 13 is a 1 (Octal code 01). Bits 6 to 1 select the device that is to respond to the instruction; the format thus allows for 64 codes. In all input/output instructions Bits 11 to 7 specify the complete function to be performed; bits 11 and 10 either controlling or sensing Busy and Done, as shown under Function in the Selection Chart, (fig. 2.17). If Bits 9 to 7 are all set (Mode 111) there is no transfer, and Bits 11 to 10 specify a skip condition. Bit 12, where relevant, specifies A or B register (A=1, E=0).

Instruction Format

| | | |
|------|-------|---|
| Bits | 17-13 | denotes the Operation Code 01 |
| | 12 | denotes which Accumulator the instruction refers to 1= Accumulator A 0= Accumulator E |
| | 11-10 | denote the function bits; the functions are as follows unless Mode 111 is applicable (see below) 00 = no operation 01 = set busy, clear done (start device) 10 = clear busy, clear done (idle device) 11 = input/output pulse |
| | 9-7 | denote the mode |
| | 6-1 | denote the device code ie the means whereby a device is addressed |

The modes available are as follows:

- 000- this has no I/O transfer.
- 001- DTI1 - enables input from register 1 of the specified device (each device can have up to 3 registers or buffers) into Accumulator A or B.
- 010- DTI2 - enables input from register 2 of the specified device, usually into Accumulator B; used to inspect status codes.
- 011- DTI3 - enables input from register 3 of the specified device into Accumulator A or B.
- 100- DTO1 - enables output to register 1 of the specified device from accumulator A or B.
- 101- DTO2 - enables output to register 2 to the specified device from Accumulator A or B.
- 110- DTO3 - enables output to register 3 of the specified device from Accumulator A or B.
- 111- Skip Mode - if this mode is set there is no I/O transfer and the function bits then specify a skip condition.

| | |
|-------------------|-----------------------|
| 00 = skip if busy | 01 = Skip if not busy |
| 10 = Skip if done | 11 = Skip if not done |

Input/Output

Every peripheral device has up to three buffer registers, an Interrupt Disable Flag, Busy and Done Flags and a 6-bit device selection network. This selection network decodes Bits 6-1 of the Input/Output Instruction (which contain the device address), thus ensuring that only the current device responds to signals sent by the processor over the in-out bus. The Busy and Done flags together denote the basic state of the device. When both are clear, the device is idle. The overall sequence of Busy and Done states is determined by both the program and the internal operation of the device.

The data-in or data-out instruction that the program gives in response to the setting of Done can also restart the device. When all data has been transferred, the program generally clears Done so that the device neither requests further interrupts nor appears to be in use. (Busy and Done both set is a meaningless situation).

Any device whose Interrupt Disable Flag is set, cannot cause an interrupt to start and is, therefore, regarded by the program as being of low priority. The Interrupt Disable Flags are used in setting up a priority structure which enables higher priority devices to interrupt an interrupt already in progress. This priority is determined by the use of a mask which controls the states of the Interrupt Disable Flags in the different devices.

Program Interrupts

The use of Input/Output Interrupts in the current program sequence allows apparently simultaneous operation of the main program and a number of peripheral devices. The hardware also allows conditions internal to the processor (e.g. mains failure, memory parity etc.) to signal the program by requesting an Interrupt.

Information transfer is taken care of, in general programming, by the Input and Output subroutines, described later in the manual. To place a device in operation, the program 'starts' it by the instruction 'Set Busy, Clear Done'. If the device is to be used for output (ie. the transfer of data from the processor to the peripheral) the program simultaneously gives a DTO1 instruction which sends the first unit of data - a word or character depending upon how the device handles information - from the specified Accumulator to Register 1 of the device. When it has processed the unit of data, the device clears Busy and sets Done to indicate that it is ready to receive new data for Output, or that it has Data ready for Input. If the Interrupt Disable Flag is clear, the setting of Done signals the program by requesting an Interrupt.

As soon as any Interrupt occurs, the processor stops the execution of the current program and begins to execute the Interrupt Control Routine. It does this by carrying out an automatic JSER, by hardware, to zero page, Column 00, step 02. The result of this action is that the Program Counter, which will contain the address of the next instruction to be performed in the mainstream program, is stored in this location, and the program then transfers control to the next word in sequence (000003). This is an Indirect Jump to the next step, which holds the address of the Interrupt Control Routine (Page 01, Column 12, step 00). When this has been accomplished, the processor is said to have 'honoured' the Interrupt.

The Interrupt Control routine 'services' the Interrupt by:

- a) saving the contents of Accumulators A and B.
- b) saving the Program Counter.
- c) saving the status of the Carry and Greater Than Flags.
- d) saving the current memory bank no.
- e) determining which device requires service (by the use of a look up Table) or whether it's an internal interrupt. The routine identifies the device by giving an Acknowledge Interrupt instruction.
- f) servicing the device, which includes checking the status and either entering data from the buffer into A or B, or putting data from A or B into the buffer, depending on whether it's an Input or Output device.

Once the device has been 'serviced', the Interrupt is 'dismissed' by restoring the contents of Accumulators A and B, the Program Counter and the states of the Carry Flag, Greater than Flag and Memory Bank. The last step of the Interrupt Control routine is an Indirect Jump via zero page, column 00, step 02 (the Program Counter) so that the original program may be resumed at exactly the point the Interrupt occurred. During the 'dismiss' routine Interrupt is turned off so that no further Interrupts are enabled.

Data Channel

The processor contains a data channel which transfers data directly to and from memory by inserting a special memory cycle, rather than by interrupt to a service routine. The data channel is used by devices requiring very high data transfer rates, such as disk.

The following information was obtained from the records of the Department of Health and Human Services, Office of the Assistant Secretary for Health, regarding the activities of the National Center for Human Genome Research. The information was obtained from a review of the records of the Department of Health and Human Services, Office of the Assistant Secretary for Health, regarding the activities of the National Center for Human Genome Research. The information was obtained from a review of the records of the Department of Health and Human Services, Office of the Assistant Secretary for Health, regarding the activities of the National Center for Human Genome Research.

Page 2 of 2

The following information was obtained from the records of the Department of Health and Human Services, Office of the Assistant Secretary for Health, regarding the activities of the National Center for Human Genome Research. The information was obtained from a review of the records of the Department of Health and Human Services, Office of the Assistant Secretary for Health, regarding the activities of the National Center for Human Genome Research. The information was obtained from a review of the records of the Department of Health and Human Services, Office of the Assistant Secretary for Health, regarding the activities of the National Center for Human Genome Research.