# ASSEMBLER Language User Guide

olivetti L1

# ASSEMBLER Language User Guide

# PREFACE

This manual is produced for programmers using the M20 to create Assembly Language programs. The Assembly Language of the Z8001 cpu of the M20 is described in the "M20 Z8000 Assembler Reference Manual". The Reference manual gives the complete instruction set and deals with other aspects of the cpu like operational characteristics, architectural features, etc. This manual supplies additional information to enable the programmer to create Assembly Language programs to run on the M20.

This manual is divided in two parts:
Part I illustrates the characteristics of an M20 source file and describes how an executable binary file can be obtained from a source file.
Part II details all the M20 System Calls.

|  |  |
|---|---|
| REFERENCES: | Z8000 Assembler Reference Manual (code 3982410 M(0)) |
|  | PCOS (Professional Computer Operating System) User Guide (code 3982980 P(0)) |
|  | Basic Language Reference Manual (code 3982430 P(2)) |
|  | I/O with External Peripherals User Guide (code 3982300 N(0)) |
| DISTRIBUTION | General (G) |
| EDITION: | March 1983 |
| RELEASE: | 2.0 |

PUBLICATION ISSUED BY:

Ing. C. Olivetti & C., S.p.A.
Servizio Centrale Documentazione
77, Via Jervis - 10015 IVREA (Italy)

CONTENTS

## APPENDICES

PAGE

# PART I

# 1. INTRODUCTION

## ABOUT THIS CHAPTER

This part of the manual describes how to create Assembly Language programs on the M20. In this chapter a brief step by step description of the process is given. In each step of this description reference is made to the relevant chapter or manual where it is described in detail.

## CONTENTS

## CREATING AN EXECUTABLE FILE

An Assembly Language program must be written in an Editor environment; on the M20 this can be done in the Video File Editor environment which is described in the "M20 PCOS (Professional Computer Operating System) User guide". This edited version of the program is known as the source file. The source file is described in chapter 2, where the Directives and the Assembler Conventions for the M20 are defined. Chapter 2 ends with a description of the PCOS Standard, which defines the format of a source file meant to execute like any PCOS routine.

The next step is to assemble the program using the ASSEMBLER (ASM) command. This command takes a source file as input and outputs a z-type object file. The ASM command is described in chapter 3.

The final step in creating an executable file is performed by the LINK command which is described in chapter 4. LINK takes one or more object files as input and outputs a single executable binary file. Note that z-type object files created using other computer languages can be linked to z-type object files output by the ASM command.

The process of creating an executable file is shown schematically in fig 1-1

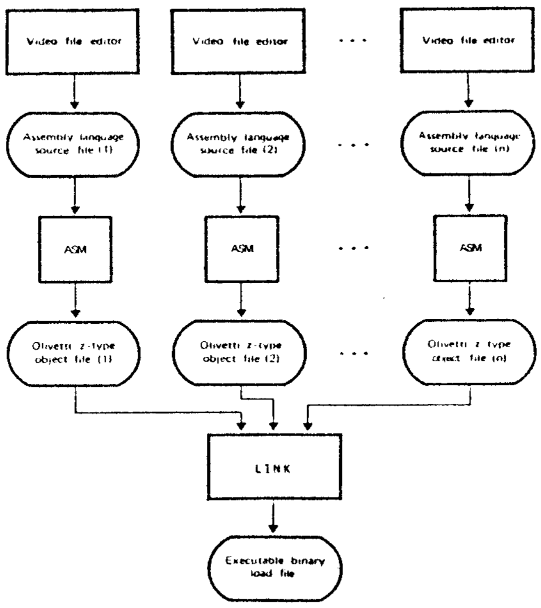

Video file editor
Video file editor
Video file editor
Assembly language source file (1)
Assembly language source file (2)
Assembly language source file (n)
ASM
ASM
ASM
Olivetti z-type object file (1)
Olivetti z-type object file (2)
Olivetti z-type object file (n)
LINK
Executable binary load file

Fig. 1-1 Creating an executable binary file

## THE M20 ASSEMBLER PACKAGE

The M20 Assembler package also includes the PDEBUG (Program DEBUG) utility, detailed in chapter 5, and three auxilliary commands, TEXTDUMP, HDUMP and MLIB described in chapter 6. All the Assembler and Video File Editor routines must be invoked from the PCOS environment.

## SYSTEM CONFIGURATION

The Assembler package can run on any M20 system configuration.

# 2. THE ASSEMBLER SOURCE FILE

# THE ASSEMBLER SOURCE FILE

## ABOUT THIS CHAPTER

This chapter contains the main steps to be taken and the Assembler conventions the programmer must adhere to, in order to build source files for the user's own utilities.

## CONTENTS

# THE ASSEMBLER SOURCE FILE

## INTRODUCTION

As previously mentioned, to construct the source file, the programmer will make use of the Video File Editor (as described in the "PCOS (Professional Computer Operating System) User Guide"), by means of which he can insert the instructions and the Assembler directives. The instruction set used is precisely that of the Z-8001 CPU, described in detail in the "M20 Z8000 Assembler Reference Manual", which is useful to the programmer for what regards mnemonics, addressing and machine code. As far as the Assembler conventions and directives are concerned, however, (which are M20 specific), these will be examined in more detail in the next two sections entitled "Assembler Conventions" and "Assembler Directives".

The section on "Assembler Conventions" describes in depth the way to represent operands, numerical constants, strings, comments, arithmetic operations, which may appear on a source program line.

The next section provides a description of the "Assembler Directives" i.e. those instructions which are not translated by the Assembler in executeable machine code, but which are used by the Assembler itself to leave uninitialised space in the object program, define strings within the program, make references to variables outside the program and to perform operations which facilitate the programmer's work.

The last section "The PCOS Standard" deals with the structure an Assembler source file must have, so that the user can build himself a utility which is coherent with the PCOS utilities standards, for invoking and for passing parameters.

## ASSEMBLER CONVENTIONS

### ASSEMBLY LANGUAGE STATEMENT FORMAT

The most fundamental component of an assembly program is the assembly language statement, a single line of text consisting of an instruction and its operands, with an optional comment. The instruction describes an action to be taken; the operands supply the data to be acted upon.

An assembly language statement can include four fields in the following order, from left to right on the line:

- Symbolic Label;
- Instruction Mnemonic;
- Operands;
- Comment.

All fields can be optional depending on the instruction chosen. Each field of the statement must be separated from the others by white space (one or more spaces or tabs). If a field other than the symbolic label is to be omitted but subsequent fields on the line are not, it may be coded as a solitary comma (,). Fields other than the comment field may not contain white space except for the case of character constants or strings in operands (which are enclosed in apostrophes or quotation marks respectively).

## Symbolic Label Field

Any statement may contain a symbolic label. Some instructions require it. If provided, the label must begin with the first character of the text line. The absence of the field is indicated by the first character of the line being a white space character. The only way in which a symbol may be defined anywhere in the assembly is for it to appear in the label field of a statement. A particular symbol may appear only once in a label field within one module. Note: a comment line, which is not an assembly instruction, is indicated by the first character of the line being an asterisk (*).

## Instruction Field

The instruction is the assembly-language mnemonic describing a specific action to be taken. This may represent either a Z8000 machine instruction or an assembler directive instruction. The instruction must be separated from its operands by white space (one or more spaces or tabs).

```
  LD  R2,ALPHA    Load register 2 from memory location ALPHA

  JP  BETA             JUMP to location BETA
```

Many of the operations of the Z8000 can be applied to word, byte, or long operands. A simple naming convention has been adopted to distinguish the size of the operands for these particular instructions: the suffix "B" designates a byte instruction, the suffix "L" designates a long word instruction, and no suffix designates a word instruction:

```
        ADD     R0,R1           Add word operands

        ADDB    RH0,RL0         Add byte operands

        ADDL    RR0,RR2         Add long operands
```

## Operand Field

Depending on the instruction specified, this field can have zero or more operands. If two or more operands are needed, each must be separated by a comma with no intervening white space. If there are no operands and a comment field is to be placed on the same statement, the operand field must be a single comma standing alone.

```
    RET   ,                        No operand

    TEST  R2                       One operand

    LD    R2,R1                    Two operands

    LDM   R2,ALPHA,#7              Three operands

    CPD   R2,@R4,R6,EQ             Four operands
```

Operands supply the information the instruction needs to carry out its action. An operand of a Z8000 machine instruction can be:

- Data to be processed (immediate data);
- The address of a location from which data is to be taken (source address);
- The address of a location where data is to be put (destination address);
- The address of a program location to which program control is to be passed;
- A condition code, used to direct the flow of program control.

Although there are a number of valid combinations of operands, there is one basic convention to remember: the destination operand always precedes the source operand. Refer to the specific instructions in the Reference Manual for valid operand combinatons.

Immediate data can be in the form of a constant , an address , or an expression (constants and/or addresses combined by operators).

```
  LD    R2,#7         Load 7 into register 2

  LD    R2,#ALPHA     Load address of ALPHA into register 2

  LD    R4,#BETA/2    Load value of expression [BETA/2] into
                      register 4
```

As far as the conventions are concerned, for expressing numeric constants and alphanumeric strings, these will be dealt with later in the appropriate section.

Source, destination, and program addresses can also take several forms. Addressing modes are described in detail later. Some examples are:

```
LD    R1,@R2              Load value whose address is in register 2
 ,     ,                  into register 1

LD    R1,ALPHA            Load value located at address labeled
 ,     ,                  ALPHA into register 1

LD    R1,ALPHA+1          Load value at location following that
 ,     ,                  addressed by ALPHA into register 1

JP    EQ,BETA             Jump to program address labeled BETA if
 ,     ,                  EQ flag is set

JP    NE,BETA+16          Otherwise, jump to location sixteen bytes
 ,     ,                  following BETA
```

Condition codes are listed in the Reference Manual.

Operands of an assembler directive instruction can be:

- A numerical value or expression;
- Expressions or strings representing initialization data;
- A string such as a file name, a module name, or a section name (such strings cannot be referenced elsewhere in the program);
- A keyword.

Examples of assembler directives:

```
MODULE      device.1,segmented

AT          BETA+16

DSB         27

DDL         %7F01FFF,'AB'
```

The assembler directives are dealt with later in the appropriate section.

## Comments

Comments are used to document program code as a guide to program logic and also to simplify present or future program debugging A text line which begins with an asterisk as the first non-white-space character is copied as it appears to the listing file but is ignored by the assembler for all other purposes.

Examples of comment lines:

```
*   This routine is used to compare two strings.  The operands are
*   pointers to the first characters of each string.  The
*   strings are of variable length with a zero byte marking
*   the end of the string.

*   The returned value of this routine is:

*           -1:  first string less than second
*            0:  strings equal
*            1:  first string greater than second
```

Comments may also be placed on the end of each assembler statement All text which appears after the operand field on the line is a comment and is reproduced in the listing file but ignored otherwise. If the operand field or the instruction field are to be omitted the comment field may only be included if the omitted field(s) are coded as a solitary comma (,).

Examples of on-statement comments:

```
         CLR     R2                 Initialize register 2

         IRET    ,                  return from the interrupt NOW!

START.UP ,       ,                  THIS IS THE ENTRY POINT OF THE PROGRAM

         JP      Z,BETA+12 this is a close comment
```

## SYMBOLS, CONSTANTS, and STRINGS

### Symbols

A symbol may consist of the letters A-Z (upper or lower case), the digits 0-9, the underscore character (_), or a period (.). A symbol may not begin with a digit (0-9). The maximum length of a symbol is 16 characters.

Upper and lower case letters are considered different characters. Thus "Start" and "start" are different symbols.

The following are valid symbols:

```
        ValueAssignments
        Initial_values
        start_up
        Pass_2
        sort
```

## Constants

A constant is a value which stands for itself. It may be either a number or a character sequence.

Numbers can be written in decimal, hexadecimal, binary, or octal notation. The latter three are preceded by a percent sign (%) and, in the case of binary and octal, by a base specifier enclosed in parentheses. If a number has no prefix, decimal is assumed.

| | |
|---|---|
| 42 | decimal |
| %42 | hexadecimal |
| %(8)42 | octal |
| %(2)10110010 | binary |

A character sequence is a sequence of one to four characters enclosed in apostrophes. Any ASCII character can be included in the character sequence, for example;

```
'A'
'Open'
```

A character can also be represented in a character sequence in the form "%hh," where "hh" is the hexadecimal equivalent of the ASCII code for the character, for example;

```
'E=%1B'
```

For convenience, certain ASCII characters have been assigned shorter, more mnemonic codes as follows:

| | |
|---|---|
| %L or %1 | Linefeed |
| %T or %t | Tab |
| %R or %r | Carriage Return |
| %P or %p | Page (Form Feed) |
| %% | Percent Sign |
| %Q, %q, %' | Apostrophe (Single Quote) |

Example:

```
        '1%r2%r'     represents the ASCII sequence: 1 /CR/ 2 /CR/
and     '%Qt=%Q'     represents the ASCII sequence: 't='
```

## Strings

Strings are sequences of any length of ASCII characters, enclosed in quotation marks. They can be defined only by using the DDB directive (see Data Generation Directives).

Strings also use the above ASCII mnemonic forms. Since strings are enclosed in quotation marks, the mnemonic %" is used for embedded quotation marks.

## ARITHMETIC OPERANDS

### Run-Time and Assembly-Time Arithmetic

Arithmetic is performed in two ways in an assembly language program. Run-time arithmetic is done while the program is actually executing.

```
  ADDB RHO,RL2   Add the contents of register
  ,    ,         RL2 to the contents of register RHO
```

Assembly-time arithmetic is done by the assembler when the program is assembled and involves the evaluation of arithmetic expressions in operands, such as the following:

```
  LDL RR14,#[2*one+%10]

  JP  Z,BETA+34

  AND R5,ALPHA-3
```

Assembly-time arithmetic is more limited than run-time arithmetic.

All assembly-time arithmetic is computed using 32-bit representations of the numbers. Any number in excess of 32 bits (4,294,967,296) loses the extra bits on the left, so all values are calculated "modulc 4,294,967,296". Depending on the number of bits required by the particular instruction, only the rightmost 4, 8, 16, or 32 bits of the resulting 32-bit value are used. If the result of assembly-time arithmetic is to be stored in four bits, the value is taken "modulo 16" to give a result in the range 0 to 15. If the result is to be stored in a single byte location, the value is taken "modulo 256" to give a result in the range 0 to 255 (or -128 to 127 if signed representation is intended). If the result is to be stored in a word, the value is taken "modulo 65536" to give a result in the range 0 to 65535 (or -32768 to 32767 if signed representation is intended).

```
   LDB    RH7,#one*2          Result of "one*2" must be in
*                             range 0 to 255

   JP     BETA+2              Modulo 65536.  Result is the
*                             address 2 bytes beyond BETA

   SUBL   RR2,#one*%80000     Result of "one*%80000" is taken
*                             modulo 4,294,967,296
```

## SYMBOLIC VALUES

A symbol can be assigned a value other than that of the current assembly location counter by means of the assembler directive instructions which are described later in this chapter. In this way a symbol can be made to represent an absolute constant value or a relocatable memory location in the same section, in a different section of the same module or in a completely different module. That symbol may then be used in operand expressions anywhere that a value of its type is permissible.

## EXPRESSIONS AND OPERATORS

Expressions are formed using arithmetic, logical, shift, and relational operators in combination with constants and variables. These operators allow both unary (one-operand) and binary (two-operand) expressions, as shown below.

### Arithmetic Operators

The arithmetic operators are the following:

| Operator | Operation |
|---|---|
| + | Unary plus, binary addition |
| - | Unary minus, binary subtraction |
| * | Multiplication |
| / | Divison |
| \ | Modulus |

The division operator (/) truncates any remainder. The modulus operator (\) performs the modulo function (i.e. returns the remainder after division)

```
9/2 = 4

9\2 = 1

-9/2 = -4
```

If zero is specified as the right operand for either of these operators, the result is undefined.

Examples:

| | |
|---|---|
| SUBB RL0,#1 | 1 is subtracted from RL0 |
| SUB R10,#one+[10-3] | Value of one + 7 is subtracted from register 10 |

## Logical Operators

The logical operators are the following:

| Operator | Operation |
|---|---|
| ~ | (Unary) Logical COMPLEMENT |
| & | Logical AND |
| ! | Logical OR |
| ^ | Logical EXCLUSIVE OR |

Logical COMPLEMENT (~) simply complements the bit pattern of its single operand (i.e. all one bits are changed to zero and vice-versa).

```
LD   R11,#~CONSTANT1   Reverse the bits of CONSTANT1 and load into
                       reg 11
```

The effect of Logical AND, Logical OR, and Logical EXCLUSIVE OR can be seen from the following examples. Although 32-bit arithmetic would actually be done by the assembler, 4-bit arithmetic is shown for clarity. Assume two constants, CONSTANT1 and CONSTANT2, which have the bit patterns 1100 and 1010, respectively. The expressions:

```
CONSTANT1&CONSTANT2
CONSTANT1!CONSTANT2
CONSTANT1^CONSTANT2
```

will result in the following evaluations of the operands:

```
AND   1100         OR   1100        EXCLUSIVE OR   1100
      1010              1010                       1010
      ----              ----                       ----
      1000              1110                       0110
```

The assembly-time logical operations performed by Logical COMPLEMENT, Logical AND, Logical OR and Logical EXCLUSIVE OR can also be done at run time by the Z8000 instructions COM, AND, OR, and XOR respectively. The assembly-time operations require less code and register manipulation. The run-time operations allow greater flexibility, however. For example, they can operate on registers (variables) whose contents are not known at assembly time, as well as on known constant values.

## Shift Operators

The shift operators are as follows:

| | |
|---|---|
| {SHR} | Logical shift right |
| {SHL} | Logical shift left |

When used in expressions, the shift operators have the form

```
d operator n
```

where "d" is the data to be shifted and "n" specifies the number of bits to be shifted. Vacated bits are replaced with zeros. For example, if CONSTANT1 has a value of 00001100, the statement

```
LD      R10,#[CONSTANT1{SHL}2]
```

would load the value 00110000 into register R10. If the second operand supplied is negative (that is, if the sign bit is set), it has the effect of reversing the direction of the shift.

```
  LD    R10,#[CONSTANT1{SHR}-2]       CONSTANT1 is shifted
*                                     two bit positions LEFT
```

## Relational Operators

There are two basic types of relational operators: those which consider their operands to be signed 32-bit integers, and those which consider their operands to be unsigned 32-bit integers.

Signed:

| | |
|---|---|
| < | Less than |
| <= | Less than or equal |
| = | Equal |
| <> | Not equal |
| >= | Greater than or equal |
| > | Greater than |

Unsigned:

| | |
|---|---|
| {ULT} | Less than |
| {ULE} | Less than or equal |
| {UEQ} | Equal |
| {UNE} | Not equal |
| {UGE} | Greater than or equal |
| {UGT} | Greater than |

The relational operators return a logical TRUE value (all ones) if the comparison of the two operands is true, and return a logical FALSE value (all zeros) otherwise.

```
LD  R0,#[1=2]        Reg 0 is loaded with zeros
LD  R0,#[2+2]<5      Reg 0 is loaded with ones
```

## Precedence of Operators

Expressions are generally evaluated left to right with operators having the highest precedence evaluated first. If two operators have equal precedence, the leftmost is evaluated first.

The following lists the assembly-time operators in order of precedence:

- Unary operators: +, -, ~
- Multiplication/Division/Modulus/Shift/AND: *, /,\, {SHR}, {SHL},&
- Addition/Subtraction/OR/XOR: +, -, !,^
- Relational operators:
  <, <=, =,<>, >=, >, {ULT}, {ULE}, {UEQ}, {UNE}, {UGE}, {UGT}

Square brackets ([ ]) can be used to change the normal order of precedence. Items enclosed in brackets are evaluated first. If brackets are nested, the innermost are evaluated first.

```
100/4 - 48/2 = 1
100/[4 - 48/2]= -5
```

**Note:**
Square brackets are used instead of the traditional parentheses. This is done to avoid all confusion and conflict whether it be syntactical, semantical or conceptual, with the indexed address operand forms described further on in this chapter.

## Segmented Address Operators

Two special operators are provided to ease the manipulation of segmented addresses. While addresses can be treated as a single value with a symbolic name assigned by the programmer, occasionally it is useful to determine the segment number or offset associated with a memory location.

The SEGMENT unary operator, { SEGMENT } , is applied to an address expression that contains a symbolic name associated with an address, and returns a 16-bit value. This value is the 7-bit segment number associated with the expression and a one bit in the most significant bit of the high-order byte, and all zero bits in the low-order byte.

The "OFFSET" unary operator, {OFFSET} , is applied to an address expression and returns a 16-bit value which is the offset value associated with the expression.

### Example

```
*    Load the segmented address of buffer_pointer into register pair RR12.

          LD   R12,#{SEGMENT}buffer_pointer
          LD   R13,#{OFFSET}buffer_pointer

*    This is functionally equivalent to the following statement:

          LDL  RR12,#buffer_pointer
```

Because of the special properties of these address operators, no other operators can be applied to a subexpression containing a SEGMENT or OFFSET operator, although other operators can be used within the subexpression to which the operator is applied:

```
          {SEGMENT}[buffer_pointer+4]      Valid
         [{SEGMENT}buffer_pointer]+4       Invalid
        - [{OFFSET}buffer_pointer]         Invalid
```

## Z8000 ADDRESSING MODES

With the exception of immediate data and condition codes, all Z8000 machine instruction operands are expressed as addresses: register, memory, and I/O addresses. The various address modes recognized by the Z8000 assembler are as follows:

- Immediate Data
- Register
- Indirect Register
- Direct Address
- Indexed Address
- Relative Address
- Based Address
- Based Indexed Address

Special characters are used in operands to identify some of these address modes. The characters are:

- "R" preceding a word register number;
- "RH" or "RL" preceding a byte register number;
- "RR" preceding a register pair number;
- "RQ" preceding a register quadruple number;
- "@" preceding an indirect-register reference;
- "#" preceding immediate data;
- "()" used to enclose the displacement part of an indexed, based, or based indexed address;
- "$" signifying the current program counter location, usually used in relative addressing.

### Immediate Data

The operand value used by the instruction in Immediate Data addressing mode is the value supplied in the operand field itself.

Immediate data is preceded by the special character "#" and can be either a constant (including character constants and symbols representing constants) or an expression as previously described. Immediate data expressions are evaluated using 32-bit arithmetic. Depending on the instruction being used, the value represented by the rightmost 4, 8, 16, or 32 bits is actually used. An error message is generated for values that overflow the valid range for the instruction.

```
        ADDB  RL7,#98        Add 98 to the contents of register RL7

        LDL   RR14,#6789*FOUR
        ,     ,                  Load the value of the multiplication
        ,     ,                  into register pair 14
```

If a variable name or address expression is prefixed by '#', the value used is the address represented by the variable or the result of the expression evaluation, not the contents of the corresponding data location.

The assembler automatically creates the proper format for a long offset address which includes the segment number and the offset in a 32-bit value. It is recommended that symbolic names be used wherever possible, since the corresponding segment number and offset for the symbolic name will be automatically managed by the assembler and can be assigned values later when the module is linked or when the program is loaded for execution.

For those cases where a specific segment is desired, the following notation can be used (the segment designator is enclosed in double angle brackets):

```
          <<segment>>offset
```

where "segment" is a constant expression that evaluates to a 7-bit value, and "offset" is a constant expression that evaluates to a 16-bit value. This notation is expanded into a long offset address by the assembler.

```
    LDL   RR2,#MESSAGE           Load the address of MESSAGE into
    ,     ,                      register pair RR2


    LDL   RR2,#<<2>>%10          Load the segmented address
    ,     ,                      with segment 2, offset %10
    ,     ,                      into register pair RR2
```

**Register Address**

In register addressing mode, the operand value is the content of the specified general-purpose register. There are four different sizes of registers on the Z8000:

- Word register (16 bits),
- Byte register (8 bits),
- Register pair (32 bits), and
- Register quadruple (64 bits).

A word register is indicated by the "R" followed by a number from 0 to 15 (decimal) corresponding to the 16 registers of the machine. Either the high or low byte of the first eight registers can be accessed by using the byte register constructs "RH" or "RL" followed by a number from 0 to 7. Any pair of word registers can be accessed as a register pair by using "RR" followed by an even number between 0 and 14. Register quadruples are equivalent to four consecutive word registers and are accessed by the notation "RQ" followed by one of the numbers 0, 4, 8, or 12.

If an odd register number is given with a register pair designator, or a number other than 0, 4, 8, or 12 is given for a register quadruple, an assembly error will result.
In general, the size of a register used in an operation depends on the particular instruction. Byte instructions, which end with the suffix "B" are used with byte registers. Word registers are used with word instructions, which have no special suffix. Register pairs are used with long word instructions, which end with the suffix "L". Register quadruples are used only with three instructions (DIVL, EXTSL and MULTL) which use a 64-bit value. An assembly error will occur if the size of a register does not correspond correctly with the particular instruction.

```
LD     R5,#%5A5A         Load register 5 with the
,      ,                 hexadecimal value 5A5A

LDB    RH3,#%A5          Load the high order byte of
,      ,                 word register 3 with the
,      ,                 hexadecimal value A5

ADDL   RR2,RR4           Add the register pairs 2-3 and
,      ,                 4-5 and store the result in 2-3

MULTL  RQ8,RR12          Multiply the value in register
,      ,                 pair 10-11 by the value in
,      ,                 register pair 12-13 and store the
,      ,                 result in register quadruple
,      ,                 8-9-10-11
```

## Indirect Register Address

In Indirect Register addressing mode, the operand value is the content of the location whose address is contained in the specified register. A register pair is used to hold the address. Any general-purpose register (register pair) can be used except R0 or RR0.

Indirect Register addressing mode is also used with the I/O instructions and always indicates a 16-bit I/O address. Any general-purpose word register can be used except R0.
An Indirect Register address is specified by a "commercial at" symbol (@) followed by either a word register or a register pair designator. For Indirect Register addressing mode, a word register is specified by an "R" followed by a number from 1 to 15, and a register pair is specified by an "RR" followed by an even number from 2 to 14.

```
        LD    @RR2,#15    Load immediate value 15 into
        ,     ,           location addressed by register
        ,     ,           pair 2-3
```

**Direct Address**

The operand value used by the instruction in Direct addressing mode is the content of the location specified by the address in the instruction. A direct address can be specified as a symbolic name of a memory or I/O location, or an expression that evaluates to an address. For all I/O instructions, the address is a 16-bit value. The memory address is either a 16-bit value (short offset) or a 32-bit value (long offset). All assembly-time address expressions are evaluated using 32-bit arithmetic.

```
    LD    R10,TABLE              Load the contents of the
    ,     ,                      location addressed by TABLE
    ,     ,                      into register 10

    LD    ARRAY+2,R2             Load the contents of register
    ,     ,                      2 into the location addressed
    ,     ,                      by adding 2 to ARRAY

    LDB   RH5,55                 Load the contents of the I/O
    ,     ,                      location addressed by 55 into
    ,     ,                      RH5
```

The assembler automatically creates the proper format which includes the segment number and the offset. It is recommended that symbolic names be used wherever possible, since the corresponding segment number and offset for the symbolic name will be automatically managed by the assembler and can be assigned values later when the module is linked or loaded for execution.

For those cases where a specific segment is desired, the following notation can be used (the segment designator is enclosed in double angle brackets):

```
        <<segment>>offset
```

where "segment" is a constant expression that evaluates to a 7-bit value, and "offset" is a constant expression that evaluates to a 16-bit value. This notation is expanded into a long offset address by the assembler.

To force a short offset address, the segmented address can be enclosed in vertical bars ( || ). In this case, the offset must be in the range 0 to 255, and the final address includes the segment number and short offset in a 16-bit value.

```
LD  R10,|TABLE|              Load the contents of the
 ,   ,                       location addressed by TABLE
 ,   ,                       (short offset format) into
 ,   ,                       register 10

LD  <<SEGMENT>>OFFSET,R10    Load the contents of reg-
 ,   ,                       ister 10 into the location
 ,   ,                       addressed by the segment
 ,   ,                       named SEGMENT offset by
 ,   ,                       OFFSET (long offset format)

JP  |<<SEGMENT>>OFFSET|      Jump to location addressed
 ,   ,                       by segment, offset
 ,   ,                       (short offset format)
```

**Indexed Address**

An Indexed address consists of a memory address displaced by the contents of a designated word register (the index). This displacement is added to the memory address and the resulting address points to the location whose contents are used by the instruction. The memory address is specified as an expression that evaluates to either a 16-bit value (short offset) or a 32-bit value (long offset). All assembly-time address expressions are evaluated using 32-bit arithmetic. This address is followed by the index, a word register designator enclosed in parentheses. For Indexed addressing, a word register is specified by an "R" followed by a number from 1 to 15. Any general-purpose word register can be used except R0.

```
LD  R10,TABLE(R3)          Load the contents of the
 ,   ,                     location addressed by TABLE
 ,   ,                     plus the contents of reg-
 ,   ,                     ister 3 into register 10
```

The assembler automatically creates the proper format for the memory address, which includes the segment number and the offset. As with Direct addressing, symbolic names should be used wherever possible so that values can be assigned later when the module is linked or loaded for execution.

For those cases where a specific segment is desired, the following notation can be used (the segment designator is enclosed in double angle brackets):

```
<<segment>>offset(r)
```

where "segment" is a constant expression that evaluates to a 7-bit value, "offset" is a constant expression which evaluates to a 16-bit value, and "r" is a word register designator. This notation is expanded into a long offset address by the assembler.

To force a short offset address, the segmented address may be enclosed in vertical bars (||). In this case, the offset must be in the range 0 to 255, and the final address includes the segment number and short offset in a 16-bit value.

```
    LD  R10,|TABLE|(R3)           Load the contents of the
    ;   ;                         location addressed by
    ;   ;                         TABLE (short offset format)
    ;   ;                         plus the contents of reg-
    ;   ;                         ister 3 into register 10

    LD  <<5>>8(R13),R10           Load the contents of regis-
    ;   ;                         ter 10 into the location ad-
    ;   ;                         dressed by segment 5
    ;   ;                         offset by 8 (long off-
    ;   ;                         set format) plus the con-
    ;   ;                         tents of register 13
```

## Relative Address

Relative address mode is implied by its instruction. It is used by the Call Relative (CALR), Decrement and Jump If Not Zero (DJNZ), Jump Relative (JR), Load Address Relative (LDAR), and Load Relative (LDR) instructions and is the only mode available to these instructions. The operand, in this case, represents a displacement that is added to the contents of the program counter to form the destination address that is relative to the current instruction. The original content of the program counter is taken to be the address of the instruction byte following the instruction. The size and range of the displacement depends on the particular instruction, and is described with each instruction in the Z8000 Assembler Reference Manual.

The displacement value can be expressed in two ways. In the first case, the programmer provides a specific displacement in the form "$+n" where n is a constant expression in the range appropriate for the particular instruction and $ represents the contents of the program counter at the start of the instruction. The assembler automatically subtracts the value of the address of the following instruction to derive the actual displacement.

```
  JR OV,$+ONE   Add value of constant ONE to program
  ; ;           counter and jump to new location if
  ; ;           overflow has occurred
```

In the second case, the assembler calculates the displacement automatically. The programmer simply specifies an expression that evaluates to a number or a program label as in Direct addressing. The address specified by the operand must be in the valid range for the instruction, and the assembler automatically subtracts the value of the address of the following instruction, to derive the actual displacement.

```
DJNZ R5,BETA        Decrement register 5 and jump to
,    ,              BETA if the result is not zero

LDR  R10,ALPHA      Load the contents of the location
,    ,              addressed by ALPHA into register 10
```

## Based Address

A Based address consists of a register that contains the base and a 16-bit displacement. The displacement is added to the base and the resulting address indicates the location whose contents are used by the instruction.

The segmented based address is held in a register pair that is specified by an "RR" followed by an even number from 2 to 14. Any general-purpose register pair can be used except RR0. The dispacement is specified as an expression that evaluates to a 16-bit value, preceded by a '#' symbol and enclosed in parentheses.

```
LDL    RR2,R1(#255)        Load into register pair 2-3 the
,      ,                   long word value found in the
,      ,                   location resulting from adding
,      ,                   255 to the address in register1

LD     RR4(#%4000),R2      Load register 2 into the loca-
,      ,                   tion addressed by adding %4000
,      ,                   to the segmented address found
,      ,                   in register pair 4-5
```

## Based Indexed Address

Based Indexed addressing is similar to Based addressing except that the displacement (index) as well as the base is held in a register. The contents of the registers are added together to determine the address used in the instruction.

The segmented based address is held in a register pair that is specified by an "RR" followed by an even number from 2 to 14. Any general-purpose register pair can be used except RR0. The index is held in a word register that is specified by an "R" followed by a number from 1 to 15. Any general-purpose word register can be used except R0.

```
LDB    RR14(R4),RH2     Load register RH2 into the
,      ,                location addressed by the seg-
,      ,                mented address in RR14 indexed by
,      ,                the value in R4
```

## ASSEMBLER DIRECTIVES

Assembler Directives are program statements which have the same format as machine instructions but whose operation field does not correspond to any machine instruction mnemonic. These are used to control the operation of the assembler with regard to functions other than producing the machine code for an instruction.

Directives fall into two major categories: data generation directives which allocate and possibly initialize program data areas, and control directives which control and affect the operation of the assembler.

## DATA GENERATION DIRECTIVES

These cause data space to be reserved at the current assembly location. Directives differ in element size and ability to initialize the data space.

### DS

This directive is used to define uninitialized data. It takes a single required operand which is an expression which evaluates to an absolute value (i.e. not relocatable). No forward referencing of symbols is allowed in the expression. The given number of two-byte words is reserved at the current location, after rounding up to the next even boundary. Note that an operand of "0" may be used to force rounding of the location counter up to an even boundary without reserving any space for data. Also, if a label is defined in the label field of the same statement its value is set to that of the location counter after the rounding operation, but before the data definition.

```
        DS    0       round up to next word boundary

BUFFER  DS    100     reserve a one hundred-word buffer
```

### DSB

This directive is used to define uninitialized data. It takes a single required operand which is an expression which evaluates to an absolute value (i.e. not relocatable). The given number of bytes is reserved at the current location. No forward referencing of symbols is allowed in the expression.

```
                 DSB    100              reserve 100 bytes

keyboard_buffer  DSB    number_base_16   define keyboard_buffer
```

### DSL

This directive is used to define uninitialized data. It takes a single required operand which is an expression which evaluates to an absolute value (i.e. not relocatable). No forward referencing of symbols is allowed in the expression. The given number of four-byte longwords is reserved at the current location, after rounding up to the next even boundary. Note that an operand of "0" may be used to force rounding of the location counter up to an even boundary without reserving any space for data. Also, if a label is defined in the label field of the same statement its value is set to that of the location counter after the rounding operation, but before the data definition.

```
                       DSL     100         leave exactly 400 bytes
*                                          uninitialized

     buffer_pointer    DSL     1           define memory pointer
*                                          variable
```

### DD

The DD directive is used to define initialized data areas consisting of two-byte word values. The directive may take any number of operands and repetition factors may be applied to groups of them (described below). Each operand is an expression which evaluates to either an absolute value or to a relocatable value. In either case only the low-order 16 bits of the value is used. One word of data is generated for each operand supplied at the current location after rounding up to the next even boundary. Also, if a label is defined in the label field of the same statement its value is set to that of the location counter after the rounding operation, but before the data definition.

```
        DD      10244       define one word with contents 10,244 (%2804)

    * Define a power-of-two table of words:

    TABLE   DD  0,1,2,4,8,16,32,64,128
            DD  %100,%200,%400,%800,%1000,%2000,%4000,%8000

    Key     DD  'A'         define word containing %0041
```

### DDB

The DDB directive is used to define initialized data areas consisting of byte values. The directive may take any number of operands and repetition factors may be applied to groups of them (described below). Each operand is an expression which evaluates to an absolute value, or a string.

If the operand is a value, only the low-order 8 bits are used and one byte of data is generated at the current location.

```
        DDB      'A'!%40,['Z'+1]!%40      two data bytes
```

String operands are sequences of any length (including zero) of ASCII characters. They are delimited by quotation marks, so an embedded quotation mark is written %" and an embedded percent sign is written %%. The discussion of hexadecimal and mnemonic equivalents for ASCII characters (see Constants) applies as well to strings. One byte of data is generated for each byte of a string, at the current location.

```
string  DDB      "this is a string"

EndOff  DDB      7,%0D,%0A             bell, carriage return, line feed

MESSAGE DDB      "ERROR - INVALID INPUT%r",7,0
```

**DDL**

DDL is used to define initialized data areas consisting of four- byte long values. The directive may take any number of operands and repetition factors may be applied to groups of them (described below). Each operand is an expression which evaluates to either an absolute value or to a relocatable value. Two words of data are generated for each operand supplied at the current location after rounding up to the next even boundary. Also, if a label is defined in the label field of the same statement its value is set to that of the location counter after the rounding operation, but before the data definition.

```
*  Define table of three long words, the address of the
*  start of the region, the address of the end of the
*  region and the size in byte of the region.

     DDL      START,END,END-START

     DDL      %7f017fff,'AB'     define two long words the first
     ,        ,                  containing hex 7f017fff, and the
     ,        ,                  second hex 00004142
```

The DD, DDB and DDL directives each take an arbitrary number of operands and allow repetition factors to be applied to them. A repetition factor takes the form of an absolute expression. The repetition factor must be followed by the operand enclosed in parentheses. This has the effect of the enclosed operands appearing in sequence, the number of times given by the expression.

Repetitions may be nested. No forward referencing of symbols is allowed.

```
ARRAY   DD   1000(0)     define array of 1000 words,
*                        all initialized to zero.

* define and initialize 8 bytes
CrcTab  DDB  2("asdf")   which would be 8 bytes.
```

The DD directives with repetition factors have the potential to produce voluminous listings. If the generated code is too large to fit the space to the left of the source line, the code will follow the listing line in groups of 8, 16, or 32 data elements (for DDL, DD, and DDB respectively).

## CONTROL DIRECTIVES

### MODULE

A MODULE statement defines the beginning of each module in the source file. It must occur as the first instruction of each module in the input source file. A module ends either at the next MODULE statement or at the end of the input source file. Modules within the same file are completely unrelated; no symbols may be shared or passed between them.

The first operand of the MODULE statement, the module name, is required. This operand follows the composition rules of a normal symbol, but cannot be referenced elsewhere in the program. The second operand is also required. It must be the keyword "SEGMENTED" to tell the module to contain code for a segmented Z8000.

```
MODULE test_seg,segmented
```

### SECTION

A module is composed of sections which are named explicitly by the user. A section is the smallest unit of relocatability within the programming system. Portions of the same section cannot be split further and placed separately at link time.

A SECTION directive must appear in each module before the first machine instructions or data generating directive. The SECTION directive has one required operand which is the section name. This operand follows the composition rules of a normal symbol, but cannot be referenced elsewhere in the progam.

If a section name duplicates another section name already declared in the same module, it is taken as a continuation of the same section. The assembly location counter is set to 0 at the beginning of a new section or to the value it had at the previous end of a continued section. The special character asterisk (*) may be specified in place of the section name to indicate the most recent section is to be continued.

All symbols defined within a module must be unique. Thus, symbols may be cross-referenced between sections of the same module.

```
    section     some_examples

    SECTION     examples

    SECTION     *
```

## AT

This directive is used to change the assembly location counter. It takes a single operand which is a numeric expression. The expression defines the offset in the current section at which the next instruction or data is to be generated. It may be used to move forward, leaving an uninitialized gap, or to move backward, overwriting code or data previously generated at that location.

The expression must use symbols which have already been defined or constants; no forward referencing of symbols is permitted.

In order to specify a symbolic location with a numeric expression, label the beginning of the section. If the label at the beginning of the section is, for instance, START.up, you could make the following assignments:

```
    AT      [$-START.up]+10        same as "DSB  10"

    AT      START.up+%100          resume assembling at offset %100
```

## TEMPLATE

This directive allows the definition of assembly-time symbols by means of suspending the actual generation of code/data. The effect of the TEMPLATE instruction is to cause all subsequent source statements to be processed normally but no code or data to be generated in the output object file. Thus all symbols are defined, but they are not assigned to any location. Normal processing of assembler instructions is reinstated by the next SECTION, MODULE, COMMON, or TEMPLATE statement.

The TEMPLATE directive takes one required operand. It is an expression which is absolute, internally relocatable or externally relocatable. The symbols subsequently defined are given values relative to that expression.

```
*  The following statements define the layout
*  of the REQUEST CONTROL BLOCK.  No memory is
*  reserved at this time but the four symbols
*  become defined as absolute constants which
*  are their respective offsets in the block.

                TEMPLATE    0

RCB.RQCODE      DSB         1
RCB.STATUS      DSB         1
RCB.DATAPTR     DSL         1
RCB.COUNT       DS          1
```

## COMMON

The COMMON directive is used to declare a common data area. Generation of code or data in the object module is suspended until the next MODULE, SECTION, TEMPLATE or COMMON directive. The instructions which follow have the effect of defining the symbols therein declared and of defining the length of the common area. The COMMON directive has no operand but a common name must be provided in the label field of the instruction. This follows the composition rules for external symbols and is itself an external symbol; the COMMON statement serves to declare it as such.

No memory space is reserved for the common area by the assembler. The name and size of the common is placed into the output object module for use by the linker. The common name is a bonafide external symbol and may be used in other places in the assembly where an external symbol is allowed.

```
*   Define named common area to contain all globally used variables.

GLOBAL_VARIABLES    COMMON
Buff.Ptr            DSL     1
Glob.Flag           DSB     1
CmdLength           DS      1          *** WARNING, rounding will
                    ,       ,              occur for alignment ***
```

## ASSIGN

ASSIGN is used to define an assembly-time symbol. The symbol to be defined appears in the label field of the instruction. The value to be assigned to it is given as the operand. The operand is an expression which may be absolute, internally relocatable or externally relocatable. The new symbol takes on the value and type of the expression. Symbols in the expression may not be forward referenced. The defined symbol must be unique within the module; it is not permissible to redefine a symbol with an ASSIGN statement.

```
CCCC           ASSIGN    %F         defines a constant symbol

KEY            ASSIGN    'A'        defines a character value

ABSOLUTE_ADDR  ASSIGN    <<3>>%100  defines an absolute address

LOOP2          ASSIGN    $          equivalent to "LOOP2 DSB 0"
               ,         ,          or to LOOP2 standing alone
               ,         ,          on a line

LOOP_X         ASSIGN    LOOP2+2    program location after first
               ,         ,          word of LOOP2 routine.
```

**GLOBAL**

The GLOBAL directive is used to define a global symbol. This symbol is accessible within the current module, and is also made accessible at link time to all other modules. There are no operands to the directive. The symbol to be defined is given in the label field of the instruction, and must be unique within the module. It receives the value of the current assembler location. This directive may only occur within a section; it may not appear within the range of a TEMPLATE or a COMMON directive.

```
compare global          label first instruction of routine
*                       so it may be used by all modules

*  Define a global word variable, initialized to
*   all ones.

        DS          0   align, to make sure
ONES    GLOBAL
        DD          %(2)1111111111111111
```

**EXTERNAL**

The EXTERNAL directive is used to declare a symbol which is to be defined at link time in another module. There are no operands. The symbol to be declared is given in the label field of the instruction. Since the symbol is not associated with any particular section, its declaration may appear anywhere in the module.

```
*   Declare routines in utility module needed by this module.

BCD_ADD      EXTERNAL
BCD_SUB      EXTERNAL
BCD_DIV      EXTERNAL
```

## IF and ENDIF

These directives are used to implement a conditional assembly facility. The IF instruction takes a single operand which is an expression which may be of any type, but may not contain forward symbol references. If the value of the expression is exactly zero, ALL statements following the IF and before the corresponding ENDIF are treated as comments. An ENDIF takes no operands. IF-ENDIF pairs may be nested.

Assume an assembly program is to be assembled in one of two different ways, depending on which machine, X or Y, it is going to run on. Using the ASSIGN directive we set the symbols X and Y to show which the current assembly is for. One is set to 1, the symbol for the machine being selected, the other to 0, for that not selected. A portion of the assembly might appear as follows:

```
    *   If assembling for the X machine,  invert the value.

            if   X               could also say IF X<>0
            COM  RO
            endif
```

## LISTON and LISTOFF

These directives allow the selective inclusion of portions of the assembly in the listing file. They take no operands. If no listing file was named in the assembler command line, then these have no effect since no listing is being produced anyway. Rather than being just an on/off switch listing control is accomplished with a signed counter. The counter starts at zero, each occurance of a LISTON increments it by one and each LISTOFF decrements it by one. Text is placed into the listing file whenever the counter is greater than or equal to zero. This technique provides hierarchical levels of control. The counter is not reinitialized for each new module encountered in the input source file.

## PAGE

This directive forces a page break in the listing file following the newline character of the previous line. A page heading along with the current title string is produced following a form-feed character. If no line has been printed since the last automatic or requested page break then the entire instruction is ignored. With no operand, PAGE forces a form feed. With an operand, the operand will set the number of lines per page. This does not include the 5 lines of header information. To get 50 lines per page, the PAGE operand would be 55.

### TITLE

This directive allows the programmer to provide a title to be placed in the upper left corner of each listing page. It takes a single operand which is a string enclosed in quotation marks ("). An automatic page break including a new heading is produced using the new title string.

```
    TITLE   "LINKER RELEASE 7.44 -- PASS ONE"
```

### INCLUDE

This directive causes the insertion of the source from another file into the current assembly at the point at which the directive occurs. There is a single operand consisting of the filename enclosed in quotation marks. The listing file always has the entire line containing the INCLUDE instruction before the insertion is done. If a page break occurs for any reason while in the included file the page heading shows the name of the file currently being processed. INCLUDEs may be nested, but they may not contain MODULE directives.

```
    include "stdio.h"          get standard i/o package definitions

    INCLUDE "Def_Insert"       place insertion source for Def here
```

## THE PCOS STANDARD

This section describes how to write Assembler source programs in order to obtain maximum compatibility with the operating system (PCOS) routines.
This will allow user programs to use the same procedures as for any PCOS utility for invoking and for passing parameters to the Assembler program.

The following figure shows the way in which an Assembler utility is connected to various parts of the system.

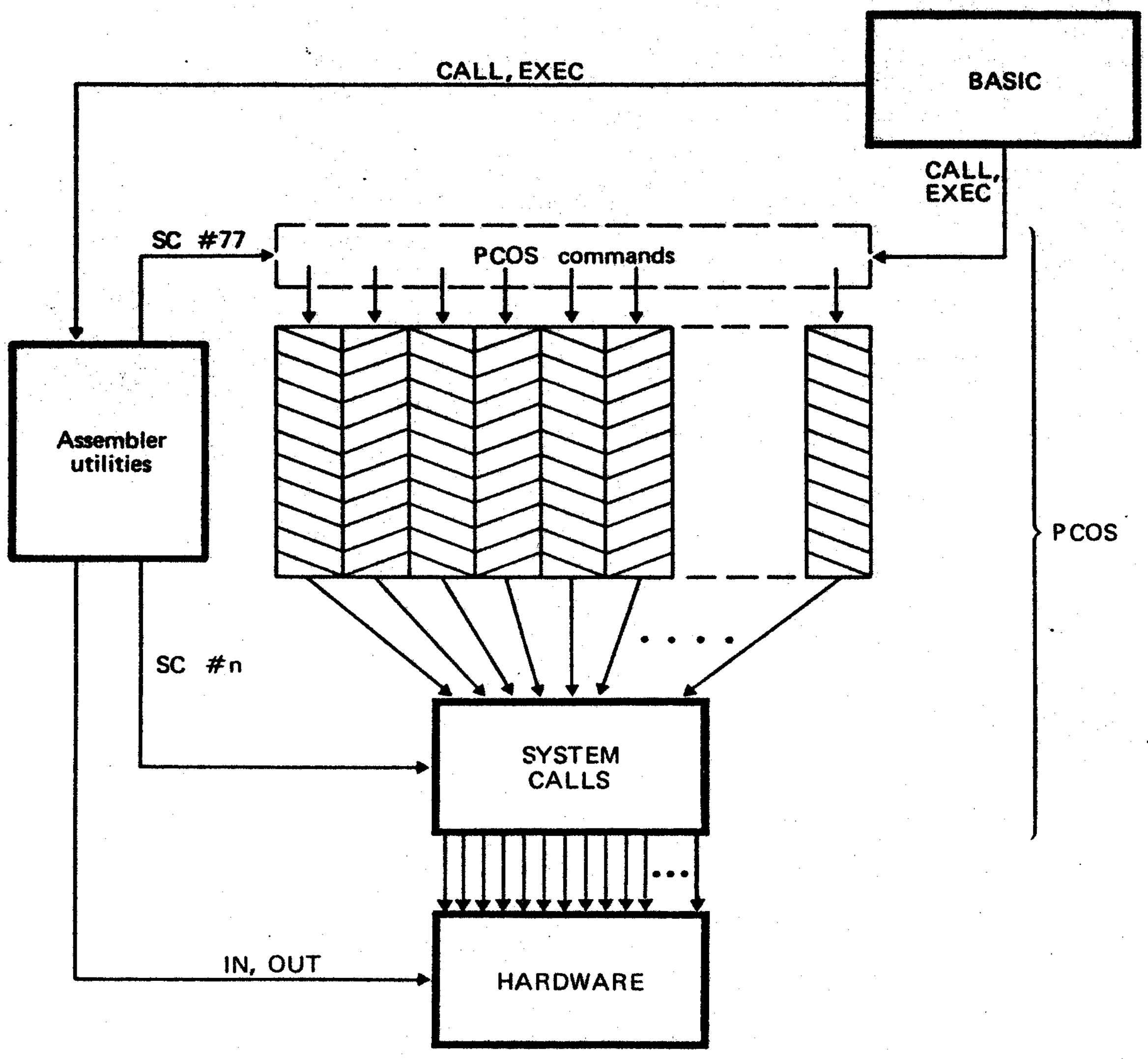


Fig. 2-1 Connection between Assembler utilities and other parts of the system

If Assembler routines are written following a certain standard, it is possible to invoke them like a simple PCOS command, or from a BASIC program.
By means of conventions on the passing of parameters, the same Assembler utilities can call PCOS commands or access a group of small routines (system calls), that are also used by the operating system (PCOS). These provide a certain number of elementary operations on the system hardware, thus facilitating programming.
Direct access to the system hardware will consequently be possible, by means of the Assembler instructions IN, OUT (see Appendix F for a list of I/O port assignments and consult M20 hardware literature).
It is also possible to access PCOS commands from an Assembler utility, using the Assembler instruction SC 77 which is described in the second part of this manual.
Let us now summarise the various ways to call (from PCOS and BASIC respectively) an Assembler utility (e.g. MYFILE) which is written according to the PCOS standard, to which the parameters para1, para2 and para3 are passed.

PCOS

```
MYFILE    PARA1,PARA2,PARA3
```

BASIC

```
CALL "MYFILE"(PARA1,PARA2,PARA3)
```

Where PARA1,PARA2,PARA3 can be either constant or variable parameters.

or

```
EXEC "MYFILE PARA1,PARA2,PARA3"
```

Where PARA1,PARA2,PARA3 can only be constant parameters.

Furthermore, certain conventions within our Assembler source file, will also make it possible to obtain the identification of our program, while the program is being loaded (by using the PCOS commands PLOAD or PDEBUG).

The instructions and the Assembler directives to be used in order to obtain a routine compatible with the PCOS standard, are dealt with in this order:

1. Configuration code

2. Header

3. How to pass the parameters

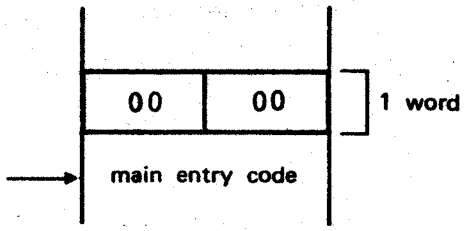
4. Exit Routine

5. Example

## 1. Configuration Code

The first "word" of an executable program, will provide information (while the program itself is being loaded) on how it will be configured in memory. This word, being the first word of the program, must assume the value zero and indicates that the word immediately following, is the entry point.

Schematically:



To obtain a source program complying with configuration code 0, the first statement must be DD 0.
Other types of configuration codes are allowed by the system software, but cannot be utilised by the user.

## 2. Header

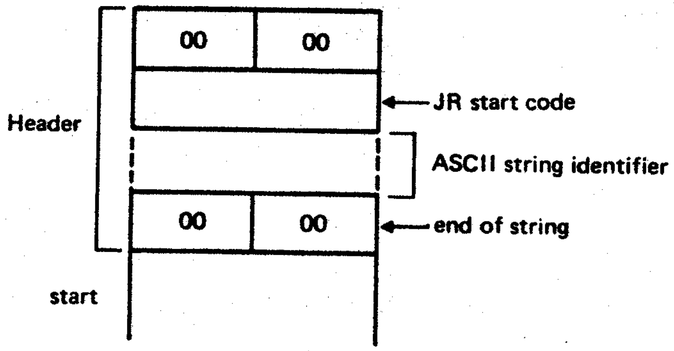
When an executable file is being loaded using PLOAD or PDEBUG, the M20 displays some information on the screen, amongst which the program name. This program name can be inserted at source program level in the "header" of the program itself.

The header is that part of the program containing both the configuration code previously mentioned and a string identifier which will be the program name. For example, the "header" of a source file can contain any of the following Assembler instructions:

```
                 module         echo, segmented
                 section        example
  Header       [ dd             0             type 0
    -->        | JR             start
               | ddb            "File Echo "   string ident. prog.
               [ dd             0
  start          .
                 .
                 .
               program
                 .
                 .
```

In practice, the string identifier is placed in memory between the second word and the first occurrence of a "null (00)". This string must be skipped by means of the instruction "JR start" in the source program. It is important that the jump instruction of the string identifier is JR and not JP, as JR only occupies 1 word, thus allowing the start of the string from the third word of the executable program.

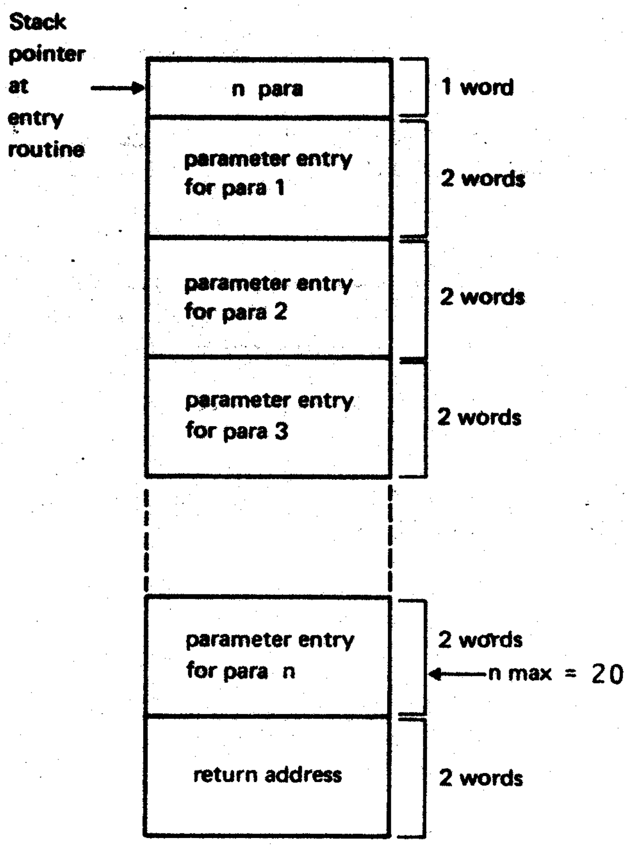
The situation of the program in memory will be the following:



## 3. How to pass the parameters

When an assembler utility is invoked by PCOS or by BASIC, all the parameters passed to it are placed (pushed) in the stack by the system so that they can be extracted (popped) from the stack in the order and in the way in which they were placed.

The maximum number of parameters that can be passed is 20. The pointers to the parameters (parameter entry) will be allocated in the stack when the routine is invoked, in the following way:

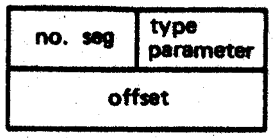

Stack pointer at entry routine
n para
1 word
parameter entry for para 1
2 words
parameter entry for para 2
2 words
parameter entry for para 3
2 words
parameter entry for para n
2 words
n max = 20
return address
2 words

The user program must however extract information about the various parameters by means of as many "pop" instructions from the stack, as the corresponding number of parameters.
As seen in the figure, the number of parameters is given by the first word addressed by the stack pointer when the routine is invoked by PCOS or BASIC.

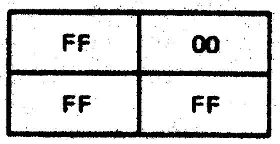
It is possible to have 3 types of parameters:

| | |
|---|---|
| - Null | with hexadecimal code 00 |
| - Integer | " " " 02 |
| - String | " " " 03 |

The code for each type of these parameters is memorized in the 2nd byte of the 1st word for each "parameter entry"



For the type "null" the "parameter entry" does not contain an actual pointer, but for compatibility, it will be of the type:

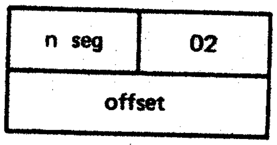


This type of "parameter entry" is created when, for example, the routine is invoked in the following way:

```
                my para1,,para3
```

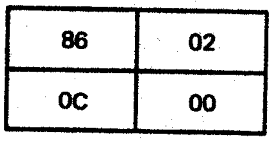
It can be seen therfore, that the second parameter has been jumped (para 2). This means in practise, that a pointer to a dummy parameter (parameter entry) is created (with FF00 FFFF) in order to maintain compatibility with the standard.

For the integer type (02) there will be a real pointer to the parameter, constructed in the following way:



The segment number and the offset constitute the effective address to a word integer (this is a Z-8001 compatible segmented address)

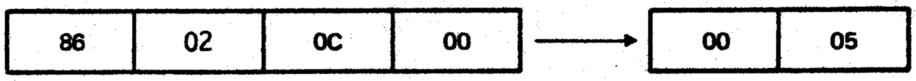
For example, the "parameter entry" for integer 5 could be:



In this case, the address for the word containing integer 5 will be:

≪6≫ 0C00

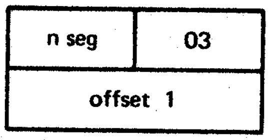
This can be represented schematically as:

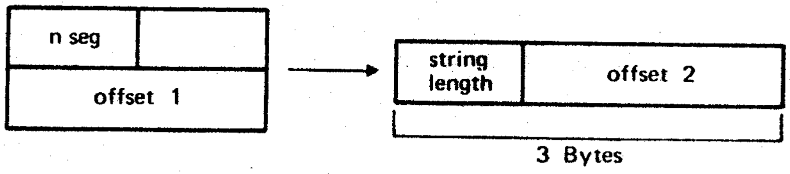


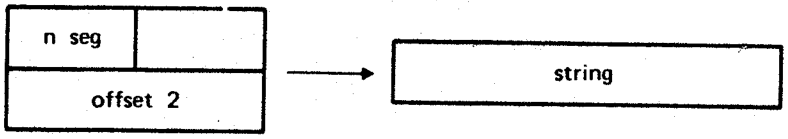
(Note that once the type has been identified the second byte is ignored)

However, if the parameter is a "string" (type 03), the procedure for pointing to the string is more complex than the previous two.
In this case, the pointer (entry parameter) points to a set of 3 bytes, the first of which contains the string length, whereas the other two contain the address (significant only for the offset part as the seg. no. is the same as the entry parameter) pointing to the string itself.

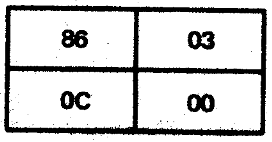
e.g.

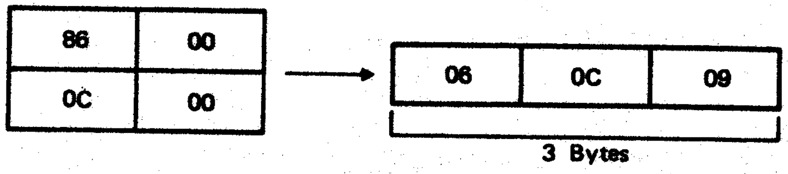


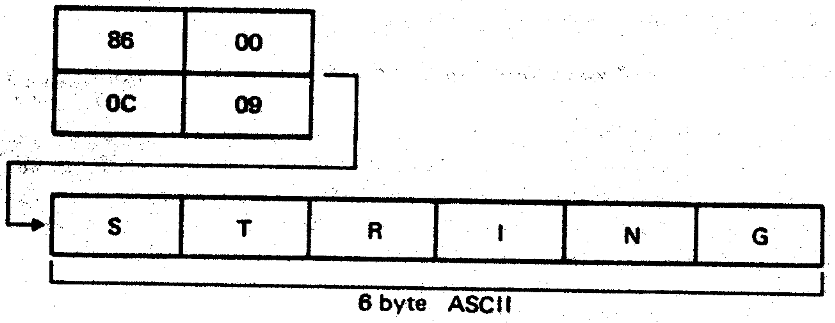




For example, the "Parameter Entry" for the string "STRING" must have the following structure:







## 4. Exit Routine

The Assembler programmer is advised to write his programs so that he can easily handle the exit from the program by means of the instruction RET, in order to return to the environment from which it was called.

It is convenient to save in 2 words (RETADR) the stack address which points to the program return address. In this way, the stack pointer can be set to this address before exiting the program (using the "Ret" instruction). In order to access the program return address, you will have to use the "number of parameters" saved in the first stack location.

To accomplish this, the following Assembler instructions can be used at the start of the utility:

```
                 .
                 .
                 .
                 POP R0,@RR14  no. par in R0
                 CLR R2
                 LD  R3,R0
                 SLL R3,#2        no. par x4
                 ADD RR2,RR14    pointer to reurn address in RR2
                 LDL RETADR,RR2  store RR2
                 .
                 .
                 .
               program
                 .
                 .
                 LDL RR14,RETADR
                 RET
          RETADR DSL 1
                 .
                 .
                 .
                 .
```

## 5. Example

Here a complete example is given of a simple Assembler source program in which the standard (which we have seen up till now) is taken into account. In input, this program takes a string as a parameter and echoes the string itself in output. Once the program has been linked and assembled in an executable file echo.cmd, it can be called from PCOS in the following way:

```
ec string /CR/
```

```
*****************************************************************************
*                                                                           *
*   Echo string input to this routine.                                      *
*   An example of the use of the M20 Assembler Package.                     *
*                                                                           *
*                                                                           *
*****************************************************************************
*
        MODULE      echo,SEGMENTED
        SECTION     example
        TITLE       "ROUTINE SEGMENTED ECHO"
*
*   program header
*
        DD          0                   configuration code--MANDATORY HERE
        JR          echo                PCOS expects this instruction format
str     DDB         "File Echo "        program identifier
        DDB         "%r"                carriage return
        DDB         0                   end of program header
*
*   code
*
echo    ASSIGN      %
        LDA         RR12,str            point to message
        SC          #89                 display string identifier
        POP         R0,@RR14            get parameter count
        CLR         R2                  insure no errors in stack computation
        LD          R3,R0               use R3 as working register
        SLL         R3,#2               multiply # parameters by 4
        ADDL        rr2,rr14            add to stack to point to return addr
        LDL         retadr,RR2          save it for later return
*
*   Now test for # parameters passed and reject if wrong
*
        TEST        R0                  how many parameters?
        JP          NZ,echo1            not zero parameters so go on
        LD          erconu,#90          Message = "Error in parameter"
        JP          error               exit with error message
*
*   So we have one or more parameters passed.
*   Transfer parameters to registers, checking data types...
*
echo1   ASSIGN      %
        POPL        RR2,@RR14           get pointer to parameter in rr2
        CPB         RL2,#3              is parameter a string?  (type 3)
        JP          EQ,echo2            yes, go service, else....
        LD          erconu,#13          Message = "Bad data type"
        JP          error
*
*   Main program code here
*
echo2   ASSIGN      %                   print input string to screen
        CLRB        RL2                 clear data type byte
        CLR         R7
        CLRB        RH6
        LDB         RL6,@rr2            parameter lenght in RL6
        INC         R3                  rr2 points to the next byte
        LDB         RH1,@RR2
        INC         R3                  rr2 points to the next byte
        LDB         RL1,@RR2
```

```
                LD          R12,R2
                LD          R13,R1              RR12 points to the st ing parameter
                CLR         R8                  prepare rr8 for adding two bytes to
                CLR         R9                  end of string
                LDL         RR8,RR12            setup rr8 to point to string
                ADD         R9,R6               rr8 points to the end of string
                LDB         @RR8,#13            add a carriage return
                INC         R9
                LDB         @RR8,#0             add a null for SC #89
                SC          #89                 echo the string
                CLR         R5                  assume no error returned by SC #89
                JR          n_return            jump around error service
*
*   Exit with appropriate error message
*
error       ASSIGN      *
            LD          R5,erconu               must have been setup first !!!
            SC          #88                     display error message
*
*   Normal return
*
n_return    ASSIGN      *
            LDL         RR14,retadr             point stack pointer to return address
            RET         .                       and return to caller
*
*   Storage area
*
            SECTION     area
*
retadr      DSL         1                       storage for return address
erconu      DS          1                       storage for error type code
*
*   End (echo)
*
```

# 3. THE ASSEMBLER (ASM) COMMAND

# THE ASSEMBLER (ASM) COMMAND

## ABOUT THIS CHAPTER

This chapter details the ASSEMBLER (ASM) command. This command processes an Assembly language source file and produces an object file.

## CONTENTS

## THE ASSEMBLER (ASM) COMMAND

### ASM

The ASM command processes an Assembler language source file of ASCII text and produces a file containing the corresponding Z8000 machine code. This file is known as an object file. Optionally the ASM command produces a listing file. When such a file is listed the video displays the source file program lines on the right and the generated codes or symbol values along with other information about each program line on the left. If the XREF option is specified for a listing file then it will also include a cross-reference table at the end. An example of a listing file is shown at the end of this chapter.

The ASM command is called from PCOS like any other PCOS command. When called it is loaded into memory and executed. After execution the system returns to the PCOS environment. The command syntax is shown in figure 3.1 below.

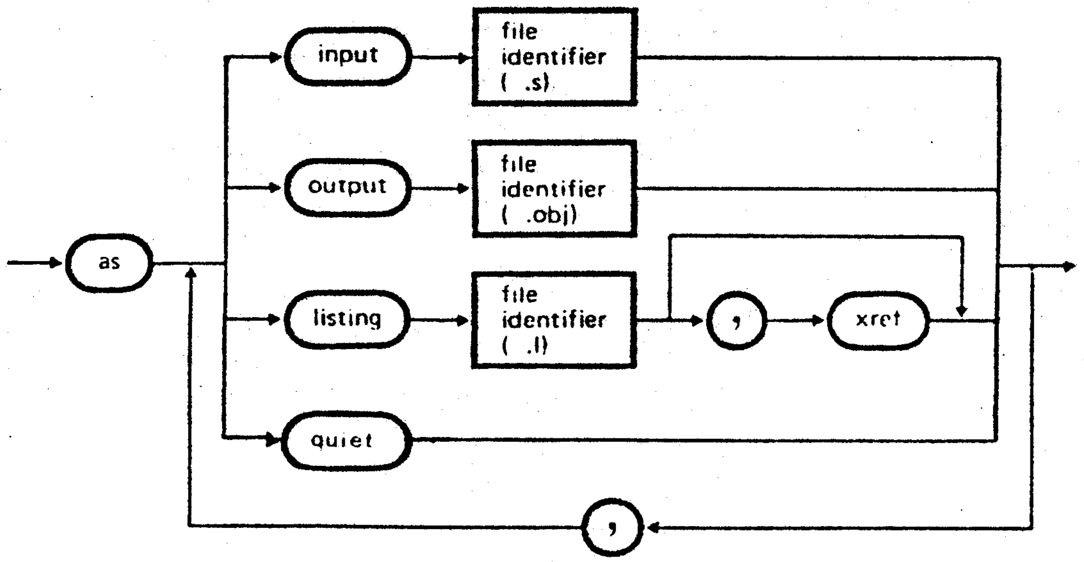


Fig. 3-1 The ASM command

**Where**

| SYNTAX ELEMENT | MEANING |
|---|---|
| input | The keyword which must precede the source file identifier |
| file identifier (.s) | The source file identifier.<br>Usually a source file name is assigned a '.s' extension. |
| output | The keyword which must precede the object file identifier |
| file identifier (.obj) | The object file identifier. Here again it is good programming practice to assign the extension .obj to an object file name. If the file specified does not exist then it will be created; if on the other hand the file exists then it will be overwritten with the new object file. |
| listing | The keyword which must precede the listing file identifier. |
| file identifier (.l) | The listing file identifier. A listing file name is usually assigned the extension '.l'. If the file specified does not exist then it will be created; if on the other hand the file exists then it will be overwritten with the new listing file. |
| xref | The Cross-Reference keyword. If specified then a cross-reference table is included at the end of the listing file. This table contains an entry for each symbol defined in the assembly with the following information:<br>- The statement number at which the symbol is defined.<br>- Its value and type.<br>- An ordered list of statement numbers which reference the symbol. |
| quiet | The QUIET keyword. Specifying this keyword in an ASM command line will supress all the messages normally output on the video except for error messages which abort the command. |

An ASM command parameter is identified by the command line interpreter by its keyword; for this reason parameters can be entered in any order.

The command "AS" by itself causes the command parameters to be displayed on the screen.

If the OUTPUT and LISTING options are omitted then the respective object and listing files will not be created.

## Characteristics

The ASM command is executed in a number of stages depending on the number of modules in the input source file. In the first stage the header is assembled; each module is then assembled in subsequent stages. Each assembly stage is done in two passes.

During execution, unless the QUIET keyword is specified, the video displays information indicating the end of each pass, and short messages for each error discovered. Error messages specify the line number where the error was detected and the error type. When execution is complete the video displays a summary line with the total number of errors detected. A listing file printout will turn out to be very useful for subsequent analysis of errors. A list of ASM error codes with their meaning is given in appendix B.

## Examples

| IF you enter | THEN ... |
|---|---|
| as input 1:test.s,output 1:test.obj /CR/ | the source file "test.s" which is on the disk inserted in drive 1 is assembled. The resulting object file is written into a file called "test.obj" on the disk inserted in drive 1. If this file already exists then it will be overwritten with the new object file, if on the other hand it does not exist then it will be created. |
| as input 1:myfile.s,output 1:myfile.obj,listing 1:myfile.l,xref /CR/ | the source file "myfile.s" which is on the disk inserted in drive 1 is assembled, as in the previous example, however this time a listing file is also created. The listing file "myfile.l" is created on the disk inserted in drive 1. The file "myfile.l" will have a cross-reference table included. |

The sample printout on the following pages is that of the listing file corresponding to the source file shown in chapter 2. This listing file includes a cross-reference table. This file was obtained using the following command:

```
        as input 1:echo.s,output 1:echo.obj,listing 1:echo.l,xref /CR/
```

```
                                           M20 Assembler v 2. 1                    02/10/1983 15:02:09  Page  1

                                Source  Ref  File  1:echo.s
Location Code/Value               Line  Line  Line  Source Text
-------------------------------------------------------------------------------------------------------------
                                     1           1  ***********************************************************
                                     2           2  *
                                     3           3  *   Echo string input to this routine.
                                     4           4  *   An example of the use of the M20 Assembler Package.
                                     5           5  *
                                     6           6  *
                                     7           7  ***********************************************************
                                     8           8  *

8 Lines, 0 Warnings, 0 Errors in Header

                                     9           9           MODULE     echo,SEGMENTED
                                    10          10           SECTION    example
                                    11          11           TITLE      "ROUTINE SEGMENTED ECHO"
                                    12          12  *
                                    13          13  *   program header
                                    14          14  *
00,0000 0000                        15          15           DD         0                  configuration code--MANDATORY HERE
00,0002 E806                        16    23    16           JR         echo               PCOS expects this instruction format
00,0004 46 69 6C 65 20 45 63        17          17  str      DDB        "File Echo " .     program identifier
00,0008 68 6F 20
00,000E 0D                          18          18           DDB        "%r"               carriage return
00,000F 00                          19          19           DDB        0                  end of program header
                                    20          20  *
                                    21          21  *   code
                                    22          22  *
        00,0010                     23          23  echo     ASSIGN     %
00,0010 760C 00,0004                24    17    24           LDA        RR12,str           point to message
00,0016 7F59                        25          25           SC         #89                display string identifier
00,0018 97E0                        26          26           POP        R0,@RR14           get parameter count
00,001A 8028                        27          27           CLR        R2                 insure no errors in stack computatio
00,001C A103                        28          28           LD         R3,R0              use R3 as working register
00,001E B331_0002                   29          29           SLL        R3,#2              multiply # parameters by 4
00,0022 96E2                        30          30           ADDL       rr2,rr14           add to stack to point to return add
00,0024 5D02 01,0000                31    90    31           LDL        retadr,RR2         save it for later return
                                    32          32  *
                                    33          33  *   Now test for # parameters passed and reject if wrong
                                    34          34  *
00,002A 8D04                        35          35           TEST       R0                 how many parameters?
00,002C 5E0E 00,0040                36    43    36           JP         NZ,echo1           not zero parameters so go on
00,0032 4D05 01,0004 005A           37    91    37           LD         erconu,#90         Message = "Error in parameter"
00,003A 5E08 00,0086                38    76    38           JP         error              exit with error message
                                    39          39  *
                                    40          40  *   So we have one or more parameters passed.
                                    41          41  *   Transfer parameters to registers, checking data types
                                    42          42  *
        00,0040                     43          43  echo1    ASSIGN     %
00,0040 95E2                        44          44           POPL       RR2,@RR14          get pointer to parameter in rr2
00,0042 0A0A 0303                   45          45           CPB        RL2,#3             is parameter a string?  (type 3)
00,0046 5E06 00,005A                46    52    46           JP         EQ,echo2           yes, go service, else...
00,004C 4D05 01,0004 000D           47    91    47           LD         erconu,#13         Message = "Bad data type"
00,0054 5E08 00,0086                48    76    48           JP         error
                                    49          49  *
                                    50          50  *   Main program code here
```

```
ROUTINE SEGMENTED ECHO                     M20 Assembler v 2. 1                      02/10/1985 15:02:32 Page  2

                               Source  Ref  File  1:echo.s
Location Code/Value             Line   Line Line  Source Text
------------------------------------------------------------------------------------------------------------------------
                                 51          51   *
          00,005A                52          52   echo2      ASSIGN     *            print input string to screen
00,005A 8CA8                     53          53              CLRB       RL2          clear data type byte
00,005C 8078                     54          54              CLR        R7
00,005E 8C68                     55          55              CLRB       RH6
00,0060 202E                     56          56              LDB        RL6,@rr2     parameter lenght in RL6
00,0062 A930                     57          57              INC        R3           rr2 points to the next byte
00,0064 2021                     58          58              LDB        RH1,@RR2
00,0066 A930                     59          59              INC        R3           rr2 points to the next byte
00,0068 2029                     60          60              LDB        RL1,@RR2
00,006A A12C                     61          61              LD         R12,R2
00,006C A11D                     62          62              LD         R13,R1       RR12 points to the string parameter
00,006E 8088                     63          63              CLR        R8           prepare rr8 for adding two bytes to
00,0070 8098                     64          64              CLR        R9           end of string
00,0072 94C8                     65          65              LDL        RR8,RR12     setup rr8 to point to string
00,0074 8169                     66          66              ADD        R9,R6        rr8 points to the end of string
00,0076 0C85 0000                67          67              LDB        @RR8,#13     add a carriage return
00,007A A990                     68          68              INC        R9
00,007C 0C85 0000                69          69              LDB        @RR8,#0      add a null for SC #89
00,0080 7F59                     70          70              SC         #89          echo the string
00,0082 8058                     71          71              CLR        R5           assume no error returned by SC #89
00,0084 E804                     72     82   72              JR         n_return     jump around error service
                                 73          73   *
                                 74          74   *   Exit with appropriate error message
                                 75          75   *
          00,0086                76          76   error      ASSIGN     *
00,0086 6105 01,0004             77     91   77              LD         R5,erconu    must have been setup first !!!
00,008C 7F58                     78          78              SC         #88          display error message
                                 79          79   *
                                 80          80   *   Normal return
                                 81          81   *
          00,008E                82          82   n_return   ASSIGN     *
00,008E 540E 01,0000             83     90   83              LDL        RR14,retadr  point stack pointer to return address
00,0094 9E08                     84          84              RET        ,            and return to caller
                                 85          85   *
                                 86          86   *   Storage area
                                 87          87   *
                                 88          88              SECTION    area
                                 89          89   *
01,0000                          90          90   retadr     DSL        1            storage for return address
01,0004                          91          91   erconu     DS         1            storage for error type code
                                 92          92   *
                                 93          93   *   End (echo)
                                 94          94   *

96 Lines, 0 Warnings, 0 Errors in Module echo
```

```
Index for Module echo                          M20 Assembler v 2. 1                          02/10/1983 15:02:41  Page

                                                               Source Lines
Type and (Base or Section)          Name           Value     Defining  Uses
-------------------------------------------------------------------------------------------------------------------
Section                             area           01,0006     88#
Module                              echo         0000_009C      9#
Relocatable (example)               echo           00,0010     23#      16
Relocatable (example)               echo1          00,0040     43#      36
Relocatable (example)               echo2          00,005A     52#      46
Relocatable (area)                  erconu         01,0004     91#      37     47     77
Relocatable (example)               error          00,0086     76#      38     48
Section                             example        00,0096     10#
Relocatable (example)               n_return       00,008E     82#      72
Relocatable (area)                  retadr         01,0000     90#      31     83
Relocatable (example)               str            00,0004     17#      24
```

```
ROUTINE SEGMENTED ECHO                         M20 Assembler v 2. 1. 7. 0                    02/10/1983 15:02:49  Page

                                  Source  Ref  File  1:echo.s
Location Code/Value                Line   Line  Line  Source Text
-------------------------------------------------------------------------------------------------------------------
94 Lines, 0 Warnings, 0 Errors in 1:echo.s
```

# 4. THE LINK COMMAND

# THE LINK COMMAND

## ABOUT THIS CHAPTER

This chapter describes the LINK command and all its keyword parameters. The chapter ends with an example and sample printouts of a command file and a map file.

## CONTENTS

## LINK

LINK is a linkage editor and locater which converts z-type object modules into a PCOS 2.0 relocatable load file. The LINK command must be called from the PCOS environment like any other PCOS command. LINK inputs a group of Olivetti Z-type object files, and outputs a single load file. The LINK command allows a number of optional features described below.

## PARAMETERS

There are seven types of parameters which can be passed to LINK. Six of these are of the Keyword type, and can have parameters of their own. The seventh is a Block Descriptor; this parameter determines the order in which program sections will be loaded in memory.

The seven types of parameters are the following:

- Multifile keywords
- File keywords
- Value keywords
- Entry keyword
- Message keyword
- Simple keywords
- Block descriptor

The command syntax is shown in figure 4.1 below.

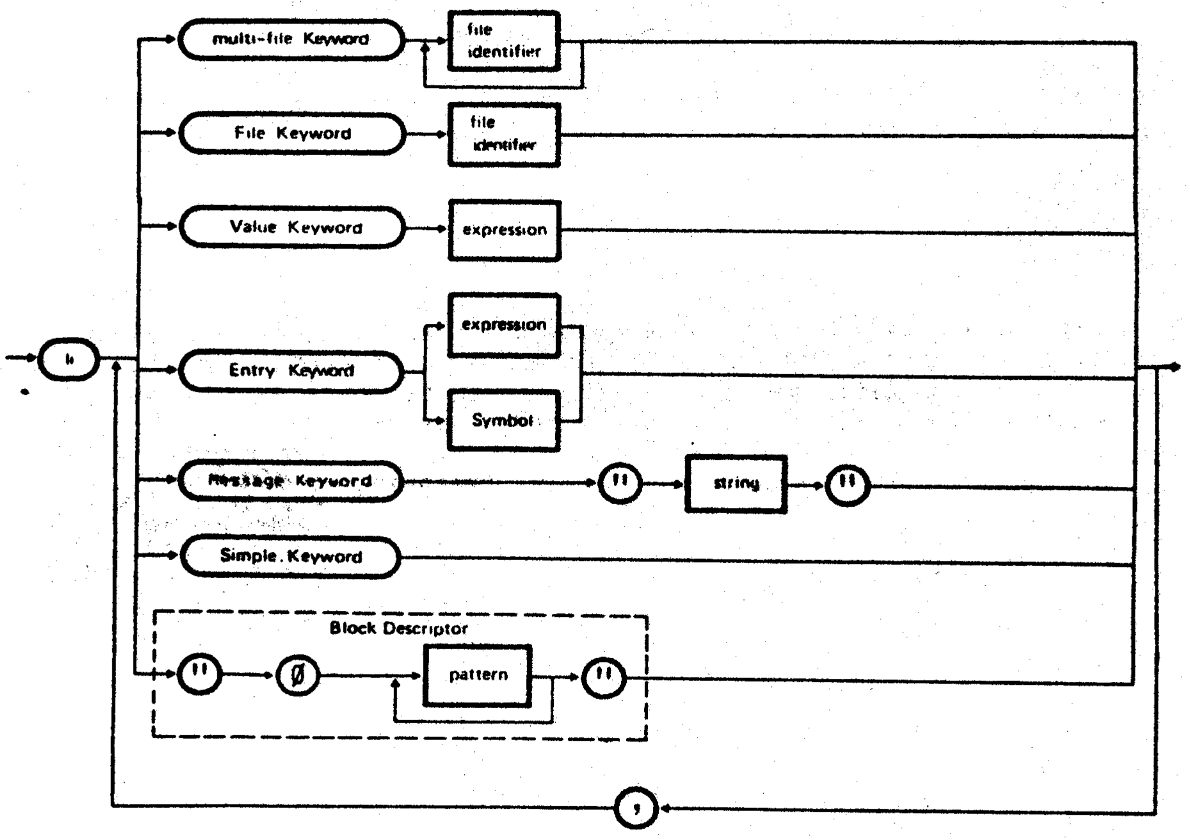


Fig. 4-1 The LINK command

**Where**

| SYNTAX ELEMENT | MEANING |
|---|---|
| file identifier | The name of a file complete with any necessary volume identifier, and/or file password. Depending on the keyword in question the file will be accessed or created. In the latter case if the file specified already exists it will be overwritten with the new output. |

| | |
|---|---|
| expression | Any expression denoting the desired value. This can be either a simple term or a number of terms separated by operators (See section on Terms Expressions and Operators below). |
| symbol | Any symbol that exists in the input modules. |
| string | Any string of ASCII characters. |
| pattern | A string of characters, which may also include wild cards, denoting one or more section names. An asterisk (*) matches any string of characters, a question mark (?) matches any character, [ab...] which matches any one character inside the brackets, and [a-b] which matches matches any one character in the interval a-b. |

**Note:**

**Care must be taken that no more than 20 parameters are specified in one LINK command; this is the maximum number of parameters that the PCOS command line interpreter can handle. In cases where more than 20 parameters need to be specified the COMMAND keyword can solve the problem (the COMMAND keyword is described below in the section on File Keywords).**

## TERMS EXPRESSIONS and OPERATORS

All values are 32-bit unsigned whole numbers. There may be no spaces or punctuation characters between the terms and operators of an expression. A number will be interpreted as a hexadecimal number only if it is preceded by the "%" sign, otherwise it is interpreted as a decimal number.

The '+', '-', '|' and '&' operators simply perform addition, subtraction, bitwise-or and bitwise-and, respectively, on their operands. The '≫' operator performs the operation of the left operand shifted left 24 bits plus the right operand. This facilitates the formation of segmented addresses when coupled with the ignored '≪' token which can precede any term, thus: %0e00143c, %0e≫%143c, and ≪%0e≫%14c3 all evaluate to the same address.

The only permissible case of a null term is immediately following the '≫' operator at the end of an expression. In this case zero is used as the right operand for the operator thus allowing ≪33≫ to stand for ≪33≫0.

## COMMENTS

Comments, enclosed in exclamation marks, can be inserted in a LINK command line between parameters. A map file (which can be created using the map keyword described below in the section on File Keywords) contains a copy of the LINK command being executed, and any comments made on the command line can help render future reference clearer.

## MINIMUM COMMAND ELEMENTS

The required elements of a LINK command which outputs a PCOS 2.0 executable file are the following:

- The multifile keyword INPUT followed by the file identifier(s) if the input file(s).
- The file keyword OUTPUT followed by the file identifier of the output file.
- The BLOCK DESCRIPTOR "0 *"

Commonly used options are:

- The multifile keyword LIBRARIES followed by the file identifier(s) of a library file(s).
- The file keyword MAP followed by the file identifier of a map file.
- The ENTRY keyword followed by the the program entry point.
- The file keyword COMMAND followed by the file identifier of a file containing part of a LINK command line.

These and other keywords are described in more detail in the next section.

## THE KEYWORDS

In the following section all the LINK keywords are described. Each description has the keyword as a heading. In the command line keywords must be entered as they appear in this heading in either capital or small letters.

### MULTI-FILE KEYWORDS

#### INPUT

The INPUT keyword may occur any number of times. It specifies files containing Z-type object modules which contain all code sections to be located.

#### LIBRARY

This keyword instructs the program to select from the named library files the modules which have been referenced in the input file.

A library file can be created using the MLIB command described in chapter 6.

### FILE KEYWORDS

#### COMMAND

The COMMAND keyword can be used in the command line to insert parameters from another named file (Command file). Only one level is allowed (i.e. you cannot insert a COMMAND keyword in your specified file).

Such files containing part of a command line can be created using the Video File Editor. An example of a Command file is shown at the end of this chapter.

#### OUTPUT

The OUTPUT keyword occurs once and only once. It specifies a file to receive the executable binary load file. The file is created if it does not exist or is completely replaced with the new output if it does exist.

The load file can be assigned any legal name, however there are two filename extensions which have a special meaning to PCOS; these are ".cmd" and ".sav". These filename extensions allow files to be called and executed from the PCOS environment like any other PCOS command (i.e. by entering the first two characters of the file name). If a file has niether of these extensions it can be invoked by entering the complete file identifier. When a file which has no ".sav" extension is called it will be loaded from disk to the M20's memory, and executed. After execution the memory space that was occupied by the program is again made available to the system. This means that if the program is to be executed a second time it will have to be reloaded from disk to memory. In the case of a ".sav" extension the file will be permanently loaded and executed. In this case the file can be executed again even if the disk the file was loaded from is removed from its disk drive.

### MAP

The MAP keyword may occur once. It specifies the file to receive the formatted map. It is created if it does not exist or is completely replaced with the new map if it does exist. If no MAP keyword is given, no map file is produced.

A map file will contain a copy of the LINK command line being executed, diagnostic messages, a location ordered map of sections and an alphabetical list of section names and global symbols with their corresponding locations.

## VALUE KEYWORDS

### ATTRIBUTE

The parameter passed to the ATTRIBUTE keyword is placed in an "attribute" byte of the header part of the output load file.

FOR ROUTINES TO RUN ON RELEASE 2.0 OF PCOS IT IS NECESSARY TO SET THIS BYTE TO ONE. IF NOT SPECIFIED, LINK WILL SET THIS BYTE TO ONE BY DEFAULT, IT IS THEREFORE NOT NECESSARY TO SPECIFY THIS KEYWORD AT ALL.

### TYPE

This keyword sets an "attribute" byte in the header part of the output load file.

FOR ROUTINES TO RUN ON RELEASE 2.0 OF PCOS THIS BYTE MUST BE SET TO ZERO, AND AS LINK SETS THIS BYTE TO ZERO BY DEFAULT IT IS NOT NECESSARY TO SPECIFY THE TYPE KEYWORD AT ALL.

## ENTRY KEYWORD

### ENTRY

The ENTRY keyword may occur once. It provides either a numeric value or a global symbol name which is to be made the entry point of the executable program. The entry point is determined as follows:

- If an ENTRY keyword is given, then the entry point specified is used, regardless of any definition within the input module itself.

- If no ENTRY keyword is given, then the entry point is set as defined in the input module.

## MESSAGE KEYWORD

### MESSAGE

A MESSAGE keyword supplies the ASCII text (which must be one string) to go in the message record of the load file. There may be any number of MESSAGE keywords in one LINK command. The message record may be used for comments, remarks, date and time of operation, etc., and does not form part of the executable program itself.

## SIMPLE KEYWORDS

### NOWARNINGS

By default, various warning messages are included in the diagnostic messages produced by the locater. If the NOWARNINGS keyword is given, then all warning messages are suppressed.

### QUIET

The QUIET keyword causes output normally sent to the standard output to be suppressed, except for fatal error messages. If no QUIET keyword is given, interesting information is sent to the standard output. This normally consists of the echoed command line and all warning messages (if the NOWARNINGS keyword has not been given).

### STATISTICS

The STATISTICS keyword, if specified, causes the program to output statistics on how much of LINK 's memory was used.

### SQUEEZE

The SQUEEZE keyword instructs LINK to use minimum-sized buffers. The effect of this keyword is to allow the maximum possible memory space for the link-time data structure and symbol table; the tradeoff is decreased speed.

### OPTIMIZE

Specifying the OPTIMIZE keyword in the command line causes the output file to be optimized by not including uninitialized memory at the end of the program text in the program text section of the output load file.

## BLOCK DESCRIPTOR

The block descriptor specifies the order in which code sections, by "section name" (as defined in the input modules), are to be loaded in memory by the loader.

The "patterns" are section names which correspond to the same names in the input object modules, but may include the pattern-matching characters "*" which matches any string, "?" which matches any single character, "[ab...]" which matches any one character inside the brackets, and "[a-b]" which matches any one character in the interval a-b. A pattern stands for all section names which match that pattern. The names are taken in the order that they occur in the input object modules (i.e. lower section numbers first).

This feature can be used, for example, to separate program sections from data: If all program sections are assigned the extension ".prg" and all data sections the extension ".data", then specifying the block descriptor

```
"0 *.prg *.data"
```

will cause the program sections to be collected first followed by the data sections.

If the ordering of the program sections is of no importance then it is sufficient to specify the following block descriptor:

```
"0 *"
```

## KEYWORD ORDER

The order in which keywords appear has no gross effect on the outcome of the operation. The effects of ordering are due to the fact that files are opened and flags are set when their respective keywords are encountered. For example, keywords which appear before the MAP keyword do not get echoed into the MAP file, and keywords before a QUIET are echoed to standard output unless the QUIET keyword appears on the command line before any other keyword. The SQUEEZE keyword is not totally effective unless it occurs before the MAP keyword, since the map I/O buffer is allocated when the MAP command is encountered.

## ERRORS

If any fatal error occurs during the parsing of keywords or the execution of the locate operation, the program is stopped immediately with an error message on standard output and, if it was specified, the map file.

## Examples

The following LINK command will create an executable file "echo.cmd" from the object file created in the example shown in chapter 3, "echo.obj". The command will also create a map file "echo.map".

```
li map 1:echo.map, input 1:echo.obj,output 1:echo.cmd,"0 *" /CR/
```

The same result can also be obtained using the command file shown below in the following command:

```
li command 1:comlist /CR/
```

The following is a listing of the file "comlist":

```
                    ! Command file for LINKing the ECHO example !


                              MAP 1:echo.map

  ! create a map file "echo.map" on the disk inserted in drive 1. Note that as !
  ! this is the first keyword  in the file all that follows will appear in the !
  ! map file.                                                                  !



                              INPUT 1:echo.obj

  ! if more than one  file  need  to  be  specified  these  can follow even on !
  ! successive lines as long as there are no intervening keywords.             !



                              OUTPUT 1:echo.cmd

  !                     Only one output file is allowed                        !


                                   0 *

  ! The block descriptor here is not  enclosed  in  quotes.  Quotes  are  not !
  ! allowed in a command file.                                                !
```

On the following page is a listing of the map file created by this command.

```
Olivetti LINK -- Release s1.3

Commands:

     Map     1:echo.map

     ! create a map file "echo.map" on the disk inserted in drive 1. Note that as !
     ! this is the first keyword in the file all that follows will appear in the !
     ! map file.                                                                  !


                              INPUT 1:echo.obj

     ! if more than one  file  need    to  be  specified  these  can follow even on !
     ! successive lines as long as there are no intervening keywords.        !



                              OUTPUT 1:echo.cmd

     !                    Only one output file is allowed                    !


                                   O *

     ! The block descriptor here is not  enclosed  in  quotes.   Quotes  are  not !
     ! allowed in a command file.                                                 !
======================================================================

Warnings:

(Pass One - 1:echo.obj)
(Pass Two - 1:echo.obj)
(Generating Maps)

======================================================================

Input Map:


File    Module  Section          Size (HEX)

1:echo.obj
        echo
                example          0096
                area             0006


======================================================================

Segment Map (all values are in HEX):

Segment Offset Size End  Section

<<00>>
        0000  0096 0095 example
        0096  0006 0098 area

======================================================================

Global Symbols and Section Names ( - indicates Section Name):

Symbol           Location (HEX) Section

area               <<00>>0096   -
example            <<00>>0000   -

LINK Complete
```

# 5. THE PDEBUG UTILITY

# THE PDEBUG UTILITY

## ABOUT THIS CHAPTER

This chapter describes how to load the PDEBUG utility, and details all the PDEBUG commands.

## CONTENTS

# THE PDEBUG UTILITY

## INTRODUCTION

The PDEBUG (Program DEBUG) utility is used for debugging and testing programs. When the PDEBUG utility is invoked the M20 enters the PDEBUG environment, the prompt is changed to an asterisk and the cursor stops blinking; the M20 is ready to execute any PDEBUG command. This utility is stored on disk in a ".sav" type of file so that once it is loaded in the M20's memory it remains there until the system is re-booted.

## LOADING AND INVOKING PDEBUG

There are two ways in which the "pdebug.sav" file can be loaded in the M20's memory for the rest of a working session; 1. by executing a PDEBUG command from PCOS (see below), or 2. by PLOADing the utility (see the PLOAD command in the "M20 PCOS User Guide").

When PDEBUG is in memory the user can enter the PDEBUG environment in any of the following ways:

- by executing a PDEBUG command from PCOS (see below)

- by pressing /CTRL//B/ when the M20 is in Execution mode (see below)

Moreover as PDEBUG modifies some tables in PCOS when it is loaded into memory, the following conditions also cause PDEBUG to be entered: Segment Violation Traps, Extended Processing Traps. Priveledged Instruction trap, Illegal Vectored Interrupts, and Non-Maskable Interrupts.

Another way of entering and exiting the PDEBUG environment is possible with the use of breakpoints. This is described in detail in the PDEBUG BREAKPOINT command description.

## PDEBUG

Loads and invokes the PDEBUG utility, optionally loading a specified program from disk to memory.

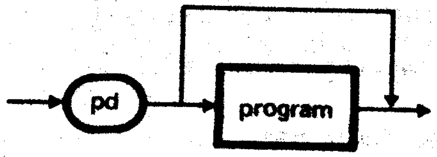


Fig. 5-1 The PDEBUG command

Where

| SYNTAX ELEMENT | MEANING |
|---|---|
| program | EITHER<br>the first two letters of a program name which has a ".sav", or a ".cmd" extension,<br>OR<br>the file identifier of a program file complete with any necessary volume identifier, extension, and/or file password. |

**Example**

If both the PDEBUG utility and the program file "myprog.cmd" exist on any disk inserted in any of the two drives, and,

| IF you enter | THEN ... |
|---|---|
| pd my /CR/ | the program "myprog.cmd" is PLOADed and the M20 enters the PDEBUG environment. When the M20 PLOADs "myprog.cmd" the video displays information about the location of "myprog.cmd" in memory. This information will enable the user to access "myprog.cmd" directly in memory. |

## /CTRL//B/

When the M20 is in program execution mode, the /CTRL//B/ key combination will invoke the PDEBUG utility if it is already resident in memory. When /CTRL//B/ is pressed the video displays a message specifying the location in memory where program execution was halted, and the PDEBUG prompt is returned. The interrupted program remains in memory, and control can be returned to it by using the PDEBUG GO or JUMP commands.

## TERMINATING A PDEBUG SESSION

At the end of a PDEBUG session the user can exit the PDEBUG environment and return to PCOS using the QUIT command.

```
q /CR/
```

If the state of the CPU is modified during a PDEBUG session (e.g. by breakpoint usage) then the QUIT command will force a re-boot of PCOS. If the state is not modified then a simple return to PCOS is done.

## ENTERING PDEBUG COMMANDS

PDEBUG commands can be entered when the PDEBUG prompt (*) appears on the screen. Commands can be entered in either upper or lower case and are terminated by a carriage return. All numbers input to and output by PDEBUG are in hexadecimal ASCII format, and may be entered in either upper or lower case.

An address is specified either with a segment number and an offset, or with just an offset. The segment number is enclosed on the left with a less than symbol (<) and on the right with a greater than symbol (>) (i.e. <6> for segment 6). If only an offset is specified then either the last segment number used since PDEBUG was loaded, or, if none were specified yet, segment 0 is assumed by default.

An alternate method of specifying addresses is to use one of the 26 address registers ("a" to "z") preceded by the "@" sign. For example "@r25e" specifies the address given by the contents of register "r" plus "25E". An address register can be set using the OFFSET (register) command.

All the PDEBUD commands are described in this chapter. The commands are listed in alphabetical order. At the end of this chapter there are two PDEBUG tutorial sessions which demonstrate the use of the more commonly used PDEBUG commands.

A list of all the commands is displayed on the screen if the user enters a question mark (?) followed by a carriage return whenever the PDEBUG prompt (*) is returned.

## CALCULATOR FACILITY

When in the PDEBUG environment the M20 can be used as a calculator for quick calculations in hexadecimal. The following binary operations can be performed:

| | |
|---|---|
| + A,B | adds B to A |
| - A,B | subtracts B from A |
| * A,B | multiplies A by B |
| / A,B | divides A by B |

where A and B are positive hexadecimal numbers in the range 0 to FFFF.

In each of these cases the returned result is also in this range, thus if the absolute value of the result (say C) is outside this range then the value returned will be hexadecimal "C mod 10000". For example,

```
        - 2,6           will return the value FFFC

and     + ffff,1        will return the value 0000
```

## THE COMMANDS

### BREAKPOINT

Sets a breakpoint or displays the currently active breakpoints.

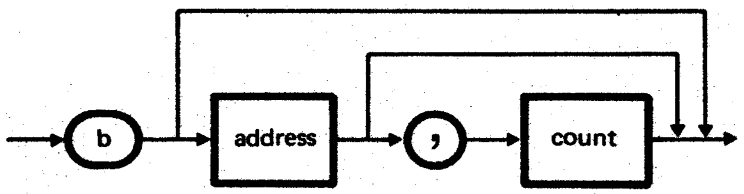


Fig. 5-2 The BREAKPOINT command

**Where**

| SYNTAX ELEMENT | MEANING |
|---|---|
| address | The breakpoint address |
| count | The number of times the breakpoint is meant to execute when encountered. If this parameter is set to 0 then the specified breakpoint executes every time it is encountered, and is not deleted until specifically cleared using the CLEAR breakpoint command. If not specified the breakpoint is deleted when it is hit for the first time. Note that this parameter must be expressed in hexadecimal. |

**Note:**

The BREAKPOINT instruction is not placed in memory until a GO or JUMP command is executed. Thus provisions have to be made to return to PCOS using any one of these commands if the set breakpoints are to be executed.

When the M20 is in execution mode and a breakpoint is encountered, execution is halted, the video displays a break message with the address where the break was encountered, and the PDEBUG prompt is returned.

## CLEAR BREAKPOINT

Clears either an active breakpoint specified by its memory address or all currently active breakpoints.

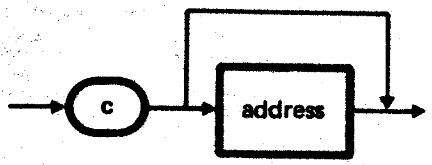


Fig. 5-3 The CLEAR BREAKPOINT command

Where

| SYNTAX ELEMENT | MEANING |
|---|---|
| address | The memory address of an active breakpoint. If this parameter is not specified then all the currently active breakpoints will be cleared. |

## CHANGE I/O

Switches the main input and output from the console to the RS-232-C serial port and vice versa.

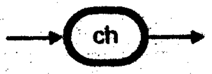


Fig. 5-4 The CHANGE I/O command

Issuing the CHANGE I/O command while using an external terminal causes the main I/O channel to be switched back to the console.

## COMPARE MEMORY

Compares two blocks of memory and returns any differences encountered.

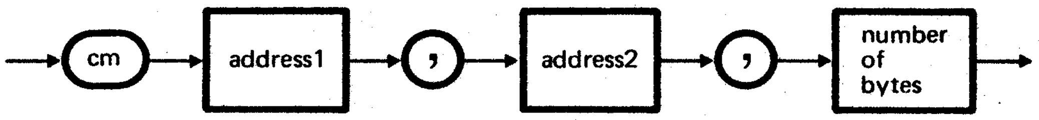


Fig. 5-5 The COMPARE MEMORY command

**Where**

| SYNTAX ELEMENT | MEANING |
|---|---|
| address 1 | The starting point of the first block |
| address 2 | The starting point of the second block |
| number of bytes | The number of bytes to be compared |

While the differences are being output the screen image can be suspended by pressing /CTRL//S/. The command can be aborted by pressing any key. If no differences are found this command simply returns the PDEBUG prompt.

**Note:**

This command uses byte compare operations.

## DISPLACEMENT REGISTER

Sets up a displacement value that will be added to all addresses input and subtracted from all addresses output by the PDEBUG program.

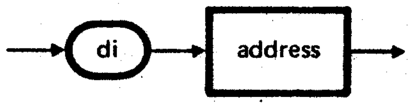


Fig. 5-6 The DISPLACEMENT REGISTER command

Where

| SYNTAX ELEMENT | MEANING |
|---|---|
| address | The displacement value which will be added to all addresses specified in subsequent PDEBUG commands |

The command

```
di /CR/
```

will cause the current default segment and offset to be displayed.

This facility is very useful if a user is working on a listing that has a displaced origin in memory. Using this command the displacement register can be set to the value of the address where the listing begins so that all addresses input and output will match the listing.

## DISPLAY MEMORY

Displays blocks of memory or single memory locations. In the latter case the command interacts with the user for modification of single memory locations.

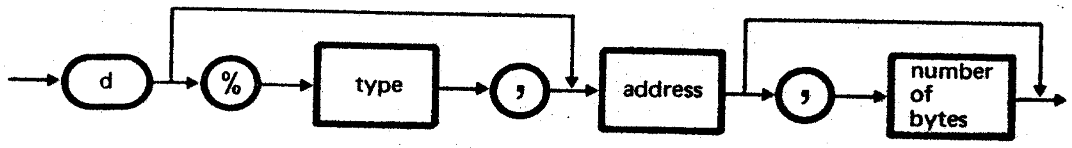


Fig. 5-7 The DISPLAY MEMORY command

Where

| SYNTAX ELEMENT | MEANING |
|---|---|
| type | Word or Byte operations, specified as "W" or "B" (capital or small letters) respectively. Depending on whether the Word or the Byte option is in operation the information will be displayed accordingly. The default value is either the option specified in the last DISPLAY MEMORY or FILL MEMORY command executed in the same PDEBUG session or, in the absence of any, the Word option. |
| address | The memory address where the display is to start |
| number of bytes | The number of bytes to be displayed starting from the address specified in the "address" parameter.<br><br>Note: this number must be expressed in hexa-decimal, and must be greater than 1. |

Characteristics

When the "number of bytes" parameter is specified, the M20 displays the specified memory block in lines of sixteen bytes each. Each line is organized in the following way:-

> The memory address of the first of the sixteen bytes is on the extreme left followed by the contents of the sixteen bytes expressed in hexadecimal code and grouped in words (or in bytes if the "B" (byte) option is specified). If the "number of bytes" paremeter is greater than or equal to sixteen, then the ASCII translation of the sixteen bytes is displayed on the right on the same line. Codes that have no ASCII translation are represented by dots.

When blocks of memory are being displayed, any scroll movement can be halted by entering any character on the keyboard, output can be resumed by entering any character on the keyboard a second time. If you enter the key combination /CTRL/ /C/ then the output will be terminated and the PDEBUG promt is returned.

## Modification of Words

If the "number of bytes" parameter is not specified, then the word starting at the memory address specified is displayed followed by the cursor. At this point you can do any of the following operations:-

| IF you enter | THEN ... |
|---|---|
| /CR/ | the next memory word is displayed. |
| ^/CR/ | the preceding memory word is displayed. |
| (a valid hex number) /CR/ | the content of the displayed word is changed to the hex number entered, and the next memory location is displayed. |
| @/CR/ | the current and next words are interpreted as an address and the word specified by that address is displayed. |
| "(string) /CR/ | the string entered is written directly into memory (in hex code) starting from the current address. |
| q /CR/ | the PDEBUG prompt is returned. |

## FILL MEMORY

Fills a specified block of memory with a given word or byte pattern.

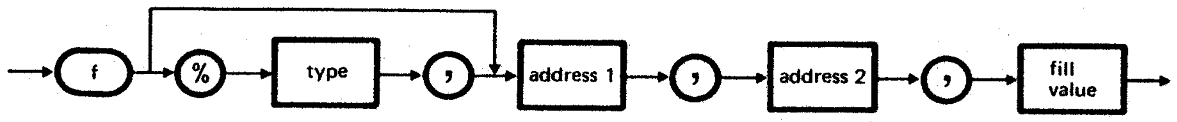


Fig. 5-8 The FILL MEMORY command

Where

| SYNTAX ELEMENT | MEANING |
|---|---|
| type | Word or byte operations, specified as "W" or "B" (capital or small letters) respectively. Depending on whether the Word or the Byte option is in operation the fill value will be interpreted as a word or a byte respectively. The default value is either that specified in the last DISPLAY MEMORY or FILL MEMORY command executed in the same PDEBUG session, or, in the absence of any, the Word option. |
| address 1 | The memory address where the writing operation is to start. |
| address 2 | The memory address where the writing operation is to end. Note that the final location is not written to. |
| fill value | Fill Value. This is the word (or byte if "B" is specified in the "type" parameter) pattern, expressed in hexadecimal code to be written in the specified memory block. |

## GO

Resumes the execution of a program at the location specified by the program counter.

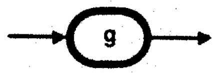


Fig. 5-9 The GO command

## Characteristics

Execution of this command causes all the breakpoints (previously specified in the same PDEBUG session) to be placed in memory prior to the start of execution.

## JUMP

Executes a memory resident program starting from a specified address.

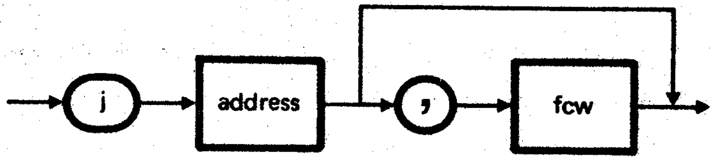


Fig. 5-10 The JUMP command

Where

| SYNTAX ELEMENT | MEANING |
|---|---|
| address | The memory address where execution is to start |
| fcw | Flag and Control Word. |

## Characteristics

This command causes all of the breakpoints (previously specified in the same PDEBUG session) to be placed in memory prior to the start of execution.

## MOVE MEMORY

Copies a source memory block into a target memory block.

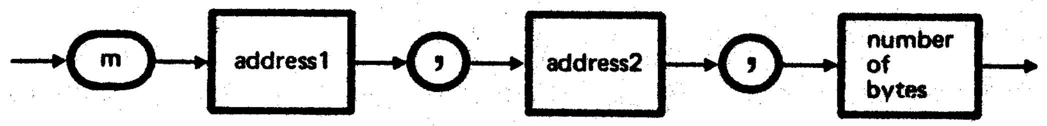


Fig. 5-11 The MOVE MEMORY command

**Where**

| SYNTAX ELEMENT | MEANING |
|---|---|
| address 1 | The memory address where the source memory block begins. |
| address 2 | The memory address where the target memory block begins. |
| number of bytes | The number of successive bytes starting from the beginning of the source block to be copied. |

## OFFSET REGISTER

Sets an offset register to a given address.

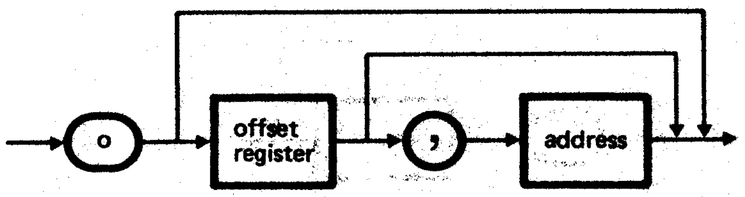


Fig. 5-12 The OFFSET REGISTER command

Where

| SYNTAX ELEMENT | MEANING |
|---|---|
| offset register | Any one of the 26 offset registers ("a" to "z") |
| address | The memory address to be associated with the offset register. |

When the "address" parameter is left out the specified register is printed with its current address. The command without parameters prints all the offset registers with their current addresses.

Offset registers can be used when specifying an address in any PDEBUG command. If register "x" is set to "<2>1000" then "@x5" will represent the address "<2>1005" in any PDEBUG command. This facility is very useful when dealing with module listings; offset registers can be set to the beginning address of each section.

## NEXT

Executes one or more program instructions starting at the location specified by the Program Counter (PC).

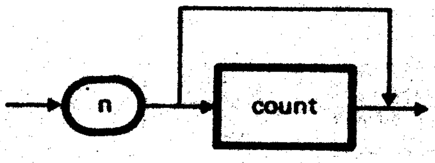


Fig. 5-13 The NEXT command

**Where**

| SYNTAX ELEMENT | MEANING |
|---|---|
| count | The number of instructions to be executed. The default value is one instruction. |

Characteristics

When a specified number of instructions are executed using a NEXT command, the registers are saved, and a message indicating the address of the last instruction executed and the current value of the PC (i.e. the address of the next instruction) is displayed.

A NEXT command is aborted if a breakpoint is encountered in the specified sequence of instructions.

The following situations cause the NEXT command to crash:

- using NEXT through instructions that modify the PSAP (Program Status Area Pointer) in the CPU.

- using NEXT through instructions that disable the non-vectored interrupt.
- using NEXT through instructions that change the programming of the 8253 timer chip.

**PORT (I/O) READ**

Reads a specified I/O port.

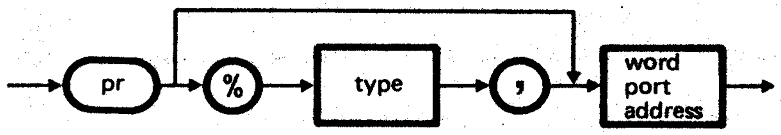


Fig. 5-14 The PORT (I/O) READ command

Where

| SYNTAX ELEMENT | MEANING |
|---|---|
| type | Word or Byte operations specified as "W" or "B" (capital or small letters) respectively. The default value is either the option specified in the last PORT (I/O) READ or PORT (I/O) WRITE command executed in the same PDEBUG session, or, in the absence of any, the Byte option. |
| port address | A valid I/O port address. A list of all the M20 I/O port addresses is given in appendix F. |

## PORT (I/O) WRITE

Writes to a specified port address

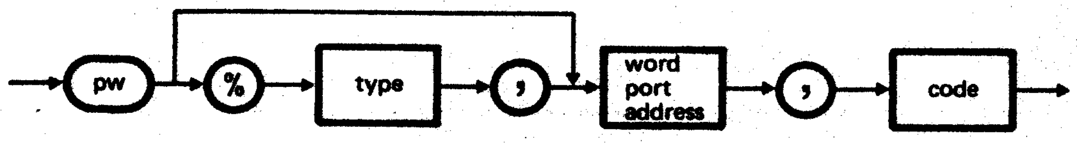


Fig. 5-15 The PORT (I/O) WRITE command

Where

| SYNTAX ELEMENT | MEANING |
|---|---|
| type | Word or Byte operations specified as "W" or "B" (capital or small letters) respectively. The default value is either the option specified in the last PORT (I/O) READ or PORT (I/O) WRITE command executed in the same PDEBUG session, or, in the absence of any, the Byte option. |
| port address | A valid I/O port address. A list of all the M20 I/O port addresses is given in appendix F. |
| code | The hexadecimal code of the byte (or word, if the "word" option is specified) to be written to the port. |

## PRINT OUTPUT

Toggles a flag which causes all output from the PDEBUG program to be sent to a parallel printer as well as to be displayed on the console.

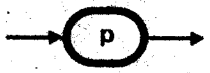


Fig. 5-16 The PRINT OUTPUT command

This means that the first "p" command during a PDEBUG session will cause output to be sent to the printer, and the second will turn off the output to the printer.

## QUIT

Causes a return to the PCOS environment.

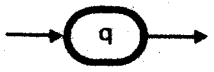


Fig. 5-17 The QUIT command'

Note:

Depending on the state of the CPU the QUIT command will cause either a simple return to the PCOS environment or a re-boot of PCOS.

## REGISTER

Displays or modifies the registers saved in memory.

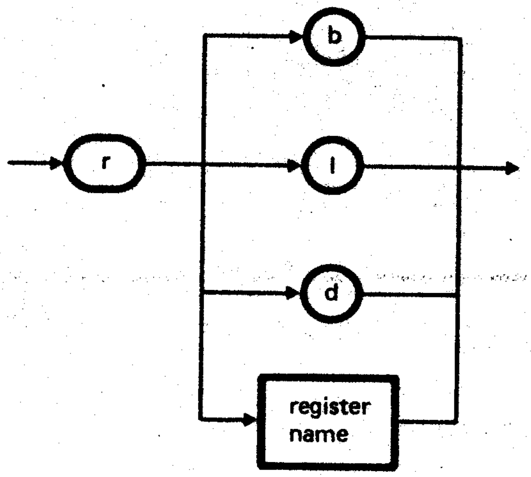


Fig. 5-18 The REGISTER command

**Where**

| SYNTAX ELEMENT | MEANING |
|---|---|
| b | The registers are displayed as byte registers. |
| l | The registers are displayed as word registers. |
| d | All the registers changed by the last GO or JUMP command will be displayed. |
| register name | A valid register name. With this option the specified register will be displayed, and and subsequently the user can modify the contents of it by entering a valid hexadecimal number. |

If the command is entered without any parameters, then all the registers are displayed as word registers.

## The Registers

When the PDEBUG environment is invoked the registers are initialized to the following values:

| REGISTER | INITIALIZED TO |
|---|---|
| r0 to r13 | zero |
| System Stack Pointer and Normal Stack Pointer | a stack space of 16 words in length |
| Program Status Area Pointer (PSAP) | the PCOS program status area |
| Flag and Control Word (FCW) | system mode, segmented mode with interrupts enabled. |
| Program Counter (PC) | the "return to PCOS" address. |

## TRACE

Traces through "count" number of instructions, starting from the instruction specified by the program counter, optionally including any calls, call relatives, or system calls (otherwise treated as a single instruction), and optionally displaying any changed registers after each instruction.

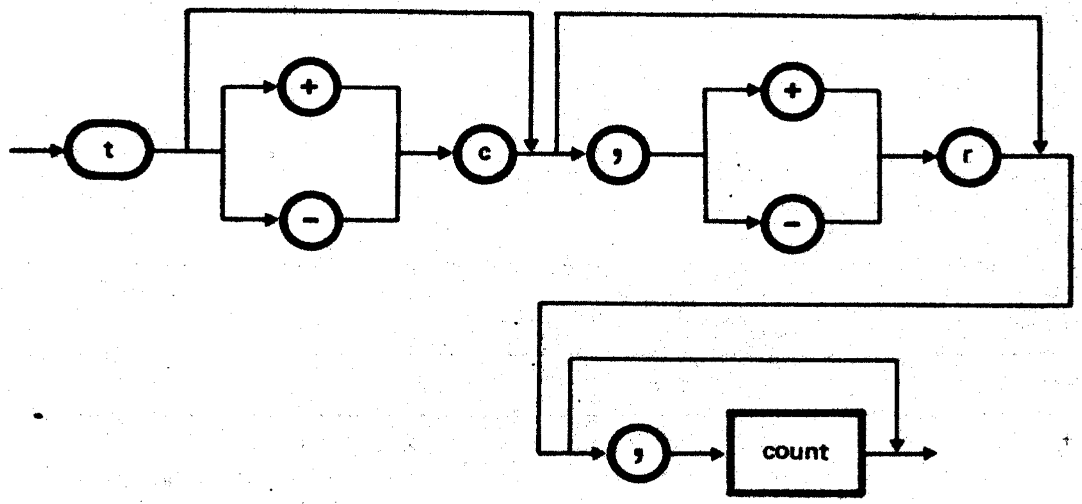


Fig. 5-19 The TRACE command

**Where**

| SYNTAX ELEMENT | MEANING |
|---|---|
| c | Calls will be included while tracing |
| r | Any changed registers will be displayed after each instruction. |
| count | The number of instructions to be executed in each command. |

The "+" and "-" sign turn the "c" and "r" options on and off respectively.

When not specified, parameters assume the values specified in the last TRACE command, or, in the absence of any, the following command is executed:

```
        t -c,+r,1
```

TUTORIAL PDEBUG SESSION 1

| DISPLAY | COMMENTS |
|---|---|
| 0>pd ec Disk file name = echo.cmd Program name = File Echo Operation Mode = Segmented / System Main entry = <0A>%DD08; Init entry = --None-- Memory allocated: Block No. %00; Starting address = <0A>%DD06; Size = %009E PDebug Rev. 2.0 | This command PLOADs the program file "myprog.cmd" and invokes the PDEBUG environment from PCOS. The PLOAD signon message is displayed and the PDEBUG prompt is returned. |
| * di <a>dd06 | Here the DISPLACEMENT REGISTER command is used to set up a displacement value so that address "<a>dd06" (which in this case is the starting address of "echo.cmd") now becomes "0" |
| * r pc pc =<0>4EDA : <a>0002 * r r0 r1 r2 r3 r4 r5 r6 r7 0000 0000 0000 0000 0000 0000 0000 0000 r8 r9 r10 r11 r12 r13 r14 r15 0000 0000 0000 0000 0000 0000 2468 5555 r14' r15' fcw psap pc 8200 FFCC 0820 <2>23FA <2>0002 | The REGISTER command is here used to set the program counter (PC) to the main entry address of the program in memory. |
| * t +c,+r,3 pc=<A>0016 12=6A00 13=000A File Echo pc=<A>0018 | Having set the program counter to the main entry of "echo.cmd" this command traces through the first three instructions. |
| * t pc=<A>001A 0=0A00 pc=<A>001C pc=<A>001E 3=0A00 | This second trace command assumes the parameter values of the preceding one. |
| * q | This QUIT command causes a re-boot of PCOS. |

TUTORIAL PDEBUG SESSION II

This tutorial session is only aimed at demonstrating PDEBUG commands, it does not do anytbing in particular.

| DISPLAY | COMMENTS |
|---|---|
| 0> pd ec<br><br>Disk file name = echo.cmd<br><br>Program name = File Echo<br><br>Operation Mode = Segmented / System<br><br>Main entry = <0A>%DD08; Init entry = --None--<br><br>Memory allocated:<br><br>Block No. %00; Starting address = <0A>%DD06; Size = %009E<br><br>PDebug Rev. 2.0 | This command PLOADs the program file "echo.cmd" and invokes the PDEBUG environment from PCOS. The PLOAD signon message is displayed and the PBEBUG prompt is returned |
| * b <a>dd08 | A BREAKPOINT command is used to set up a breakpoint in in segment ten offset DD08. The breakpoint will be deleted the first time it is encountered. |
| * g<br>1> ec param<br>*** BREAK AT <A>DD08 | Here the GO command causes a return to PCOS. This happens because of the default setting of the PC on entering the PDEBUG environment. The previously set breakpoint is placed in memory so that it is executed when "echo.cmd" is invoked. |
| * r<br><br>r0 r1 r2 r3 r4 r5 r6 r7<br>0A00 D800 0A00 DD08 9200 FFC8 FFFF FFFF<br><br>r8 r9 r10 r11 r12 r13 r14 r15<br>FFFF FFFF 0A00 FF34 8200 16DF 2468 5555<br><br>r14' r15' fcw pseg pc<br>8200 FFC2 D600 <2>0100 <A>DD08 | When the breakpoint is hit, the PDEBUG prompt is returned and any PDEBUG command can be executed; in this case the REGISTER command is used to display the current values of all the registers.<br><br>Note that the PC is now set to the next instruction of the interrupted program. |
| * o a,<a>dd06 | Here the OFFSET REGISTER command is used to set the offset register "a" to the main entry address of "echo.cmd". |

| | |
|---|---|
| `* d @a2,30`<br><br>`A-DD08 3E06 4669 6C65 2045 6368 6F20 0D00 760C ..File Echo ..v.`<br>`A-DD18 8A00 DD0A 7F59 97E0 8D29 A103 B331 0002 .....Y...[...1..`<br>`A-DD28 96E2 5DC2 8A00 DD9C 9D04 5E0E 8A00 DD46 ..!.......^....F` | Having set the offset register "a", it is here used in a DISPLAY MEMORY command. |
| `* d <2>ffc2`<br>`<2>FFC2 0301 :`<br>`<2>FFC4 8203 :`<br>`<2>FFC6 1687 :-`<br>`<2>FFC4 8203 :@`<br>`<2>1687 DC05 :q` | Here the DISPLAY MEMORY command is used in an interactive way. The PDEBUG prompt is returned when "q" is entered. |
| `* b @a1c` | Another breakpoint is set, here again to execute only once. |
| `* g`<br>`File Echo`<br>`***BREAK AT <A>DD22` | Here the GO command resumes the execution of the interrupted program. However execution is again interrupted by the next breakpoint. |
| `* g`<br>`param`<br>`1>` | This final GO command resumes the execution of the program, and subsequently the PCOS prompt is returned. |

# 6. TEXTDUMP HDUMP AND MLIB

# TEXTDUMP HDUMP AND MLIB

## ABOUT THIS CHAPTER

This chapter describes the following three commands

| | | |
|---|---|---|
| - | TEXTDUMP | For dumping files of ASCII text. |
| - | HDUMP | For dumping files in hexadecimal code |
| - | MLIB | For creating library files. |

## CONTENTS

## TEXTDUMP

Dumps a formatted version of an Assembler source file into a specified file, or on the printer, or to standard output.

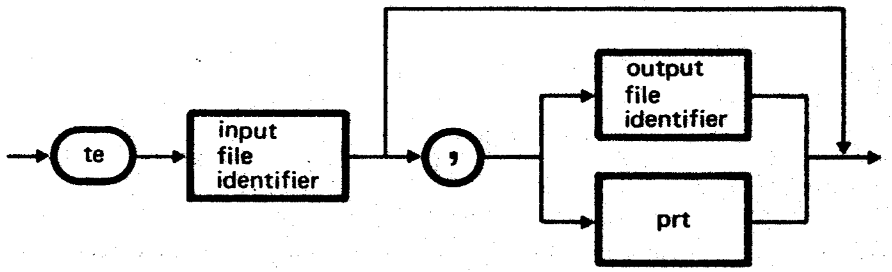


Fig. 6-1 The TEXTDUMP command

**Where**

| SYNTAX ELEMENT | MEANING |
|---|---|
| input file identifier | The name of the source file to be dumped. This must be complete with any necessary volume identifier and/or file password. |
| output file identifier | The file name, complete with any necessary volume identifier and/or file password, of the file which is to contain the formatted dump. This file will be created if it does not exist. If it exists it will be overwritten with the new output. |
| prt | Print output.<br>If this option is specified then the command output will be sent to the printer. |

Characteristics

The TEXTDUMP command without an output parameter (file identifier or prt) will display the output on the screen. In this case the output is displayed 24 lines at a time and the operator must press any key to continue.

**Note:**

Files handled by the TEXTDUMP command utilise either the ASCII "RS" (record separator), or a carriage return, or a carriage return/line feed pair to indicate end-of-line.

**HDUMP**

Dumps the contents of one or more files in hexadecimal code on the standard output device.

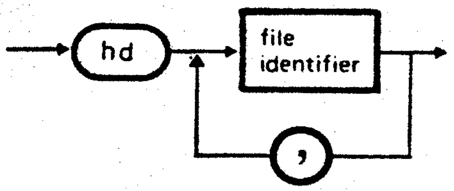


Fig. 6-2 The HDUMP command

**Where**

| SYNTAX ELEMENT | MEANING |
|---|---|
| file identifier | The name of a file complete with any necessary volume identifier, and/or password. |

Characteristics

When a file is dumped using this utility the screen will display the hexadecimal code of the contents of the file on the left side of the screen in lines of sixteen bytes each. Each line is preceded by the byte count of the first byte on the line from the beginning of the file. On the right of each line is the corresponding ASCII representation of the bytes. Code that has no ASCII representation is represented by a dot.

## MLIB

Creates a library file of object modules from a group of Olivetti z-type object files.

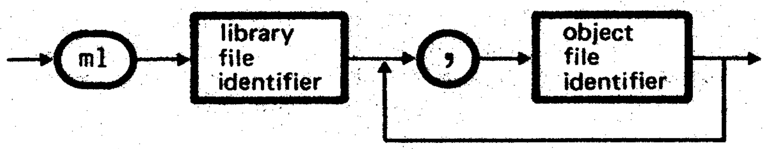


Fig. 6-3 The MLIB command

**Where**

| SYNTAX ELEMENT | MEANING |
|---|---|
| library file identifier | The name of the file which is to contain all the object modules in the specified object files. This must be complete with any necessary volume identifier and/or file password. A library file name is usually assigned the extension ".lib". |
| object file identifier | The name of an object file complete with any necessary volume identifier and/or file password. |

It is common practice to use a library of subroutines to be made available to a series of programs. Mathematical programs, for instance, might use a library of subroutines for calculating trigonometric functions, and Text oriented programs might use a library of string comparison functions.

When LINK discovers an external variable which is not present in any input file, it searches through the list of subroutine names in the library file(s) for a "global" definition. Once the subroutine name is found, the module containing this subroutine is incorporated into the output file. Only the modules referenced by input files, or by library modules which have already been included in the LINK operation, are included in the output file along with the rest of the input modules.

**Note:**

During execution the MLIB command needs to create a temporary work file on the same disk where the command was called from. This means that the file "mlib.cmd" must be called from a write unprotected disk. The best way to do this is to copy this file on to the disk where you want to create your library file; you must then make sure that it is this copy of the command file that is loaded and executed.

**Example**

| IF you enter | THEN |
|---|---|
| ml 1:asm.lib,1:prog1.obj, 1:prog2.obj /CR/ | The file "asm.lib" is created on the disk inserted in drive 1. This file will contain all the object modules contained in the two object files "prog1.obj" and "prog2.obj" both of which are resident on the same disk inserted in drive 1. |

# PART II

# 7. INTRODUCTION TO SYSTEM CALLS

# INTRODUCTION TO SYSTEM CALLS

## ABOUT THIS CHAPTER

This chapter is a general description of the M20 System Calls. The calls are divided in functional groups and the characteristics of each group are discussed. This is followed by the call descriptions.

## CONTENTS

# INTRODUCTION TO SYSTEM CALLS

## INTRODUCTION

These two chapters describe all of the System Calls (SCs) developed for the M20. System Calls are PCOS procedures, used to interface with I/O or to manage memory. System Calls can be accessed by assembly language programs.

All calls made from BASIC, some other utility program, or from user code, will access the I/O and resource management facilities of PCOS via the Z8000 System Call (SC) instruction. The SC instruction includes a 1-byte request code which indicates the function to be performed.

**Example:**

```
        sc #3           system call, request code = 3
```

Parameters are generally passed in registers numbered from R5 to R13. If strings or other large data structures are to be passed, pointers to the structures are passed as parameters in the registers.

In general, parameters are passed as 16-bit unsigned values. ASCII characters are passed occupying the lower bytes of a register

All system calls use R5 to return any error condition. Zero indicates no-error, non-zero indicates the error and condition code.

## SYSTEM CALL DESCRIPTIONS

Each call has been assigned an unique number and a label. The label may be used to reference the call globally, if a table assigning each call number to the respective label is created.

Each call description begins at a new page, and on the page are the name or label, the SC number, and a list of the specific register assignments for each parameter passed. This is followed by a description of the function of the call, and any error codes that might be returned.

The descriptions are arranged in ascending order by SC number.

## REGISTER ASSIGNMENTS

Register assignments are given in synopsis form, and input and output are identified. For example (see SC 32):

## INPUT/OUTPUT PARAMETERS

| | | | |
|---|---|---|---|
| Input: | R7 | ⟵ | block length |
| | RR8 | ⟵ | source address |
| | RR10 | ⟵ | destination address |
| Output: | R5 | ⟶ | error status |

Before calling SC 32, the block length, source address and destination address must be loaded in registers R7, RR8 and RR10 respectively. The only output for this call is the error status, which is returned in R5.

## ERROR MESSAGES

Following the system call, if there are no errors, a zero (0) is returned in R5. If any error occurs, the appropriate error code will be returned. A list of error codes and messages is given in the appendix.

## FUNCTIONAL GROUPS

In this chapter the System Calls are treated in general in functional groups as follows:

- Bytestream Calls
- Block transfer Calls
- Storage Allocation Calls
- Graphics Calls
- Time and Date Calls
- User Code Calls
- IEEE 488 Calls
- Miscellaneous Calls

See the Appendices for lists of system calls in functional groups, for the DIDs (Device IDs) table, as well as for lists of error codes.

## BYTESTREAM CALLS

Bytestream system calls are used for:

a) Transferring bytes of data to or from an I/O device

b) Sending control information to a device or to a device driver

c) Receiving status information from a device
The following are a list of bytestream I/O calls used to interface with the disk, printer, RS-232 communications port, and console (keyboard and video).

LookByte (9)
GetByte (10)
PutByte (11)
ReadBytes (12)
WriteBytes (13)
ReadLine (14)
Eof (16)
ResetByte (18)
Close (19)
SetControlByte (20)
GetStatusByte (21)
OpenFile (22)
DSeek (23)
DGetLen (24)
DGetPosition (25)
DRemove (26)
DRename (27)
DDirectory (28)

## DID (Device IDentifier) Numbers

A DID is an integer used to identify I/O devices (or files) like the keyboard, an open disk file etc.. The operating system maintains a table associating DIDs with a File Pointer. The latter consists of pointers to data structures and routines describing the I/O streams.

## Device Pointers

Opening a disk file creates a stream data structure, and places a pointer to it in the device pointer table. Closing the disk file sets this pointer to nil, and releases any table space associated with the file. Some 'devices' or files are always open. For example, the keyboard and the screen (the default window) are always open. They can, however, be closed and re-opened by using the PCOS Device Rerouting feature.

BASIC file numbers translate simply into PCOS DIDs, but BASIC window numbers for the screen are distinct from DIDs. A table of DID assignments is included in the Appendix.

## Disk Bytestream I/O Calls

Disk input and output are all done by bytestream system calls. A stream structure for an open file maintains a 32-bit pointer to the current position in the file, at which the next byte will be read or written. Files will be extended automatically as they are written, in increments specified by the system globals.

The functions Close, OpenFile, DSeek,DGetLen, DGetPosition, DRemove, DRename and DDirectory are all used for disk files. Of these, only DSeek, DGetPos, DDirectory, DRemove and DRename are disk specific. The other calls can be also used for other devices (printer, console or communication ports). The RS-232 device driver and device rerouting in general are described in the "M20 PCOS User Guide"

## BLOCK TRANSFER CALLS

The block transfer system calls allow the programmer to set memory to a fixed value, to transfer data from one segment to another, and clear memory. In particular, the block transfer calls may be used by the PCOS system to transfer the BASIC interpreter's fixed tables from ROM to RAM in systems with 64K RAM in BASIC's data space.

BASIC will be able to use the block transfer system calls to transfer other tables from ROM to RAM, for initialization of BASIC.

### List of Calls

The following are the Block Transfer calls:

| | |
|---|---|
| BSet (29) | BClear (31) |
| BWSet (30) | BMove (32) |

## STORAGE ALLOCATION CALLS

It is possible for a user program, or BASIC, to call PCOS and then allocate or release heap space. The heap will be in the data segment which is accessible to BASIC. Ordinarily, BASIC will not use this facility directly, since all of its workspace is preallocated at the time of entry into BASIC.

Functions which open a disk file, split a window, or close a file or a window, will use these system calls internally to either allocate heap space or release space.

## INTRODUCTION TO SYSTEM CALLS

When the BASIC command is executed to enter BASIC, 'New' is called to allocate BASIC workspace from the heap. The following are the Storage Allocation calls:

| | |
|---|---|
| NewSameSegment (33) | New (120) |
| Dispose (34) | BrandNewAbsolute (121) |
| MaxSize (99) | NewLargestBlock (122) |
| NewAbsolute (104) | StickyNew (123) |

## GRAPHIC CALLS

The screen area for the M20 display has 256 scanlines by 512 pixels for either black-and-white or (optional) colour display. There is a relationship between the pixels on the screen and the bits of an area in RAM called Bit-Map. This area is grouped in words, and each word in the Bit-Map can be identified by the first word of the graphics accumulator (C-value) described below. The following types of system calls are provided to set global variables or change attributes.

### Clear Window

System call Cls (35) clears the screen (or current window) and positions the cursor(s).

### Cursor(s)

The PCOS system provides two cursors, text or graphics, for the screen. These may be placed anywhere and XORed with the normal contents of the screen. The cursor may be blinking or nonblinking. There is only one cursor displayed for the whole screen. System calls 36 through 44 provide the capability to select the text or graphics cursor, select blinkrate, and update its position:

| | | |
|---|---|---|
| ChgCur0 (36) | ChgCur1 (37) | ChgCur2 (38) |
| ChgCur3 (39) | ChgCur4 (40) | ChgCur5 (41) |
| ReadCur0 (42) | ReadCur1 (43) | SelectCur1 (44) |

### Colour

The M20 is available with either a black and white, or a colour video. Colour videos can be of two types; one type can display 4 colours simultaneously out of a choice of 8 (the four colours can be selected using

System Call 46 "PaletteSet") and the other type can display 8 colours simultaneously.

A colour code is a value from 0 to 7 and is therefore expressed on three bits, say bit 1, bit 2 and bit 3. For a black and white system if a colour code in the range 2-7 is specified then PCOS maps the code to the value obtained when bits 0, 1 and 2 are ORed together. For a four colour system, colour codes in the range 4-7 are mapped into the value obtained when bit 2 is ORed with bit 0.

## Windows

The screen may be divided into windows by splitting along horizontal or vertical lines. There may be a maximum of sixteen windows on the screen, which are assigned window numbers 1 to 16 in order of creation. System calls 45, 47 through 51 and 113 are provided to initialize the screen, create and/or close windows:

| | |
|---|---|
| GrfInit (45) | ChgWindow (50) |
| DefineWindow (47) | CloseWindow (51) |
| SelectWindow (48) | CloseAllWindows (113) |
| ReadWindow (49) | |

## Graphics Accumulator

The graphics routines make use of a global variable referred to as the 'graphics accumulator' to define the current absolute screen location. This graphics accumulator is said to be of type 'C'. A C-variable is a 32-bit variable containing a memory address and a bit mask for the specified group of pixels at that address. The "memory address" (2 bytes) selects a word in the Bit-Map area, and is in the range %0 to %3FFE (8192 words). The "bit mask" is a word each bit of which relates a pixel on the screen to a bit in that area of the Bit-Map specified in the "memory address" (bit=1 for ON and bit=0 for OFF). For example, if the graphics accumulator is assigned the value %20208000 then the first word identifies the sixteen pixels at the centre of the screen and the second word selects the first of these sixteen pixels. Conversion routines are provided for converting local x-y coordinates for windows to or from the C-type variable in the graphics accumulator. Most plotting routines manipulate the graphics accumulator in an abstract and machine-independent way. In general, the plotting of a point is at the position defined by the contents of the graphics accumulator.

Likewise, the 'current attribute' is a global variable representing the current foreground colour. Any plotting or painting routine will set this to the colour specified in the higher-level BASIC (or other) routine by using SetAtr (set attribute), SC 61, or is assumed to be the current window's current foreground colour by default.

## INTRODUCTION TO SYSTEM CALLS

A set of system calls (52 through 67,115 and 116) are provided for scaling or converting coordinates, for manipulating the accumulator, and for drawing lines:

| | | |
|---|---|---|
| ScaleXY (52) | DownC (58) | NSetCX (64) |
| MapXYC (53) | LeftC (59) | NSetCY (65) |
| MapCXY (54) | RightC (60) | NRead (66) |
| FetchC (55) | SetAtr (61) | NWrite (67) |
| StoreC (56) | SetC (62) | ClearText (115) |
| UpC (57) | ReadC (63) | ScrollText (116) |

### Paint Graphics Calls

M20 BASIC supports a PAINT operation which fills an area of a window bounded by a specified boundary colour (and the window boundaries) with another specified brush colour. The following system calls are used to implement the PAINT operation:

| | |
|---|---|
| PntInit (68) | ScanL (71) |
| TDownC (69) | ScanR (72) |
| TUpC (70) | |

These calls set the global colour attributes, move the position of the graphics accumulator up or down, (checking first if the move is within the boundaries of the current window, if not an error is returned); and scan left or right to paint the window.

## TIME AND DATE CALLS

The M20 system has a real-time clock which maintains both date and time. This clock must be reset each time the system is turned on.

Time or date setting are done by passing the address of an ASCII string to the operating system. Likewise, the time or date may be read by transferring an ASCII string from the operating system. The format of these strings are defined by the calls listed below. These will correspond to the string values passed in Basic by manipulating the TIME$ and DATE$ pseudo-strings.

The following system calls perform clock reading and setting:

| | |
|---|---|
| SetTime (73) | GetTime (75) |
| SetDate (74) | GetDate (76) |

## USER CODE CALLS

One system call has been provided to allow the user to execute any program or routine on diskette that could be executed from the PCOS command line. This function works hand in hand with the load-user command which maps user number to physical address. The call is:

```
        CallUser  (77)
```

The call can be used in Assembler utilities to process PCOS user commands.

## IEEE 488 CALLS

The IEEE 488 package consists of a group of programs which execute the following BASIC IEEE statements:

```
        ISET, IRESET, ON SRQ GOSUB, POLL, PRINT@,
        WBYTE, RBYTE, INPUT@, and LINE INPUT@.
```

these statements allow the user to perform the following operations on an IEEE-488 bus:

a) Controlling the IFC (interface clear) and REN (remote enable) lines;

b) Receiving a service request from another device on the bus, identifying the requesting device through serial polling, and processing the service request;

c) Writing control bytes (e.g.: "Device Clear", "Device Trigger", etc.) to other devices;

d) Addressing, writing data to, and reading data from, other devices; and

e) Allowing the devices within an IEEE-488 network to transfer data on the bus (i.e.: assigning "Talker" status to one device, and "Listener" status to one or more devices).

## INTRODUCTION TO SYSTEM CALLS

The following system calls are assigned to the IEEE package. On exiting from any of these procedures, register R5 will contain hex 0A if the system does not have an IEEE option board.

- IBSrQ0 (78)
- IBSrQ1 (79)
- IBPoll (80)
- IBISet (81)
- IBRSet (82)
- IBPrnt (83)
- IBWByt (84)
- IBInpt (85)
- IBLinpt (86)
- IBRByt (87)

For further details on the IEEE-488 interface see the "M20 I/O with External Peripherals User Guide".

## MISCELLANEOUS CALLS

The following miscellaneous calls complete the list of System calls:

- Error (88)
- Dstring (89)
- CrLf (90)
- DHexByte (91)
- DHex (92)
- DHexLong (93)
- DNumW (94)
- DLong (95)
- DisectName (96)
- CheckVolume (97)
- Search (98)
- SetVol (102)
- NewAbsolute (104)
- StringLen (105)
- DiskFree (106)
- BootSystem (107)
- SetSysSeg (108)
- SearchDevTab (109)
- KbSetLock (114)

# 8. THE M20 SYSTEM CALLS

# THE M20 SYSTEM CALLS

## ABOUT THIS CHAPTER

In this chapter all the system calls are described in detail. The descriptions follow each other in numeric order. A list of system calls in functional groups is given in appendix C.

## CONTENTS

THE M20 SYSTEM CALLS

| | |
|---|---|
| 104 NewAbsolute | 8-103 |
| 105 StringLen | 8-104 |
| 106 DiskFree | 8-105 |
| 107 BootSystem | 8-106 |
| 108 SetSysSeg | 8-107 |
| 109 SearchDevTab | 8-108 |
| 113 CloseAllWindows | 8-109 |
| 114 KbSetLock | 8-110 |
| 115 ClearText | 8-111 |
| 116 ScrollText | 8-112 |
| 120 New | 8-114 |
| 121 BrandNewAbsolute | 8-115 |
| 122 NewLargestBlock | 8-116 |
| 123 StickyNew | 8-117 |

## 9 LookByte

Returns the next byte from the designated device buffer without removing the byte from the buffer.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | R8 $\longleftarrow$ | DID |
| Output: | RL7 $\longrightarrow$ | returned byte |
| | RH7 $\longrightarrow$ | buffer status (00 or FF) |
| | R5 $\longrightarrow$ | error status |

### Characteristics

This function returns the first byte of a device input buffer (undefined if none), without removing it from the buffer. The DID is an integer, identifying the device. Valid DIDs are listed below. Also returned is the status of the device buffer, FF if the buffer is not empty, 00 otherwise.

**Note:**

Ring buffers are maintained for the interrupt driven input devices. Characters are placed into the buffers immediately as they are received and are available to programs via the two system calls Look-Byte and GetByte.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

### Valid DID Numbers

| | |
|---|---|
| 17 | console (keyboard) |
| 19,25,26 | Com (RS-232-C), Com1, Com2 |

## 10 GetByte

Returns the first byte from a designated device, removing the byte from the device buffer.

### Input/Output Parameters

Input: $R8 \leftarrow$ DID

Output: $R7 \rightarrow$ returned byte
$R5 \rightarrow$ error status

### Characteristics

This call returns the first byte in the input buffer (from file or designated device) and places that byte in register R7. The DID is an integer which identifies the source of the input. Valid DID numbers are listed below.

In the case where the DID is either 17 or 19, if the input device buffer is empty , the system will wait until a byte is input and available in the buffer before returning to the caller with the byte in R7.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned. Possible error codes, from the file system are shown below (see Appendix for error list).

54, bad_mode_err; 57, disk_io_err; 60, bad_disk_err;
62, eof_err; 63, bad_rec_num_err; 77, illegal_disk_chng_err
90, param_err; 96, file_not_open_err

### Valid DID Numbers

| | |
|---|---|
| 1 - 15 | disk files (BASIC) |
| 17 | console (keyboard) |
| 20 - 24 | disk files (PCOS) |
| 19,25,26 | Com (RS-232-C), Com1, Com2 |

## 11 PutByte

Transmits a byte to a specified device.

### Input/Output Parameters

Input: R8 $\longleftarrow$ DID
RL7 $\longleftarrow$ input byte

Output: R5 $\longrightarrow$ error status

### Characteristics

This transmits the byte supplied in RL7 to the device or file specified by the DID. Valid DIDs are identified below. For files, no information is returned about the validity or EOF state of the DID.

If the device is the RS-232-C port, and the port is not ready to send, the driver will wait for a timeout period and then return an error if nothing is sent.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned. Some possible error codes are (see Appendix for list):

54, bad_mode_err; 57, disk_io_err; 61, disk_full_err;
77, illegal_disk_chng_err; 90, bad_param_err;
96, file_not_open_err

### Valid DID Numbers

| | |
|---|---|
| 1 - 15 | disk files (BASIC) |
| 17 | console (keyboard) |
| 18 | printer |
| 20 - 24 | disk files (PCOS) |
| 19,25,26 | Com (RS-232-C), Com1, Com2 |

## 12 ReadBytes

Reads and counts bytes, from a device, into a buffer in memory.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | R8 $\leftarrow$ | DID |
| | R9 $\leftarrow$ | count to be read |
| | RR10 $\leftarrow$ | pointer to memory buffer |
| Output: | R7 $\rightarrow$ | count returned |
| | R5 $\rightarrow$ | error status |

### Characteristics

FILES

This function reads a specified number of bytes from a file into memory, and returns a count of the number of bytes actually read.

The count returned is used to determine EOF status for the file. The EOF status is determined when the "count returned" in R7 is less than the "count to be read" input in R9, (because there are no more bytes to be read).

The input to RR10 is a segmented pointer to the first byte of memory where these bytes will be stored. The output "count returned" is the actual number of bytes read.

RS-232-C

This call transfers a specified number of bytes from the input buffer to the user specified buffer.

If the number of characters in the input buffer is less than the number requested, the driver will wait for the needed characters to arrive.

## Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned. Some possible error codes are (see Appendix for list):

54, bad_mode_err; 57, disk_io_err; 60, bad_disk_err;
62, eof_err; 63, bad_rec_num_err; 77, illegal_disk_chng_err
90, param_err; 96, file_not_open_err

## Valid DID Numbers

| | |
|---|---|
| 1 - 15 | disk files (BASIC) |
| 20 - 24 | disk files (PCOS) |
| 19,25,26 | Com (RS-232-C), Com1, Com2 |

## 13 WriteBytes

Writes a specified number of bytes from memory to a file or device.

### Input/Output Parameters

Input:

$$\begin{aligned} &R8 \longleftarrow \text{DID} \\ &R9 \longleftarrow \text{count} \\ &RR10 \longleftarrow \text{start} \end{aligned}$$

Output:

$$\begin{aligned} &R7 \longrightarrow \text{count returned} \\ &R5 \longrightarrow \text{error status} \end{aligned}$$

### Characteristics

FILES

This function writes a specified number of bytes from memory into a file. It returns a count of the number of bytes actually transferred. Valid DIDs are listed below.

The input "count" is the number of bytes to be transferred. The input "start" is a segmented pointer to the first byte in memory from which these bytes will be written.

The output "count returned" is the actual number of bytes transferred.

RS-232-C

This call transfers data bytes from the specified memory buffer to the RS-232-C port.

The meanings of the inputs and outputs is the same. If the port is not ready to send, the driver will wait a timeout period, and then return an error if nothing is sent.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

Some possible error codes are (see Appendix for list):

54, bad_mode_err; 57, disk_io_err; 61, disk_full_err;
62, eof_err; 77, illegal_disk_chng_err;
90, bad_par_err; 96, file_not_open_err

**Valid DID Numbers**

| | |
|---|---|
| 1 - 15 | disk files (BASIC) |
| 20 - 24 | disk files (PCOS) |
| 19,25,26 | Com (RS-232-C), Com1, Com2 |

## 14 ReadLine

Reads and counts bytes input from the keyboard, until the first /CR/, into a memory buffer (at a specified address).

### Input/Output Parameters

| | |
|---|---|
| Input: | R8 ⟵ DID |
| | R9 ⟵ count |
| | RR10 ⟵ destination |
| Output: | R6 ⟶ count returned |
| | R5 ⟶ error status |

### Characteristics

This function reads a specified number of bytes from the standard input device into memory. Input will be terminated when the next input byte is equal to /CR/ or if the maximum "count" is exceeded. The /CR/ is not put into the string.

The input DID (17) identifies the standard input. It is the only valid DID for this call. The input "count" specifies the maximum number of bytes to be read, and the input "destination" is a pointer to address of the first byte of memory where these bytes will be stored.

The output "count returned" is the actual size, in bytes, of the input string. If a /CTRL//C/ is pressed, R6 will return a 'FFFF'. Characters are echoed to the standard output device (DID 17) and editing features,(/CTRL//H/ i.e.:backspace and /CTRL//I/, i.e.: TAB) and hide mode /CTRL//G/ are implemented.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

### Valid DID Numbers

| | |
|---|---|
| 17 | Console (keyboard) only |

## 16 Eof

Checks if an input character is available from device.

### Input/Output Parameters

Input: R8 $\longleftarrow$ DID

Output: R9 $\longrightarrow$ returned status
R5 $\longrightarrow$ error status

### Characteristics

The function "EOF" (end of file) will return a zero (0) if an input character is available from the selected device.

It returns a one (1) in each of the following cases:

1. The selected file is not open.
2. The file is open for output only.
3. The console has been selected but no key has been struck.
4. The end of the disk file has been reached.

The input "DID" identifies the device; valid DIDs are listed below.

RS-232-C

For use with the RS-232-C, this call returns a zero (0) if the input buffer is not empty, and a one (1) if the buffer is empty.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

Some possible error codes are (see Appendix for list):

90, bad_par_err; 96, file_not_open_err

**Valid DID Numbers**

| | |
|---|---|
| 1 - 15 | disk files (BASIC) |
| 17 | console |
| 19,25,26 | Com (RS-232-C), Com1, Com2 |

## 18 ResetByte

Resets an input file or device.

### Input/Output Parameters

| | |
|---|---|
| Input: | R8 $\longleftarrow$ DID |
| Output: | R5 $\longrightarrow$ error status |

### Characteristics

This function is used to reset an input device. In the case of the console, it will clear the keyboard ring buffer, and initialize the screen driver. It can also be used with communications (RS-232-C), in which case it re-initializes the hardware and clears the input buffer. The input "DID" identifies the device.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

### Valid DID Numbers

| | |
|---|---|
| 17 | console |
| 19,25,26 | Com (RS-232-C), Com1, Com2 |

## 19 Close

Closes specified disk file or device.

### Input/Output Parameters

| | |
|---|---|
| Input: | R8 ⟵ DID number |
| Output: | R5 ⟶ error status |

### Characteristics

This call closes the specified file or device and then releases both buffer and table space. The input "DID" is an integer representing the file or device.

**Note:**

This call is not used to close screen windows (see CloseWindow, SC 51).

RS-232-C

When used with the RS-232-C, the call disables the hardware interrupts, and the input buffer is removed from the heap.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned. Some possible error codes are (see Appendix for list):

57, disk_io_err; 77, illegal_disk_chng_err;
90, param_err;

### Valid DID Numbers

| | |
|---|---|
| 1 - 15 | disk files (BASIC) |
| 20 - 24 | disk files (PCOS) |
| 19, 25, 26 | Com (RS-232-C), Com1, Com2 |

## 20 SetControlByte

Writes a word into the Device Parameter Table.

### Input/Output Parameters

|  |  |  |
|---|---|---|
| Input: | R8 $\longleftarrow$ | DID |
|  | R9 $\longleftarrow$ | word number |
|  | R10 $\longleftarrow$ | word |
| Output: | R5 $\longrightarrow$ | error status |

### Characteristics

This call allows a single word to be written into the Device Parameter Table (see appendix H). The input to R9 is the word number to be written to; the input to R10 is the word to be written to the Device Parameter Table.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

### Valid DID Numbers

|  |  |
|---|---|
| 19, 25, 26 | Com (RS-232-C), Com1, Com2, |

## 21 GetStatusByte

Reads a single word from the Device Parameter Table.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | R8 $\longleftarrow$ | DID |
| | R9 $\longleftarrow$ | word number |
| Output: | R10 $\longrightarrow$ | word read |
| | R5 $\longrightarrow$ | error status |

### Characteristics

This call allows a single word to be read from the Device Parameter Table (see appendix H). The input to R9 is the word number to be read. The outputs are the words read from the Device Parameter Table (in R10), and the error status (in R5).

### Errors

If there are any errors, a non-zero number will be returned in R5. If there are no errors, a zero (0) will be returned.

### Valid DID Numbers

| | |
|---|---|
| 19, 25, 26 | Com(RS-232-C), Com1, Com2 |

## 20 SetControlByte

Writes a word into the Device Parameter Table.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | R8 $\longleftarrow$ | DID |
| | R9 $\longleftarrow$ | word number |
| | R10 $\longleftarrow$ | word |
| Output: | R5 $\longrightarrow$ | error status |

### Characteristics

This call allows a single word to be written into the Device Parameter Table (see appendix H). The input to R9 is the word number to be written to; the input to R10 is the word to be written to the Device Parameter Table.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

### Valid DID Numbers

| | |
|---|---|
| 19, 25, 26 | Com (RS-232-C), Com1, Com2, |

## 21 GetStatusByte

Reads a single word from the Device Parameter Table.

### Input/Output Parameters

|  |  |  |  |
|---|---|---|---|
| Input: | R8 | ← | DID |
|  | R9 | ← | word number |
| Output: | R10 | → | word read |
|  | R5 | → | error status |

### Characteristics

This call allows a single word to be read from the Device Parameter Table (see appendix H). The input to R9 is the word number to be read. The outputs are the words read from the Device Parameter Table (in R10), and the error status (in R5).

### Errors

If there are any errors, a non-zero number will be returned in R5. If there are no errors, a zero (0) will be returned.

### Valid DID Numbers

|  |  |
|---|---|
| 19, 25, 26 | Com(RS-232-C), Com1, Com2 |

## 22 OpenFile

Opens a specified file or device for read, write, etc.

### Input/Output Parameters

| | Files | RS-232-C |
|---|---|---|
| Input: | R6 ← extent length | |
| | R7 ← mode | |
| | R8 ← DID | R8 ← DID |
| | R9 ← file identifier length | |
| | RR10 ← address | |
| Output: | R5 ← error status | R5 → error status |

### Characteristics

DEVICES

The function of this call is to open the specified device; its characteristics, however, depend upon the device. For example, for the RS-232-C there are no parameters except the input DID.

FILES

In this case the function of this call is to open the designated file, specify the mode (append, read, write, or read/write), and to allocate sectors (write or append modes only).

The input "file identifier length" is the number of characters in the file identifier. The input "address" is the address of the file identifier.

The input "mode" designates whether the file will be opened for read, write or append, as follows:

0: Read, always from current position.

1: Write, always placing a new end of file.

2: Read/Write, allocating sectors beyond old EOF.

3: Append, seeks to end upon open, and then writes.

A file that does not exist cannot be opened in the read mode. A nonexistent file, if opened by write or read/write, will be created. If it does exist, write mode will write over the old file.

If an existing file has been opened in the read/write mode, the user can then position the file pointer to its end, to extend it, using Dseek (SC 23). However, Append mode does this automatically, and then operates the same as the write mode.

The input "extent length" designates the number of sectors to be allocated if the file is to be created. The request should always be one sector larger then the data requirements. If a zero is entered, the number of sectors will be the default value (usually 8). The input DID number identifies the file (see list below).

## Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned. Some possible error codes are (see Appendix for list):

```
 7, mem_full_err; 53, file_not_found_err; 55, file_open_err;
57, disk_io_err; 61, disk_full_err; 64, bad_filenam_err;
71, volnam_not_found_err; 73, volnum_err;
75, vol_not_enab_err; 76, invalid_string_err;
77, illegal_disk_chng_err;  90, bad_par_err;
```

## Valid DID Numbers

| | |
|---|---|
| 1 - 15 | disk files (BASIC) |
| 20 - 24 | disk files (PCOS) |
| 19,25,26 | Com (RS-232-C), Com1, Com2 |

## 23 DSeek

Positions file pointer as specified.

### Input/Output Parameters

Input: R8 $\leftarrow$ DID
RR10 $\leftarrow$ position

Output: R5 $\rightarrow$ error status

### Characteristics

This will position the file pointer for the specified stream (opened file) to the position specified. The input "DID" identifies the device. The input "position" is a 32-bit pointer. Zero is the first byte.

Seeking past the EOF while the file is opened for read/write will automatically allocate new sectors.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned. Some possible error codes are (see Appendix for list):

54, bad_mode_err; 57, disk_io_err; 61, disk_full_err;
63, bad_rec_num_err; 77, illegal_disk_chng_err;
90, bad_par_err; 96, file_not_open_err;

### Valid DID Numbers

| | |
|---|---|
| 1 - 15 | disk files (BASIC) |
| 20 - 24 | disk files (PCOS) |

## 24 DGetLen

Returns the length of a file or number of bytes in the input buffer.

### Input/Output Parameters

| | Files | Devices |
|---|---|---|
| Input: | R8 ⟵ DID | R8 ⟵ DID |
| Output: | RR10 ⟶ length | R10 ⟶ zero status |
| | R5 ⟶ error status | R11 ⟶ number |
| | | R5 ⟶ error status |

### Characteristics

DEVICES

This call returns the number of bytes currently in the input buffer. There are no inputs except the DID number.

FILES

This call returns the length of the file as a long word. The output "length" is the length of the file.

### Errors

If there are no errors, a zero (0) will be returned in R5. If the disk file is not open, a -1 is returned in RR10 and error code 96 is returned in R5. If a bad parameter is input, error 90 is returned in R5.

### Valid DID Numbers

| | |
|---|---|
| 1 - 15 | disk files (BASIC) |
| 20 - 24 | disk files (PCOS) |
| 19,25,26 | Com, Com1, Com2 |

## 25 DGetPosition

Gets the position of next byte to be read or written.

### Input/Output Parameters

| | |
|---|---|
| Input: | R8 $\longleftarrow$ DID |
| Output: | RR10 $\longrightarrow$ position |
| | R5 $\longrightarrow$ error status |

### Characteristics

This call returns the position, in bytes, of the next byte to be read or written. The input "DID" identifies the file. A list of valid DIDs is given below.

The output "position" contains the position in the file, in bytes, where the next byte will be read or written.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned. Some possible error codes are (see Appendix for list):

90, param_err; 96, file_not_open_err;

### Valid DID Numbers

| | |
|---|---|
| 1 - 15 | disk files (BASIC) |
| 20 - 24 | disk files (PCOS) |

## 26 DRemove

Removes a specified file name from a disk directory.

### Input/Output Parameters

| | |
|---|---|
| Input: | R9 ⟵ length |
| | RR10 ⟵ address |
| Output: | R5 ⟶ error status |

### Characteristics

This call is used only for disk files. It removes the specified disk file (and related data) from the directory of the volume.

The input "address" points to the file identifier. The input "length" is the length of the file identifier.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned. Some possible error codes are (see Appendix for list):

```
53, file_not_found_err; 55, file_open_err; 57, disk_io_err;
64, bad_filenam_err; 71, volnam_not_found_err; 73, volnum_err;
75, vol_not_enab_err; 76, invalid_string_err;
77, illegal_disk_chng_err;
```

## 27 DRename

Renames a specified file.

### Input/Output Parameters

Input: RR6 $\longleftarrow$ old address
R8 $\longleftarrow$ old length
RR10 $\longleftarrow$ new address
R9 $\longleftarrow$ new length

Output: R5 $\longrightarrow$ error status

### Characteristics

This call is used only for disks. It will rename the file specified by the old file identifier with the new file name.

The input addresses point to the old file identifier and to the new file name respectively. The inputs called "length" are the lengths of the old file identifier and new file name, and are given in words.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned. Some possible error codes are (see Appendix for list):

53, file_not_found_err; 55, file_open_err; 57, disk_io_err;
64, bad_filenam_err; 71, volnam_not_found_err;
73, volnum_err; 75, vol_not_enab_err; 76, invalid_string_err;
77, illegal_disk_chng_err; 90, bad_par_err;

## 28 DDirectory

Displays a list of files from a specified disk.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | R9 ← | file identifier length |
| | RR10 ← | file identifier address |
| Output: | R5 → | error status |

### Characteristics

This call is used only for files. It lists the contents of the directory of the specified volume, on the current window of the M20 screen. The input "length" is the number of bytes in the file identifier. The input "address" is the address of the file identifier. The file identifier may contain a volume identifier and/or wild card characters ("*" and "?"). If R9 is zero, DDirectory assumes the name "*", and will list the entire directory.

The display lists the names of the specified files on the specified (or default) volume in compact form.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned. Some possible error codes are (see Appendix for list):

7, mem_full_err; 57, disk_io_err; 64, bad_filenam_err;
71, volnam_not_found_err; 73, volnum_err; 75, vol_not_enab_err;
76, invalid_string_err; 77, illegal_disk_chng_err;

## 29 BSet

Sets a block of bytes to a specified value.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | RL7 $\longleftarrow$ | n (byte value) |
| | RR8 $\longleftarrow$ | start |
| | R10 $\longleftarrow$ | length |
| Output: | R5 $\longrightarrow$ | error status |

### Characteristics

This call sets a block of memory to the indicated byte value. The input "start" is a segmented pointer to the first byte of memory to be set. The input "length" is the number of bytes to be set.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 30 BWSet

Sets a block of words to a specified value.

### Input/Output Parameters

|  |  |  |
|---|---|---|
| Input: | R7 $\longleftarrow$ | n (word value) |
|  | RR8 $\longleftarrow$ | start |
|  | R10 $\longleftarrow$ | length |
| Output: | R5 $\longrightarrow$ | error status |

### Characteristics

This routine sets the block of memory specified to the input value, n. The input "n" is the word value to be loaded into each memory location. The input "start" is a segmented pointer to the first word of memory to be set. The input "length" is the number of words to be set.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 31 BClear

Sets a specified block of memory to zero.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | RR8 $\longleftarrow$ | start |
| | R10 $\longleftarrow$ | length |
| Output: | R5 $\longrightarrow$ | error status |

### Characteristics

A block of bytes, of the length specified, and starting at a specified source, is set to zero. The input "start" is a segmented pointer to the first byte of memory to be set. The input called "length" is the number of bytes to be set to zero.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 32 BMove

Moves a block of bytes from one location to another.

### Input/Output Parameters

| | | | |
|---|---|---|---|
| Input: | R7 | $\longleftarrow$ | length |
| | RR8 | $\longleftarrow$ | start |
| | RR10 | $\longleftarrow$ | destination |
| Output: | R5 | $\longrightarrow$ | error status |

### Characteristics

A block of bytes, of specified length, and starting at a specified source, is moved to a block starting at a specified destination. The input "start" is a segmented pointer to the first byte of memory to be moved. The input "length" is the number of bytes to be moved. The input "destination" is a segmented pointer to the first byte of the destination memory block.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 33 NewSameSegment

Allocates a block of bytes from heap in the current segment.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | RR8 ⟵ | address of block pointer |
| | R10 ⟵ | length |
| Output: | R5 ⟶ | error status |
| | @RR8 ⟶ | block pointer |

### Characteristics

This call allocates blocks in the "SameSegment". This is segment 2 unless the program has done a "BrandNewAbsolute" system call, in which case the segment number is that specified in the most recent "Brand-NewAbsolute". This call is a subset of System Call 120 "New". It has been maintained for compatibility with preceding releases.
A simple way to change the segment number for a program is to do a SC 121 "BrandNewAbsolute" with a block length of 0.

The input "address of block pointer" is the address of a long word which specifies the start address where NewSameSegment will store the block. The input 'length' is the number of bytes to be allocated. If the block cannot be allocated, RR6 will contain a nil (hex FFFFFFFF) pointer, without returning an error in R5.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 34 Dispose

Releases heap space.

### Input/Output Parameters

```
Input:          RR8 <---- address of block pointer
                R10 <---- length

Output:         @RR8 ----> hex FFFFFFFF
                R5   ----> error status
```

### Characteristics

This routine releases memory space. The input address is a long word, pointing to the start address of this space. It is important that this be a valid heap space. Once the call has been executed, the address specified in RR8 will contain hex FFFFFFFF (nil).

EXAMPLE:

In this example assume that addptr is a long variable which has been initialized as in the example for New (SC 120):

```
LDA    RR8,addptr
LD     R10,#length
sc     #34
```

### Errors

If 'addptr' does not point to the start of a valid heap space, the system issues an error. If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 35 Cls

Clears the current window.

### Input/Output Parameters

This call has no parameters.

### Characteristics

This routine clears the current window to the current background colour (usually black). There are no parameters. The call sets the position of the text cursor to the top left of the window, and sets both the graphics cursor and the accumulator to the center of the window.

### Errors

No error checks are made and no errors are reported.

## 36 ChgCur0

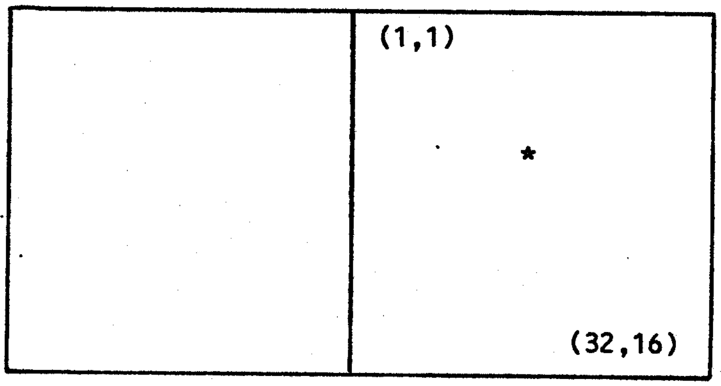
Positions the text cursor.

### Input/Output Parameters

|  |  |
|---|---|
| Input: | R8 ⟵ column<br>R9 ⟵ row |
| Output: | R5 ⟶ error status |

### Characteristics

This routine sets the position of the text cursor, on the current window, to the column and row specified. The upper left corner position of the current window is (1,1). The position of the lower right corner depends upon the display character size (64 by 16 or 80 by 25), and the size of the window (see example below).



* current window, 64 by 16 mode

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 37 ChgCur1

Positions the graphics cursor.

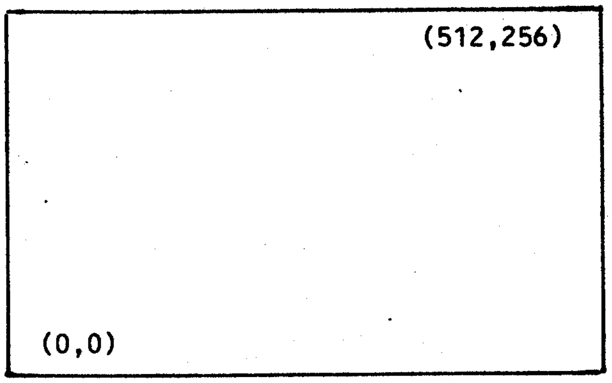
### Input/Output Parameters

| | |
|---|---|
| Input: | R8 $\leftarrow$ x<br>R9 $\leftarrow$ y |
| Output: | there is no output |

### Characteristics

This routine sets the position of the graphics cursor, of the current window, to the x-position and y-position specified.

The lower left corner position of the current window is always (0,0). The position of the upper right corner will depend upon the size of the window and the display character size (64 by 16 or 80 by 25). The example below shows the coordinates for a full screen in 64 by 16 characters format.



### Errors

Range checking is done, and if out of bounds the cursor is not moved; however no error code is returned.

## 38 ChgCur2

Sets the blink rate of the text cursor.

### Input/Output Parameters

| | |
|---|---|
| Input: | R8 ←—— rate |
| Output: | there is no output |

### Characteristics

This routine changes the blink rate of the cursor of the current window to a new value. The value will be the blink rate per second.

Valid values are 0 to 20, with a resolution of 50 ms. A zero value is non-blinking.

### Errors

No error codes are returned.

## 39 ChgCur3

Sets the blink rate of the graphics cursor.

### Input/Output Parameters

| | |
|---|---|
| Input: | R8 $\longleftarrow$ rate |
| Output: | there is no output |

### Characteristics

This routine changes the blink rate of the cursor of the current window to a new value. The value will be the blink rate per second.

Valid values range from 0 to 20, with a resolution of 50 ms. A zero value is non-blinking.

### Errors

No error codes are returned.

## 40 ChgCur4

Sets the shape of the text cursor.

### Input/Output Parameters

| | |
|---|---|
| Input: | RR8 ←—— address |
| Output: | there is no output |

### Characteristics

This call is used to change the shape of the text cursor of the current window. The input "address" points to the address of the new byte array which describes the new shape of the cursor. This array is 12 bytes long, the first byte being the first scan line of the cursor.

It is suggested that the most significant bit of each byte is not used as part of the cursor as it would then touch the previous character.

If the text cursor is being displayed at the time this call is made, it will be turned off, updated, and then turned back on.

EXAMPLES:

For a solid cursor:

```
          array = %7F %7F %7F %7F
                  %7F %7F %7F %7F
                  %7F %7F %7F %7F
```

For a checkerboard:

```
          array = %00 %55 %2A %55
                  %2A %55 %2A %55
                  %2A %55 %2A %55
```

### Errors

No errors are returned.

**41 ChgCur5**

Sets the shape of the graphics cursor.

## Input/Output Parameters

| | |
|---|---|
| Input: | RR8 $\longleftarrow$ address |
| Output: | there is no output |

## Characteristics

This call is used to change the shape of the graphics cursor of the current window. The input "address" points to the address of the new byte array which describes the new shape of the cursor. This array is 12 bytes long, the first byte being the first scan line of the cursor.

It is suggested that the most significant bit of each byte is not used as part of the cursor as it would then touch the previous character.

If the graphics cursor is being displayed at the time this call is made, it will be turned off, updated, and then turned back on.

## Errors

No errors are returned.

## 42 ReadCur0

Returns the position (column and row), and the blinkrate of the current window's text cursor.

### Input/Output Parameters

Input: $\quad$ RR10 $\longleftarrow$ address

Output: $\quad$ R7 $\longrightarrow$ blinkrate
R8 $\longrightarrow$ column
R9 $\longrightarrow$ row
R5 $\longrightarrow$ error status

### Characteristics

This call is the same as ReadCur1 (SC 43), except that it returns the blinkrate and position (column and row) of the current window's text cursor. The input 'address' points to the byte array for the current shape.

### Errors

No errors are returned.

## 43 ReadCur1

Returns the position (column and row), and the blinkrate of the current window's graphics cursor.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | RR10 ← | address |
| Output: | R7 → | blinkrate |
| | R8 → | x position |
| | R9 → | y position |
| | R5 → | error status |

### Characteristics

This call is the same as ReadCur0 (SC 42), except that it returns the x,y position and blinkrate of the current windows graphics cursor. The input 'address' points to the byte array for the current shape.

### Errors

No errors are returned.

## 44 SelectCur

Selects the graphics or the text cursor, or turns off the current cursor.

### Input/Output Parameters

| | |
|---|---|
| Input: | R8 ⟵ select |
| Output: | there is no output |

### Characteristics

This routine chooses the state of the cursor for the current window, according to the value of the input "select" as follows:

| | |
|---|---|
| 0: | Turns off the cursor for the current window. (selecting another window will also turn off the cursor). |
| 1: | Selects and displays the graphics cursor in the current window. |
| 2: | Selects and displays the text cursor in the current window. |

Note that only one cursor can be displayed at a given time, regardless of the number of windows.

### Errors

No errors are returned.

## 45 GrfInit

Initializes the screen and sets defaults.

### Input/Output Parameters

| | |
|---|---|
| Input: | there are no inputs |
| Output: | R8 $\longrightarrow$ colour flag<br>RR10 $\longrightarrow$ pointer |

### Characteristics

This function must be called to initialize the screen. It sets the screen to contain one window (number 1), sets default global attributes for the screen, and default attributes for the window.

Default conditions are: one window for a full screen, green or white colour (depending upon hardware) on a black background and cursor off.

The outputs are a pointer and a colour flag. The latter is "0" for a black and white system, and "1" for a colour system. These values are determined by hardware jumpers.

The pointer is the address of a mailbox area (8 bytes), also used by the IEEE driver, and declared globally by PCOS. These 8 bytes (0-7) are used by the IEEE-488 and keyboard drivers. On calling GrfInit, the interpreter will be passed the address of this area in RR10.

### Errors

No errors are returned.

## 46 PaletteSet

Selects a global four colour set (only for four colour systems).

### Input/Output Parameters

|  |  |
|---|---|
| Input: | R8 $\longleftarrow$ colour A |
|  | R9 $\longleftarrow$ colour B |
|  | R10 $\longleftarrow$ colour C |
|  | R11 $\longleftarrow$ colour D |
| Output: | R5 $\longrightarrow$ error status |

### Characteristics

This call selects 4 colours out of a possible 8 for the global colour set. The four inputs are chosen from the following set:

|  |  |
|---|---|
| 0 | black |
| 1 | green |
| 2 | blue |
| 3 | cyan |
| 4 | red |
| 5 | yellow |
| 6 | magenta |
| 7 | white |

and a check is made that the inputs are in the range from 0 to 7, but no check is made for colour duplications.
The BASIC COLOUR statement is implemented by a call to this routine. Also, this routine is called by GrfInit to initialize to the default colours.

**Note:**

Note that this system call has no effect on black and white and eight colour systems.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 47 DefineWindow

Creates a new window.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | R8 ⟵ | quadrant |
| | R9 ⟵ | position |
| | R10 ⟵ | vertical spacing |
| | R12 ⟵ | horizontal spacing |
| Output: | R11 ⟶ | window number |
| | R5 ⟶ | error status |

### Characteristics

This routine is used to create a new window by splitting the current window into two parts. A unique window number is returned for the new window and the current window remains selected.

The input 'quadrant' indicates that part of the old window from which the new one is to be created. The choices are as follows:

| | |
|---|---|
| 0 | TOP PORTION |
| 1 | BOTTOM PORTION |
| 2 | LEFT PORTION |
| 3 | RIGHT PORTION |

The value and meaning of the input 'position' depends upon whether the split is done horizontally or vertically. If the split is to be on a horizontal line (quadrant = 0 or 1), 'position' is measured in scanlines, from the top of the current window. The allowable range is then:

(Vspace + 1) to (Height - Vspace);

where 'Vspace' is the text line spacing of the existing window. If the split is to be on a vertical line (quadrant = 2 or 3), 'position' is measured in the number of characters, counting from the left. The allowable range is then from 1 to width minus 1.

The input 'vertical spacing' is the number of scanlines between the tops of the characters in two consecutive text lines. It may be a number from 10 to 16.

The input 'horizontal spacing' is the number of pixels between the right edges of two consecutive characters. It can have a value of 6 or 8. If the values for vertical or horizontal spacing are omitted or entered as zero, their spacing defaults to the values for the parent window.

When a window is created, it will have the same foreground and background colours as its parent window (window 1 is always initialised with foreground and background colours of 1 and 0, respectively i.e. green and black in a colour system, and white and black in a monochrome system). The new window will have its text cursor placed at the top left of the window. The graphics cursor and graphics accumulator positions will be set at the center of the new window, with no cursor displayed.

The parent window's cursor and graphics accumulator positions will automatically be adjusted by the amount taken by the new window. The parent window remains selected.

In the graphics coordinate system supported by the PCOS, the lower left-hand corner of a window is the origin, with coordinates (0,0); the coordinates will be scanlines vertically and pixels (bits) horizontally. The origin of the text coordinate system is the upper left-hand character position of the window, with coordinates (1,1).

Calling DefineWindow with quadrant = 0 and position = 0 will have the effect of setting the spacing of the current window. If window 1 is the only window and its spacing is changed, then the display character size is changed from the current format to the other. If horizontal spacing is 6 then the system goes into 80x25 format. The size of the screen is reduced from 512 by 256 pixels to 480 by 256 (with 2-byte margins on the right and left). If horizontal spacing is given as 8, then the system goes into 64x16 mode, and the screen is expanded back to 512 by 256 pixels.

## Errors

An error condition leaves the returned window number equal to -1, and leaves a 36 (wind_create_err) in R5. If no errors, a zero (0) will be returned.

## 48 SelectWindow

Selects another window.

### Input/Output Parameters

| | |
|---|---|
| Input: | R8 $\longleftarrow$ window number |
| Output: | R5 $\longrightarrow$ error status |

### Characteristics

This routine is used to change the current window to another already existing window. The input "window number" is the number of the window (1 to 16) to be selected. Any screen output routines which have a window number as a parameter must call SelectWindow.

### Errors

If there are any errors, a status code is returned in R5. If there are no errors, a zero (0) will be returned. A value of 35 (wind_not_open_err) will be returned if the given window does not exist.

## 49 ReadWindow

Returns the attributes of the current window.

### Input/Output Parameters

| | |
|---|---|
| Input: | there are no inputs |
| Output: | R7 $\longrightarrow$ window number |
| | R8 $\longrightarrow$ x |
| | R9 $\longrightarrow$ y |
| | R10 $\longrightarrow$ foreground |
| | R11 $\longrightarrow$ background |
| | R5 $\longrightarrow$ error status |

### Characteristics

This routine returns the attributes of the current window. The outputs are:

| | | |
|---|---|---|
| 'window' | -- | current window identifier number |
| 'x' | -- | window width in bytes |
| 'y' | -- | window hight in pixels |
| 'foreground' | -- | foreground colour of current window |
| 'background' | -- | background colour of current window |

### Colour Attributes

The colour values returned will belong to one of the sets shown below.

The colour selection of four (A - D) is originally made from the eight listed under PaletteSet (SC 46):

| Monochrome | | Four-Colour Systems | | Eight-Colour Systems | |
|---|---|---|---|---|---|
| 0 | black | 0 | colour A | 0 | black |
| 1 | white | 1 | colour B | 1 | green |
| | | 2 | colour C | 2 | blue |
| | | 3 | colour D | 3 | cyan |
| | | | | 4 | red |
| | | | | 5 | yellow |
| | | | | 6 | magenta |
| | | | | 7 | white |

## Errors

No errors are returned.

## 50 ChgWindow

Changes window colours.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | R8 ⟵ | foreground |
| | R9 ⟵ | background |
| Output: | R5 ⟶ | error status |

### Characteristics

This routine changes the colour attributes for the current window. The inputs 'foreground' and 'background' are integers specifying the foreground and background colours respectively. They are chosen from those listed under 'Colour Attributes' (see below).

### Colour Attributes

The colour values selected must belong to one of the sets shown below. The colour selection of four (A - D) is originally made from the eight listed under PaletteSet (SC 46):

| Monochrome | Four-Colour Systems | Eight-Colour Systems |
|---|---|---|
| 0 black | 0 colour A | 0 black |
| 1 white | 1 colour B | 1 green |
| | 2 colour C | 2 blue |
| | 3 colour D | 3 cyan |
| | | 4 red |
| | | 5 yellow |
| | | 6 magenta |
| | | 7 white |

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 51 CloseWindow

Closes the selected window.

### Input/Output Parameters

| | |
|---|---|
| Input: | R8 ⟵ window numbers |
| Output: | there are no outputs |

### Characteristics

This routine is used to close an existing window.

The input 'window' is the window number. The area of the window is returned to the parent window, and the background colour is cleared to that of the parent window.

It should be noted that window 1 cannot be closed.

### Errors

No errors are returned.

## 52 ScaleXY

Checks the coordinates against window boundaries.

### Input/Output Parameters

| | |
|---|---|
| Input: | $R8 \leftarrow x$ <br> $R9 \leftarrow y$ |
| Output: | $R10 \rightarrow$ return_value |

### Characteristics

The inputs 'x' and 'y' are graphics coordinates.

The system call checks their values against the window size of the current window,and returns a true value in R10 if and only if the coordinates are within the boundaries of the window. The 'return' is 1 for true.

### Errors

No errors are returned.

## 53 MapXYC

Converts x-y to absolute coordinates and stores the result in the graphics accumulator.

### Input/Output Parameters

| | |
|---|---|
| Input: | R8 ←—— x<br>R9 ←—— y |
| Output: | there are no outputs |

### Characteristics

The inputs 'x' and 'y' are the specified screen coordinates.

The system call converts these coordinates to the absolute screen position (of C-type) for the current window, and stores the resulting value in the graphics accumulator.

Note:

The input values are not checked for being within range. ScaleXY should be called first.

### Errors

No errors are returned.

## 54 MapCXY

Converts the C-value in the graphics accumulator to x-y coordinates.

### Input/Output Parameters

| | |
|---|---|
| Input: | there are no inputs |
| Output: | R8 $\longrightarrow$ x<br>R9 $\longrightarrow$ y |

### Characteristics

This call converts the current value in the graphics accumulator to x-y coordinates for the current window.

If the value in the graphics accumulator is outside the current window, the results are undefined.

### Errors

There are no errors returned.

## 55 FetchC

Returns the contents of the graphics accumulator.

### Input/Output Parameters

| | |
|---|---|
| Input: | there are no inputs |
| Output: | RR8 $\longrightarrow$ C-value |

### Characteristics

This call saves the current value of the graphics accumulator for future use.

There are no input parameters. The output "C-value" is the contents of the 32-bit graphics accumulator.

### Errors

No errors are returned.

## 56 StoreC

Sets the graphics accumulator to a specified C-value.

### Input/Output Parameters

| | |
|---|---|
| Input: | RR8 ←—— C-value |
| Output: | there are no outputs |

### Characteristics

This sets the graphics accumulator to a specified C-value.

The structure of the C-value is described in chapter 7. If the C-value input is outside the current window, the results are undefined.

### Errors

No errors are returned.

## 57 UpC

Moves the position of the graphics accumulator up by one pixel.

### Input/Output Parameters

This call has no parameters

### Characteristics

This call moves the graphics accumulator up by one pixel position.

There is no checking with respect to window boundaries or the screen boundary; it is expected that the calling program will perform a check before executing a sequence of code using these routines.

### Errors

No errors are returned.

### Remarks

For the routine which does perform checks, see TUpC (SC 70).

## 58 DownC

Moves the position of the graphics accumulator down by one pixel.

### Input/Output Parameters

This call has no parameters

### Characteristics

This call move the graphics accumulator down by one pixel position.

There is no checking with respect to window boundaries or the screen boundary; it is expected that the calling program will perform a check before executing a sequence of code using these routines.

### Errors

No errors are returned.

### Remarks

For the routine which does perform checks, see TDownC (SC 69).

## 59 LeftC

Moves the position of the graphics accumulator left by one pixel.

### Input/Output Parameters

This call has no parameters

### Characteristics

This call move the graphics accumulator left by one pixel position.

There is no checking with respect to window boundaries or the screen boundary; it is expected that the calling program will perform a check before executing a sequence of code using these routines.

### Errors

No errors are returned.

### Remarks

For the routine which does perform checks, see ScaleXY (SC 52).

## 60 RightC

Moves the position of the graphics accumulator right by one pixel.

### Input/Output Parameters

This call has no parameters

### Characteristics

This call move the graphics accumulator right by one position.

There is no checking with respect to window boundaries or the screen boundary; it is expected that the calling program will perform a check before executing a sequence of code using these routines.

### Errors

No errors are returned.

### Remarks

For the routine which does perform checks, see ScaleXY (SC 52).

## 61 SetAtr

Sets the current colour attribute.

### Input/Output Parameters

| | |
|---|---|
| Input: | R8 $\longleftarrow$ colour |
| Output: | R5 $\longrightarrow$ error status |

### Characteristics

The input "colour" is the desired current attribute, or brush colour. This call sets the current attribute to that colour.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 62 SetC

Plots a single point.

### Input/Output Parameters

| | |
|---|---|
| Input: | R8 ←—— operation |
| Output: | there are no outputs |

### Characteristics

This system call plots a single point. If the input 'operation' is equal to 0, a point having the current colour attribute is plotted at the position specified by the graphics accumulator.

For other values of 'operation', logical operations are performed (see table below). These are between the current attribute and the attribute of the pixel at the specified point; the result is then stored for the specified location.

| | | |
|---|---|---|
| 0 | PSET | The current attribute is stored. |
| 1 | XOR | The current attribute is XORed with the pixel. |
| 2 | AND | The current attribute is ANDed with the pixel. |
| 3 | NOT | The complement of the pixel is stored. |
| 4 | OR | The current attribute is ORed with the pixel. |
| 5 | PRESET | The current background colour is stored. |

For example, the XOR function with a current attribute of 1 for monochrome or 3 for colour can be used for plotting a temporary point or line on the screen; repeating the function will then restore the screen to its original state.

### Errors

No errors are returned.

## 63 ReadC

Returns the colour attribute of the current point.

### Input/Output Parameters

| | |
|---|---|
| Input: | there are no inputs |
| Output: | R8 $\longrightarrow$ colour |

### Characteristics

This routine returns the attribute of the current point ("colour") as an integer (0..7) for eight colour systems, (0..3) for four colour systems, or (0..1) for monochrome, and stores it in register R8.

### Colour Attributes

The colour values returned will belong to one of the sets shown below. The colour selection of four (A - D) is made from the eight listed under PaletteSet (SC 46):

| Monochrome | Four-Colour Systems | Eight-Colour Systems |
|---|---|---|
| 0 black | 0 colour A | 0 black |
| 1 white | 1 colour B | 1 green |
| | 2 colour C | 2 blue |
| | 3 colour D | 3 cyan |
| | | 4 red |
| | | 5 yellow |
| | | 6 magenta |
| | | 7 white |

### Errors

No errors are returned.

## 64 NSetCX

Draws a horizontal line.

### Input/Output Parameters

| | |
|---|---|
| Input: | R8 ⟵ count |
| | R9 ⟵ operation |
| Output: | there are no outputs |

### Characteristics

This call draws a horizontal line. The inputs are "count" (the number of points to be plotted) and "operation" which has the same meaning as used in SetC (62).

This call is the same as calling SetC (62) and RightC (60) 'count' times, but it has been optimized for speed.

### Errors

No error checking is done. It is assumed that range checking is done by the caller.

## 65 NSetCY

Draw a vertical line.

### Input/Output Parameters

| | |
|---|---|
| Input: | R8 ←—— count<br>R9 ←—— operation |
| Output: | there are no outputs |

### Characteristics

This call draws a vertical line. The inputs are "count" (the number of points to be plotted) and "operation" which has the same meaning as used in SetC (62).

Using this call is the same as calling SetC (62) and DownC (58) 'count' times, but it has been optimized for speed.

### Errors

No error checking is done. It is assumed that range checking is done by the caller.

## 66 NRead

Reads a screen rectangle into an array.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | R8 ⟵ | width (in pixels) |
| | R9 ⟵ | height (in pixels) |
| | RR10 ⟵ | pointer to byte array |
| Output: | @RR10 ⟶ | address of byte array |
| | R5 ⟶ | always cleared (no error conditions) |

### Characteristics

This call reads a screen rectangle into an array in memory.

The size (in pixels) of a rectangle on the screen is specified by the first two coordinates. The position of the upper left-hand corner of the rectangle is determined by the current Graphics Accumulator (which can be set using system call 53 MapXYC).

The third parameter is a pointer to a byte array which consists of a 6-byte header followed by an array of two-byte entries, each of which is a sixteen-bit integer.

The byte array is structured as follows:

| byte | contents |
|---|---|
| 0 | width (high byte) |
| 1 | " (low byte) |
| 2 | hight (high byte) |
| 3 | " (low byte) |
| 4 | colour flag (high byte - always 0) |
| 5 | " " (low byte) |
| 6 | picture data |
| . | . |
| . | . |
| . | . |
| n | picture data |

The colour flag is equal to 0 for a monochrome system, 1 for a 4-colour system and 2 for an 8-colour system.

If the width of the rectangle is W pixels, each scanline of the rectangle (for each colour plane) is stored in INT((W+15)/16) two-byte integer entries in the byte array, with the bit array left-justified in the integer array, so that the last two-byte entry for each scanline may have up to fifteen undefined bits.

The screen data is stored starting from top to bottom, with data for various colour memory planes interleaved scanline by scanline.
In other words, the integer array for the top scanline, plane 0 is stored first, followed in succession by the integer arrays for screen memory planes 1 and 2, if they exist on the system; these are followed in turn by the data for successive scanlines.

### Errors

The caller is assumed to have done error checking.

## 67 NWrite

Transfers a graphics rectangle from an array to the screen.

### Input/Output Parameters

```
Input:          R7   <------ logical function
                R8   <------ maximum width of rectangle in pixels
                R9   <------ maximum height of rectangle in scanlines
                RR10 <------ pointer to a byte array

Output:         R5   ------> always cleared (no error condition)
```

### Characteristics

This system call is used for inserting screen data, previously read from the screen using the NRead system call, somewhere on the screen.

Values of logical function for NWrite system calls:

| | |
|---|---|
| 0 | overwrite what is already there |
| 1 | XOR (exclusive OR) array contents with destination |
| 2 | AND array contents with destination |
| 3 | COM: complement destination, no copy |
| 4 | OR array contents with destination |
| 5 | INVERT: complement text, copy |

The logical function is useful in a variety of situations. For example, XOR may be used to display an object which can be erased with another XOR, leaving the screen as it was before the first XOR. AND may be used to selectively erase parts of the screen to colour 0, using a specially constructed array. OR may be used similarily to erase parts of the screen to all white.

The height and width parameters are used to determine what proportion of the rectangle saved in the array is actually written onto the screen; this has dimensions which are the minima of the parameters and the height and width values saved in the array; the rectangle written includes the upper left-hand corner of the saved rectangle in all cases.

As with NRead, the upper left-hand corner of the rectangle is determined by the current Graphics Accumulator.

Compatibility exists between colour and monochrome systems in the following sense:
if screen data is read with NRead on a monochrome system, and written with NWrite on a colour system, the data is written only into screen plane 0; screen plane 1 (or 2 for the 8-colour system) is left unchanged. On the other hand, if screen data is read on a colour system and written from the same array on a monochrome system, only data for colour memory plane 0 is written on the monochrome system.

## Errors

The caller is assumed to have done error checking.

## 68 PntInit

Specifies the global colour attributes for paint routines.

### Input/Output Parameters

Input: R8 $\longleftarrow$ paint colour
R9 $\longleftarrow$ border colour

Output: R5 $\longrightarrow$ error status

### Characteristics

The inputs 'paint' and 'border' must be legal screen colours as shown below. The colour selection of four (A - D) is made from the eight listed under PaletteSet (SC 46):

| Monochrome | Four-Colour Systems | Eight-Colour Systems |
|---|---|---|
| 0 black | 0 colour A | 0 black |
| 1 white | 1 colour B | 1 green |
| | 2 colour C | 2 blue |
| | 3 colour D | 3 cyan |
| | | 4 red |
| | | 5 yellow |
| | | 6 magenta |
| | | 7 white |

The attributes set are globals, like the main screen attribute, not window attributes.

This routine must be called before doing "ScanL" (SC 71) or "ScanR" (SC 72) or they will be undefined. ( Usually, both paint colour and border colour are 1).

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 69 TDownC

Moves the graphics accumulator down by one pixel after checking the window boundary.

### Input/Output Parameters

| | |
|---|---|
| Input: | there are no inputs |
| Output: | R8 $\longrightarrow$ check value |

### Characteristics

This has the same effect as DownC, except that the position of the graphics accumulator is checked against the lower boundary position of the current window before it is changed.

If the new position is out of bounds, a false 'check value' is returned in R8 and the graphics accumulator is unchanged. If the new position is within bounds, the position is moved down one pixel and a true value is returned.

### Errors

No errors are returned.

## 70 TUpC

Moves the graphics accumulator up by one pixel after checking the window boundary.

### Input/Output Parameters

| | |
|---|---|
| Input: | there are no inputs |
| Output: | R8 $\longrightarrow$ check value |

### Characteristics

This has the same effect as UpC, except that the position of the graphics accumulator is checked against the lower boundary position of the current window before it is changed.

If the new position is out of bounds , a false 'check value' is returned in R8 and the graphics accumulator is unchanged. If the new position is within bounds, the position is moved up one pixel and a true value is returned.

### Errors

No errors are returned.

## 71 ScanL

Paints left on a scanline up to a border.

### Input/Output Parameters

| | |
|---|---|
| Input: | there are no inputs |
| Output: | $R9 \longrightarrow$ count-1 |
| | $R10 \longrightarrow$ margin flag |
| | $R11 \longrightarrow$ painted flag |

### Characteristics

The purpose of this routine is to paint part of an enclosed region in the current window, moving left along a scanline.

All points starting at the initial position of the graphics accumulator are painted to the paintcolour. If any points painted were not already painted, the 'painted flag' is set.

The routine stops when the border colour has been reached or when the left margin of the window has been reached. The 'margin flag' is set if the left margin has been reached.

The output called 'count-1' is the number of pixels scanned (painted), regardless of whether their original colour was the paintcolour.

The graphics accumulator position is left at the end of the scan.

### Errors

No errors are returned.

## 72 ScanR

Paints right on a scan line up to a border.

### Input/Output Parameters

| | | | |
|---|---|---|---|
| Input: | R8 | $\longleftarrow$ | maxcount |
| Output: | RR6 | $\longrightarrow$ | C-type |
| | R8 | $\longrightarrow$ | maxcount |
| | R9 | $\longrightarrow$ | count-r |
| | R10 | $\longrightarrow$ | margin flag |
| | R11 | $\longrightarrow$ | painted flag |

### Characteristics

The purpose of this routine is to paint part of an enclosed region in the current window, moving right along a scanline. At first the routine skips over a maximum of 'maxcount' points of the border colour.

If more than 'maxcount' border points are skipped, then ScanR stops immediately and returns R8 = 0 and R9 = 0 (and RR6 undefined).

All points following the initial border region are then painted to the paintcolour. If any points painted were not already painted, the 'painted flag' is set.

The routine stops when the border colour has been reached or when the right margin of the window has been reached. The 'margin flag' is set if the right margin has been reached. The output called 'count-r' value is the length in pixels of the painted segment.

The output 'C-type' points to the position of the first pixel painted. The graphics accumulator position is left at the end of the scan.

### Errors

No errors are returned.

## 73 SetTime

Sets the system clock.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | RR8 ← address | |
| | R10 ← length | |
| Output: | R5 → error status | |

### Characteristics

The input 'address' points to an address in the caller's data area which contains the time of day. The input 'length' gives the length of the ASCII string. The format of the data in the string must be:

```
hh:mm:ss
```

where 'hh' is the hour (in 24-hour time), 'mm' is minutes, and 'ss' is seconds. Leading zeros need not be supplied. Any non-numeric character can be selected for delimiter as shown in examples below, using the PCOS SSYS (set system) command.

```
ss 04/15/82,13:12:45

ss "04 15 82",08:10:00
```

Time is initialized to 00:00:00 at system startup. If blanks are selected for delimiters, as in the second example, the expression must be put in quotes.

### Errors

The value returned in R5 is zero if the clock was correctly set.

## 74 SetDate

Sets the system date-clock.

### Input/Output Parameters

| | |
|---|---|
| Input: | RR8 ← address |
| | R10 ← length |
| Output: | R5 → error status |

### Characteristics

The input 'address' points to an address in the caller's data area which contains the date. The input 'length' gives the length of the ASCII string.

The format of the data in the string, except for the delimiter, must be:

```
dd:mm:yyyy
```

where 'dd' is the day, 'mm' is the month, and 'yyyy' is the year; leading zeroes need not be supplied.

Any non-numeric character may be used in place of the colon, as shown in the examples for SetTime (73).

The date is initialized to January 1, 1982 at system startup. If only two digits are input for the year, the century is assumed to be 19.

### Errors

The value returned in R5 is zero if and only if the date was correctly set.

## 75 GetTime

Returns the system time.

### Input/Output Parameters

Input: RR8 $\longleftarrow$ address
R10 $\longleftarrow$ length

Output: R5 $\longrightarrow$ error status

### Characteristics

This call returns the ASCII string giving the system time. The two inputs are the address and maximum length of the string, which is stored in the BASIC data area.

The format of the time returned is:

hh:mm:ss

where 'hh' is the hour (in 24-hour time), 'mm' is the minutes, and 'ss' is the seconds.

There will be leading zeroes to make each field 2 characters in length, and the character separating the various fields for the time will be that used in the last call to 'SetTime'. The system initializes the separator character to ':'.

### Errors

If there are any errors, a non-zero value is returned in R5; a zero is returned if there were no errors.

## 76 GetDate

Returns the system date.

### Input/Output Parameters

```
Input:          RR8 <------ address
                R10 <------ length

Output:         R5  ------> error status
```

### Characteristics

This call returns the ASCII string giving the system date. The two inputs are the address and maximum length of the string, which is stored in the BASIC data area.

The format of the returned date is:

```
                dd:mm:yyyy
```

where 'dd' is the day, 'mm' is the month, and 'yyyy' is the year.

There will be leading zeroes to make each field two characters in length and the character separating the various fields for the date will be that used in the last call to 'SetDate'. The system initializes the separator character to ':'.

### Errors

If there are any errors, a non-zero value is returned in R5; a zero is returned if there were no errors.

## 77 CallUser

Calls user or PCOS utility or command.

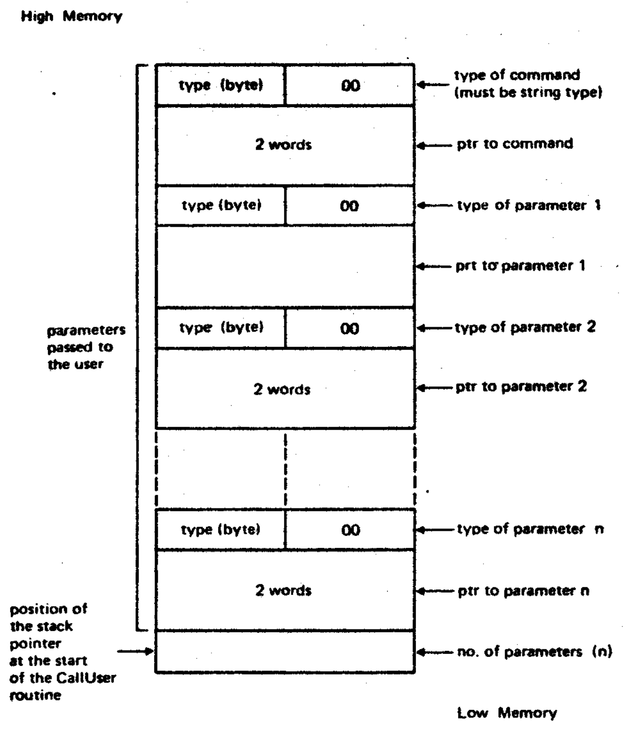
### Input/Output Parameters

Input: RR14 ← pointer

Output: R5 → error status

### Characteristics

This SC allows the Assembler programmer to invoke from his programs the PCOS utilities and other utilities resident on disk or in memory. Before invoking the SC 77 the user must prepare his parameters in the stack in the following way:



As far as the "types" are concerned, the same rules apply as previously stated in chapter 2 in the section which deals which the PCOS standard. In this case however the command parameters will have to be obtained using a series of "push"es rather than a series of "pop"s. The parameter pointers will be of the Z-8001 format.

The following table illustrates schematically the types already dealt with in chapter 2.

Data Types

| Category | Data Type | Pointer Value | Description |
|---|---|---|---|
| null | 0 | %0000FFFF | for null parameters |
| integer | 2 | segmented ptr | integers occupy one word; |
| string | 3 | segmented ptr | pointer ptr to a 3-byte descriptor: 1-byte length & 2-byte unsegmented ptr to actual string |

The following is an example of an Assembler source file, which, (by means of SC 77) makes use of the PCOS utility "filenew", which allocates a certain number of blocks on disk under the name of a given file.

In practice, it is a question of invoking from an assembler utility, that which can be invoked from PCOS in the following way:

```
fn FILE,100
```

The following is a sequence of Assembler instructions to be used for preparing the stack before the SC 77.

```
                                   .
                                   .
                                   .
                              push    @rr14,#%0300     type 3(string)
                              lda     rr2,cmd
                              ld      ptrcmd+2,r3      store offset
                              lda     rr2,ptrcmd+1
                              pushl   @rr14,rr2
                              push    @rr14,#%0300     type 3(string)
                              lda     rr2,filenam
                              ld      ptr1+2,r3        store offset
                              lda     rr2,ptr1+1
                              pushl   @rr14,rrr
                              push    @rr14,#%0200     type 2(integer)
                              lda     rr2,nblock
                              pushl   @rr14,rr2
                              push    @rr14,#2         no. of parameters
                              sc      #77
                                   .
                                   .
                                   .
   cmd                        ddb     "fn"
   ptrcmd                     dd      0002,0000
   filenam                    ddb     "FILE"
   ptr1                       dd      0004,0000
   nblock                     dd      0100             no. of blocks
                                   .
                                   .
                                   .
```

Errors

If there are any errors, the status code is returned in R5. If there are no errors a zero will be returned.

## 78 IBSrQ0

Disables the service request (SRQ) interrupt.

**Input/Output Parameters**

| | |
|---|---|
| Input: | there are no input parameters |
| Output: | R5 $\longrightarrow$ error status |

**Characteristics**

The statement "ON SRQ GOSUB 0" will cause the system call IBSrQ0 to be executed; this system call will disable the SRQ interrupt (for further details on the interrupt system, see SC 79).

**Errors**

If the system does not have an IEEE option board, R5 will contain a Hex 0A. If there are no errors, a zero (0) will be returned.

## 79 IBSrQ1

Enables the service request (SRQ) interrupt.

### Input/Output Parameters

| | |
|---|---|
| Input: | there are no input parameters |
| Output: | R5 $\longrightarrow$ error status |

### Characteristics

The statement "ON SRQ GOSUB <line number>" will cause the system call IBSrQ1 to be executed; this system call enables the SRQ interrupt.

The IEEE-488 interrupt service routine will set the global flag "srq_488" (byte) to 1 when an SRQ interrupt occurs. (This flag is stored in the mailbox area).

This flag will be tested by the interpreter before the execution of each source statement following the ON SRQ GOSUB. If set, it will be reset by the interpreter, and the subroutine entered (see call GrfInit (SC 45)).

### Errors

If the system does not have an IEEE option board, R5 will contain a Hex 0A. If there are no errors, a zero (0) will be returned.

## 80 IBPoll

Polls a specified device on an instrument bus.

### Input/Output Parameters

|  |  |
|---|---|
| Input: | R8 $\longleftarrow$ talker addr |
| Output: | RR10 $\longrightarrow$ ptr to status |
|  | R5 $\longrightarrow$ error status |

### Characteristics

This call polls the device specified, within a serial service request poll. The input 'talker addr' identifies the device.

The call tests the device address, reads the device status byte, and saves it in an address pointed to by'ptr to status'.

### Errors

If the system does not have an IEEE option board, R5 will contain a Hex 0A. If the talker address is invalid (ie., greater than 001E), R5 will contain '09'. If there are no errors, a zero (0) will be returned.

## 81 IBISet

Causes a remote enable (REN) or an interface clear (IFC) to be sent.

### Input/Output Parameters

Input: R8 $\longleftarrow$ operand

Output: R5 $\longrightarrow$ error status

### Characteristics

This call causes the remote enable (REN) message or the interface clear (IFC) pulse to be transmitted, depending upon the value of the input 'operand'.

If '0' is loaded into R8, then the REN message is sent true; if '1' is loaded, then the IFC pulse is sent.

### Errors

If the system does not have an IEEE option board, R5 will contain a Hex 0A. If there are no errors, a zero (0) will be returned.

## 82 IBRSet

Causes the remote enable (REN) message to be sent false.

### Input/Output Parameters

| | |
|---|---|
| Input: | there are no parameters |
| Output: | R5 $\longrightarrow$ error status |

### Characteristics

This call causes the remote enable (REN) message to be sent false.

### Errors

If the system does not have an IEEE option board, R5 will contain a Hex 0A. If there are no errors, a zero (0) will be returned.

## 83 IBPrnt

Checks the address and then causes output of data bytes.

### Input/Output Parameters

Input: RR6 $\longleftarrow$ buffer addr
R8 $\longleftarrow$ listener addr
R9 $\longleftarrow$ buffer len, in bytes
R10 $\longleftarrow$ delimiter

Output: R5 $\longrightarrow$ error status

### Characteristics

Before calling the driver, the BASIC interpreter will transfer the output bytes to a buffer, from which they will be sequentially transferred by the driver.

This call will test the listener address in R8; if less than 001F, writes listener address, if specified.

R10 contains zero if the "@" option (END as data-stream delimiter) is specified, and 1 if it is not (CR, END as data-stream delimiter sequence). If there are any output bytes for transfer, writes them to bus, with ATN false.

### Errors

If the system does not have an IEEE option board, R5 will contain a Hex 0A. If the listener address in R8 is greater than 001F, this call returns an error code of 09. If there are no errors, a zero (0) will be returned.

## 84 IBWByt

Outputs commands (optional) and writes data bytes (optional).

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | RR6 $\longleftarrow$ | numval addr |
| | R8 $\longleftarrow$ | comlist length |
| | R9 $\longleftarrow$ | numval length |
| | RR10 $\longleftarrow$ | comlist addr |
| Output: | R5 $\longrightarrow$ | error status |

### Characteristics

If there is a command list, asserts ATN and outputs commands. If there are any data bytes to be output, writes them to bus with ATN false.

The input 'comlist addr' points to the address of the command list. This list, if present, is stored as a sequence of bytes, 2 to the word. The input 'comlist length' is the command list length in 15 low-order bits; high-order bit: 1 if "@" option (END sent with last byte of data as statement delimiter) specified, 0 if not (END with CR terminates data).

The input 'numval addr' points to the address of the list of numeric values . It, too, is stored as a sequence of bytes, 2 to a word. The input 'numval length' is 0 if not specified.

### Errors

If the system does not have an IEEE option board, R5 will contain a Hex OA. If there are no errors, a zero (0) will be returned.

## 85 IBInpt

Places bytes received, into a buffer.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | R7 $\longleftarrow$ | buffer length |
| | R8 $\longleftarrow$ | talker addr |
| | R9 $\longleftarrow$ | listener addr |
| | RR10 $\longleftarrow$ | buffer addr |
| Output: | R5 $\longrightarrow$ | error status |
| | R7 $\longrightarrow$ | number of bytes not read |

### Characteristics

This procedure calls IBLinpt. Both IBInpt and IBLinpt place bytes received sequentially from a driver into a single buffer. They differ in that, for IBInpt, the BASIC interpreter transfers the buffer contents to the variables in the variable list provided by the user; for IBLinpt the user specifies the buffer for a single line of data.

On entry, the 'buffer length' (R7) is given in bytes; on exit, this represents the number of bytes not read (buffer length minus number of bytes read). The 'buffer addr' points to the buffer which will receive the data bytes. The 'talker addr' (R8) and 'listener addr' (R9) will both be 001F if not specified.

### Errors

The error codes which can be returned in R5 are:

| ERROR CODE | MEANING |
|---|---|
| 03 | Invalid termination of input bytestream. The two valid cases are:<br>- the number of data bytes received equals the value provided in R7 (string variable length, in bytes). The last data byte is accompanied by the END condition (EOI true, ATN false).<br>- the number of data bytes received equals the value provided in R7 (string variable length, in bytes). The last data byte is followed by a CR, LF pair with the END condition accompanying the LF. |
| 09 | Talker or Listener address greater than 1F. |
| 0A | IEEE board not present. |
| 0B | 15 second polling loop ( for 'byte in', 'byte out' , or 'input buffer empty' condition) timed out; handshake could not be completed within 15 seconds. |

If there are no errors, a zero (0) will be returned.

## 86 IBLinpt

Places bytes received into a buffer as a single line of data.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | R7 $\longleftarrow$ | buffer length |
| | R8 $\longleftarrow$ | talker addr |
| | R9 $\longleftarrow$ | listener addr |
| | RR10 $\longleftarrow$ | buffer addr |
| Output: | R5 $\longrightarrow$ | error status |
| | R7 $\longrightarrow$ | number of bytes not read |

### Characteristics

Both IBLinpt and IBInpt place bytes received sequentially from a driver into a single buffer. They differ in that, for IBLinpt the user specifies the buffer for a single line of data; for IBInpt, the BASIC interpreter transfers the buffer contents to the variables in the variable list provided by the user.

On entry, the 'buffer length' (R7) is given in bytes; on exit, this represents the number of bytes not read (buffer length minus number of bytes not read). The 'buffer addr' points to the buffer which will receive the data bytes. The 'talker addr' (R8) and 'listener addr' (R9) will both be 001F if not specified.

### Errors

The error codes which can be returned in R5 are:

| ERROR CODE | MEANING |
|---|---|
| 03 | Invalid termination of input bytestream. The two valid cases are:<br>- the number of data bytes received equals the value provided in R7 (string variable length, in bytes). The last data byte is accompanied by the END condition (EOI true, ATN false).<br>- the number of data bytes received equals the value provided in R7 (string variable length, in bytes). The last data byte is followed by a CR, LF pair with the END condition accompanying the LF. |
| 09 | Talker or Listener address greater than 1F. |
| 0A | IEEE board not present. |
| 0B | 15 second polling loop ( for 'byte in', 'byte out' , or 'input buffer empty' condition) timed out; handshake could not be completed within 15 seconds. |

If there are no errors, a zero (0) will be returned.

## 87 IBRByt

Outputs commands (optional) and reads data bytes (optional).

### Input/Output Parameters

```
Input:          RR6  <-----buffer addr
                R8   <-----comlist length
                R9   <-----buffer len, in bytes
                RR10 <-----comlist addr

Output:         R5   ----->error status
```

### Characteristics

If there is a command list, asserts ATN and outputs commands. It then reads the assigned number of bytes, and places them sequentially in a buffer.

The input 'comlist addr' points to the address of the command list. This list, if present, is stored as a sequence of bytes, 2 to the word.

The input 'comlist length' is the command list length in 15 low-order bits; high-order bit is always zero (0).

The input 'buffer addr' points to the buffer which will receive the data bytes. The input 'buffer len, in bytes' indicates the number of bytes to be read.

### Errors

If the system does not have an IEEE option board, R5 will contain a Hex 0A. If any handshake is not completed within 15 seconds, R5 will contain Hex '000B'. If there are no errors, a zero (0) will be returned.

## 88 Error

Displays standard error message.

### Input/Output Parameters

| | |
|---|---|
| Input: | RH5 $\longleftarrow$ parameter number<br>RL5 $\longleftarrow$ error code |
| Output: | there are no outputs |

### Characteristics

This procedure is only called if there are errors. The routine displays the message 'Error nn' in parameter xx' where nn is one of the standard error codes and xx is the parameter number passed in RH5. If xx is 00 then only the message 'Error nn' is displayed.

**Note:**

If the EPRINT command is resident, then an error message will be displayed.

## 89 DString

Displays a string message.

### Input/Output Parameters

| | |
|---|---|
| Input: | RR12 $\longleftarrow$ address |
| Output: | R5 $\longrightarrow$ error status |

### Characteristics

This routine displays a string message. The string must be terminated with a null (0) byte.

The message may include any number of carriage returns, but note that a linefeed will be automatically displayed after each carriage return in the string.

The input 'address' is the address of the string.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 90 CrLf

Does a CR and LF.

### Input/Output Parameters

| | |
|---|---|
| Input: | there are no parameters |
| Output: | R5 $\longrightarrow$ error status |

### Characteristics

This routine will do a carriage return and a line feed. There are no parameters.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 91 DHexByte

Displays a byte in Hex.

### Input/Output Parameters

Input: R12 $\longleftarrow$ byte

Output: R5 $\longrightarrow$ error status

### Characteristics

The byte supplied in the lower half of R12 is displayed as two hex digits.

### Errors

If there are any errors, the status code is returned in R5. If There are no errors, a zero (0) will be returned.

## 92 DHex

Displays a word in hex.

### Input/Output Parameters

|  |  |
|---|---|
| Input: | $R12 \leftarrow$ word |
| Output: | $R5 \rightarrow$ error status |

### Characteristics

This routine displays the 16-bit number in R12 as four hex digits.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 93 DHexLong

Displays a long word in hexadecimal.

### Input/Output Parameters

| | |
|---|---|
| Input: | RR12 $\longleftarrow$ long word |
| Output: | R5 $\longrightarrow$ error status |

### Characteristics

The long word supplied in RR12 is displayed as eight hex digits.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 94 DNumW

Displays a number as an unsigned decimal integer.

### Input/Output Parameters

|  |  |
|---|---|
| Input: | R12 $\longleftarrow$ integer |
|  | R13 $\longleftarrow$ field width |
| Output: | R5 $\longrightarrow$ error status |

### Characteristics

The number in R12 is displayed as an unsigned decimal integer. R13 specifies the field width for display.

The display is right-justified in the field, with leading zeroes changed to spaces.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 95 DLong

Displays a number as an unsigned decimal integer.

### Input/Output Parameters

|  |  |
|---|---|
| Input: | RR12 $\longleftarrow$ long integer |
| Output: | R5 $\longrightarrow$ error status |

### Characteristics

The number supplied in RR12 is displayed as an unsigned decimal integer, left-justified with leading zeroes omitted.

### Errors

If there are any errors, the status code is returned in R. If there are no errors, a zero (0) will be returned.

## 96 DisectName

Parses a file or a volume name.

### Input/Output Parameters

| | | | |
|---|---|---|---|
| Input: | R9 | ← | string length |
| | RR10 | ← | string address |
| | RR12 | ← | names record address |
| Output: | @RR12 | → | names record |
| | R7 | → | volume number |
| | R5 | → | error status |

### Characteristics

This call takes a file identifier of the form

"<volname>'/'<volpswd>':'<filename>'/'<filepswd>"

and parses it into its various components. A drive unit is acceptable as <volname>.

Each component is placed into the appropriate compartment of the names record as follows:

| | | |
|---|---|---|
| volname | : | 14 byte array |
| volpswd | : | 14 byte array |
| filename | : | 14 byte array |
| filepswd | : | 14 byte array |

The input string length is the length of the file identifier string (this includes the volume identifier), which in turn is input in the address specified in RR10.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 97 CheckVolume

Forces a check of disk volumes

### Input/Output Parameters

| | |
|---|---|
| Input: | there are no parameters |
| Output: | R5 $\longrightarrow$ error status |

### Characteristics

There are no input registers for this call. All volumes are forced to read their verification codes on their access.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 98 Search

Searches on a specified disk for a file name supplied by the user.

### Input/Output Parameters

Input:

R6 ⟵ drive  
R7 ⟵ search mode  
R9 ⟵ length  
RR10 ⟵ file pointer  
RR12 ⟵ name pointer

Output:

R9 ⟶ length of output filename  
RR10 ⟶ file pointer  
RR12 ⟶ modified  
R5 ⟶ error status

### Characteristics

This call searches on a disk for a file name supplied by the user. The file name may contain wild card characters.

The input called 'drive' identifies the drive to be searched (input a '-1' for the current drive). The input 'search mode' is either a '1' for a search from the beginning, or a '0' for a search from the last file found. The input 'length' is the length of the file name, in bytes. To search for any file, input a zero length.

The input 'file pointer' points to the memory location where the name of the file, if found, will be written. The input 'name pointer' is the address where the input string will be stored. If the file is found, the address of the name of the file is returned in RR10. The content of the register RR12 is modified by the Operating System.

### Errors

If there are any errors, the status code is returned in R5. If the file is found, a zero (0) will be returned.

## 99 MaxSize

Returns maximum free heap size

### Input/Output Parameters

| | |
|---|---|
| Input: | there are no parameters |
| Output: | R8 $\longrightarrow$ size |
| | R5 $\longrightarrow$ error status |

### Characteristics

This call returns the size of the largest free heap block in the current segment. Size is returned in bytes.

This call operates in segment 2 unless the program has done a Brand-NewAbsolute system call (121), in which case the segment number is that specified in the most recent "BrandNewAbsolute".

A simple way to change the segment number for a program is to do a SC 121 "BrandNewAbsolute" with a block length of 0.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 102 SetVol

Sets the active volume for the next access.

### Input/Output Parameters

| | |
|---|---|
| Input: | R7 $\longleftarrow$ vol number |
| Output: | R5 $\longrightarrow$ error status |

### Characteristics

This call sets the volume for the next access. The input 'vol number' is the volume number to be used for the next access.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 104 NewAbsolute

Allocates a block at a specified address.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | RR8 ⟵ | address of block pointer |
| | R10 ⟵ | length |
| | @RR8 ⟵ | block pointer |
| Output: | R5 ⟶ | error status |

### Characteristics

This call is similar to NewSameSegment (SC 33) except that the block allocated will be at a specified address. The input address (RR8) should be the address of a long (4-byte) memory location; this is where the desired address is stored. The input to R10 is the number of bytes requested, and must be even.

On exit from this call, the memory location that RR8 points to will contain a 32-bit address of the actual block allocated. If the requested value is too close to the end of a previous block, the actual value may be two bytes lower than the requested value, but will still include the requested length. If the space requested is not available, a nil-pointer (hex FFFFFFFF) will be returned in the memory location that RR8 points to, but no error will be returned in R5.

It is important to remember that RR8 does not contain the memory block address specified.

This call allocates blocks in the "SameSegment". This is segment 2 unless the program has done a "BrandNewAbsolute" system call, in which case the segment number is that specified in the most recent "Brand-NewAbsolute". A simple way to change the segment number for a program is to do a SC 121 "BrandNewAbsolute" with a block length of 0.
This call is a subset of system call 121 "BrandNewAbsolute". It has been maintained for compatibility with preceding releases.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 105 StringLen

Returns the length of the input string.

### Input/Output Parameters

| | |
|---|---|
| Input: | RR12 $\longleftarrow$ pointer |
| Output: | R7 $\longrightarrow$ length |
| | R5 $\longrightarrow$ error status |

### Characteristics

This call returns the length of the input string. The input 'pointer' points to the string; the output in R7 is the length read (until a null encountered, or 14, if no null in that length).

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 106 DiskFree

Returns the number of free sectors on the disk.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | R7 $\leftarrow$ | volume number |
| Output: | RR10 $\rightarrow$ | num of sectors |
| | R5 $\rightarrow$ | error status |

### Characteristics

This call returns the number of sectors that are available for use on the disk. The input 'volume number' is the volume to be checked (enter -1 for the current volume).

The number of sectors that are free on the volume will be returned in RR10.

### Errors

If there are any errors, the status code is re turned in R5. If there are no errors, a zero (0) will be returned.

## 107 BootSystem

Reboots (initializes) the system.

### Input/Output Parameters

| | |
|---|---|
| Input: | this call has no parameters |
| Output: | R5 $\longrightarrow$ error status |

### Characteristics

This system call can be used to reboot the system, exactly as does pressing the blue shift plus reset keys. In other words, the system reboots, but bypasses the diagnostic checks.

### Errors

There are no error checks with this call. If any errors occur, a status code is returned in R5. If there are no errors, a zero (0) will be returned.

## 108 SetSysSeg

Returns the caller to segmented system mode.

### Input/Output Parameters

Input: this call has no parameters

Output: R5 $\longrightarrow$ error status

### Characteristics

This call will return the caller to the segmented system mode, regardless of which mode the system is in.

### Errors

There are no error checks with this call. If any errors occur, a status code will be returned in R5. If there are no errors, a zero (0) will be returned.

**109 SearchDevTab**

Searches the system device table.

**Input/Output Parameters**

| | | |
|---|---|---|
| Input: | RR10 ⟵ | ptr to device name |
| | R9 ⟵ | device name length |
| Output: | | |
| | RL5 ⟶ | entry number |
| | RH5 ⟶ | device type |
| | RR8 ⟶ | ptr table entry |
| | R5 ⟶ | error status |

This command searches the system device table for the device named. The input 'ptr to device name' is the address where the first ASCII character of the name is stored; the input 'device name length' is the number of bytes in the name. If the call finds the device name, it returns the entry number of the device in RL5 and the device type in RH5 (1 = Read, 2 = Write, 3 = Read/Write); it also returns a pointer to the first entry in the particular device table in RR8.

EXAMPLE:

```
                         .
                         .
   table_pointer  DSL    1
   device_name    DDB    "cons"
                         .
                         .
                  ld     r9,#4                search_devtab string length
                  lda    rr10,device_name     search_devtab string pointer
                  sc     #109
                  test   r5                   name not found

                  jr     nz,command_err
                  ldl    table_pointer,rr8
                         .
                         .
```

**Errors**

If the device is not found, a Hex FFFF (nil) is returned in R5.

## 113 CloseAllWindows

Closes any existing windows from 2 to 16.

### Input/Output Parameters

This call has no parameters

### Characteristics

This call will close all existing windows except for window 1.

### Errors

No errors are returned.

## 114 KbSetLock

Sets the state of both the shift lock and the cursor lock flags.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | R6 ←——— | integer flag |
| Output: | R7 ———→ | previous flag |
| | R5 ———→ | error status |

### Characteristics

The "integer flag" input in R6 is in the range 0-3 and sets the shift lock (for the alpha keys on the alphanumeric keypad) and cursor lock (for the numeric keypad) as follows:

0 = Both flags reset

1 = Shift lock on and cursor lock off

2 = Shift lock off and cursor lock on

3 = Both flags set

**Note:**

The cursor lock condition can also be obtained with the key combination "Control /", while the shift lock with the key combination "Command /".

### Errors

If there are any errors the status is returned in R5. If there are no errors, a zero will be returned in R5.

## 115 ClearText

Clears a rectangle of text in the current window.

### Input/Output Parameters

```
Input:      R10 <----- column (left edge of cleared rectangle)
            R11 <----- row (top row of cleared rectangle)
            R12 <----- column count (width of rectangle)
            R13 <----- row count (height of rectangle)

Output:     R5  -----> error status
```

### Characteristics

ClearText simply clears the specified rectangle to the current background colour of the window. In a colour system, ClearText always clears all screen planes in the specified rectangle, which have corresponding bits set in the Colour Plane Mask parameter (see ScrollText SC 116).

In this system call, the Colour Plane Mask parameter is set to 7, so that a complete clear of the rectangle is done, no matter what system this is executed on.

The range of a column parameter is from 1 to the width of the current window, and the range of a row parameter is from 1 to the number of text lines in a window, i.e. :

1<= Column + Column_count -1<= width of window, and

1<= Row + Row_count - 1<= number of text lines in the window

### Errors

The ranges of the above parameters are checked. An error is returned in R5 if the specified rectangles are not entirely within the window.

## 116 ScrollText

Copies a rectangle of text characters in a window to another position of the same window.

### Input/Output Parameters

| | | | |
|---|---|---|---|
| Input: | R6 | ← | colour plane mask |
| | R7 | ← | logical function (0 for normal copy) |
| | R8 | ← | source column (left edge of source) |
| | R9 | ← | source row (top row of source) |
| | R10 | ← | destination column (left edge of destination) |
| | R11 | ← | destination row (top row of destination) |
| | R12 | ← | column count (width of rectangle) |
| | R13 | ← | row count (height of rectangle) |
| Output: | R5 | → | error status |

### Characteristics

ScrollText is used for copying a rectangular block of text from one portion of a screen window to another. (Note that this cannot be used for copying from one window to another window.) The source and destination areas may overlap; in this case, copying is done in such a way that the overlapped area is copied last: the destination will be a true copy of the the original source, even though the source has been overwritten.

The values for the "logical function" input in R7 are:

| | |
|---|---|
| 0 | Copy text |
| 1 | XOR (exclusive OR) source with destination |
| 2 | AND source with destination |
| 3 | COM: Complement destination, no copy |
| 4 | OR source with destination |
| 5 | INVERT: Complement text, copy |

The "colour plane mask" parameter determines which memory planes are affected by the ScrollText call. It only applies when logical functions 1, 3, or 5 are used, otherwise this parameter is ignored, and its value is preserved. This contains a bit for each memory plane to be written in: bit 0 denotes the first 16K block of screen memory, bit 1 denotes the second 16K block (in 4-colour and 8-colour sytems), and bit 2 denotes the third block used in the 8-colour system. Bits higher than appropriate for a particular system will be ignored: for example, bits 1 and 2 will be ignored on monochrome hardware.

Any program which does not make use of colour (i.e. which only uses colours 0 and 1) should use the value 1 for the "colour plane mask parameter"; this will prevent writing in the second (or third) screen planes of a colour system, if the XOR or COM functions are used.

On the other hand, programs which do use colours other than 0 and 1 should use values 3 (bits 0 and 1) for a 4-colour system, and 7 (bits 0, 1, and 2) for an 8-colour system (actually, 7 may be used for all systems, in this case, the apparent effect of certain logical functions will vary between different types of display).

If the logical function is 0, 2 or 4, ScrollText will obey the same convention regarding the number of screen memory planes written as the screen text and graphics driver: e.g. if the foreground colour is 1 and the background is 0, only the first screen memory plane will be written in.

The range of a column parameter is from 1 to the width of the current window, and the range of a row parameter is from 1 to the number of text lines in a window, i.e. :

1<= Column + Column_count -1<= width of window, and

1<= Row + Row_count - 1<= number of text lines in the window.

## Errors

The ranges of the above parameters are checked by these system calls; an error is returned if the specified rectangles are not entirely within the window. No clipping is done the rectangles specified must be entirely within the window.

**120 New**

Allocates a block of bytes from heap.

**Input/Output Parameters**

| | |
|---|---|
| Input: | RR8 $\longleftarrow$ address of block pointer |
| | R10 $\longleftarrow$ length |
| Output: | R5 $\longrightarrow$ error status |
| | @RR8 $\longrightarrow$ block pointer |

**Characteristics**

This call allocates a block of bytes from the heap, returning a pointer to the location of the first byte of the block. The input "address of block pointer" is the address of a long (4byte) memory location, that is, the address where 'New' stores the block. The input 'length' is the number of bytes to be allocated.

EXAMPLE:

```
                    .
                    .
        addptr      DSL 1
        length      ASSIGN ...
                    .
                    .
        LDA         RR8,addptr
        LD          R10,#length
        sc          #120
        LDL         RR6,@RR8
                    .
                    .
```

In this example, RR6 contains the block starting address. If the block cannot be allocated, RR6 contains a nil (hex FFFFFFFF) pointer (Note: nil = -1), but no error will be returned in R5.

Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) is returned.

## 121 BrandNewAbsolute

Allocates a block at a specified address.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | RR8 ⟵ | address of block pointer |
| | R10 ⟵ | length |
| | @RR8 ⟵ | block pointer |
| Output: | R5 ⟶ | error status |

### Characteristics

This call is similar to a New (SC 120) except that the block allocated is at a specified address.

The input address (RR8) is the address of a long (4-byte) memory location; this is where the desired address is stored. The input to R10 is the number of bytes requested, and must be even.

On exit from this call, the memory location that RR8 points to contains a 32-bit address of the actual block allocated. If the space requested is not available, a nil-pointer (hex FFFFFFF) is returned in the memory location pointed to by RR8, but no error is returned in R5.

It is important to remember that RR8 does not contain the memory address specified.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) is returned.

## 122 NewLargestBlock

Allocates the largest block of bytes from heap.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | RR8 ⟵ | address of block pointer |
| Output: | @RR8 ⟶ | block pointer |
| | R10 ⟶ | length |
| | R5 ⟶ | error status |

### Characteristics

This procedure allocates a the largest free block in memory, returning a pointer to the location of the first byte of the block and the length of that block.

The input pointer should be the address of a long (4byte) memory location; that is the address where 'NewLargestBlock' stores the block start address.

If the block cannot be allocated, RR8 contains a nil (hex FFFFFFFF) pointer but no error is returned in R5.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) is returned.

## 123 StickyNew

Allocates a block of bytes from heap that remains allocated after the program doing this call terminates.

### Input/Output Parameters

| | | |
|---|---|---|
| Input: | RR8 ⟵ | address of block pointer |
| | R10 ⟵ | length |
| Output: | @RR8 ⟶ | block pointer |
| | R5 ⟶ | error status |

### Characteristics

This call allocates a block of bytes from the heap, returning a pointer to the location of the first byte of the block.

The input "address of block pointer" is the address of a long (4byte) memory location, that is, the address where the block start address is stored. The input 'length' is the number of bytes to be allocated.

This call is just like NewAnySegment, but is used for those rare occasions when the allocated block is not to be de-allocated when the "calling" program terminates.

### Errors

If there are any errors, the status code is returned in R5. If there are no errors, a zero (0) is returned.

# APPENDICES

# A. RESERVED WORDS

## RESERVED WORDS

The following symbols are recognized for their specific meanings by the assembler. They cannot be used by the programmer as variable names. If the programmer uses one of these symbols by mistake, the assembler flags its occurrence with error 86, Multiple Definition.

| Reserved Word | Use |
|---|---|
| ADC | mnemonic |
| ADCB | mnemonic |
| ADD | mnemonic |
| ADDB | mnemonic |
| ADDL | mnemonic |
| AND | mnemonic |
| ANDB | mnemonic |
| ASSIGN | directive |
| AT | directive |
| BIT | mnemonic |
| BITB | mnemonic |
| C | condition code |
| CALL | mnemonic |
| CALR | mnemonic |
| CLR | mnemonic |
| CLRB | mnemonic |
| COM | mnemonic |
| COMB | mnemonic |
| COMFLG | mnemonic |
| COMMON | directive |
| CP | mnemonic |
| CPB | mnemonic |
| CPD | mnemonic |
| CPDB | mnemonic |
| CPDR | mnemonic |
| CPDRB | mnemonic |
| CPI | mnemonic |
| CPIB | mnemonic |
| CPIR | mnemonic |
| CPIRB | mnemonic |
| CPL | mnemonic |
| CPSD | mnemonic |
| CPSDB | mnemonic |
| CPSDR | mnemonic |
| CPSDRB | mnemonic |
| CPSI | mnemonic |
| CPSIB | mnemonic |
| CPSIR | mnemonic |
| CPSIRB | mnemonic |
| DAB | mnemonic |
| DBJNZ | mnemonic |
| DD | directive |

| Reserved Word | Use |
|---|---|
| DDB | directive |
| DDL | directive |
| DEC | mnemonic |
| DECB | mnemonic |
| DI | mnemonic |
| DIV | mnemonic |
| DIVL | mnemonic |
| DJNZ | mnemonic |
| DS | directive |
| DSB | directive |
| DSL | directive |
| EI | mnemonic |
| ENDIF | directive |
| EQ | condition code |
| EX | mnemonic |
| EXB | mnemonic |
| EXTERNAL | directive |
| EXTS | mnemonic |
| EXTSB | mnemonic |
| EXTSL | mnemonic |
| FALSE | condition code |
| FCW | control word |
| FLAGS | control word |
| GE | condition code |
| GLOBAL | directive |
| GT | condition code |
| HALT | mnemonic |
| IF | directive |
| IN | mnemonic |
| INB | mnemonic |
| INC | mnemonic |
| INCB | mnemonic |
| INCLUDE | directive |
| IND | mnemonic |
| INDB | mnemonic |
| INDR | mnemonic |
| INDRB | mnemonic |
| INI | mnemonic |
| INIB | mnemonic |
| INIR | mnemonic |
| INIRB | mnemonic |
| IRET | mnemonic |
| JP | mnemonic |
| JR | mnemonic |
| LD | mnemonic |
| LDA | mnemonic |
| LDAR | mnemonic |
| LDB | mnemonic |
| LDCTL | mnemonic |
| LDCTLB | mnemonic |
| LDD | mnemonic |
| LDDB | mnemonic |
| LDDR | mnemonic |

## RESERVED WORDS

| Reserved Word | Use |
|---|---|
| LDDRB | mnemonic |
| LDI | mnemonic |
| LDIB | mnemonic |
| LDIR | mnemonic |
| LDIRB | mnemonic |
| LDK | mnemonic |
| LDL | mnemonic |
| LDM | mnemonic |
| LDPS | mnemonic |
| LDR | mnemonic |
| LDRB | mnemonic |
| LDRL | mnemonic |
| LE | condition code |
| LISTOFF | directive |
| LISTON | directive |
| LT | condition code |
| MBIT | mnemonic |
| MI | condition code |
| MODULE | directive |
| MREQ | mnemonic |
| MRES | mnemonic |
| MSET | mnemonic |
| MULT | mnemonic |
| MULTL | mnemonic |
| NC | condition code |
| NE | condition code |
| NEG | mnemonic |
| NEGB | mnemonic |
| NONSEGMENTED | module type |
| NOP | mnemonic |
| NOV | condition code |
| NSP | control word |
| NSPOFF | control word |
| NSPSEG | control word |
| NVI | interrupt |
| NZ | condition code |
| OR | mnemonic |
| ORB | mnemonic |
| OTDR | mnemonic |
| OTDRB | mnemonic |
| OTIR | mnemonic |
| OTIRB | mnemonic |
| OUT | mnemonic |
| OUTB | mnemonic |
| OUTD | mnemonic |
| OUTDB | mnemonic |
| OUTI | mnemonic |
| OUTIB | mnemonic |
| OV | condition code |
| P | flag |
| PAGE | directive |
| PE | condition code |
| PL | condition code |

| Reserved Word | Use |
|---|---|
| PO | condition code |
| POP | mnemonic |
| POPL | mnemonic |
| PSAP | control word |
| PSAPOFF | control word |
| PSAPSEG | control word |
| PUSH | mnemonic |
| PUSHL | mnemonic |
| R0 | word register |
| R1 | word register |
| R10 | word register |
| R11 | word register |
| R12 | word register |
| R13 | word register |
| R14 | word register |
| R15 | word register |
| R2 | word register |
| R3 | word register |
| R4 | word register |
| R5 | word register |
| R6 | word register |
| R7 | word register |
| R8 | word register |
| R9 | word register |
| REFRESH | control word |
| RES | mnemonic |
| RESB | mnemonic |
| RESFLG | mnemonic |
| RET | mnemonic |
| RH0 | byte register |
| RH1 | byte register |
| RH2 | byte register |
| RH3 | byte register |
| RH4 | byte register |
| RH5 | byte register |
| RH6 | byte register |
| RH7 | byte register |
| RL | mnemonic |
| RL0 | byte register |
| RL1 | byte register |
| RL2 | byte register |
| RL3 | byte register |
| RL4 | byte register |
| RL5 | byte register |
| RL6 | byte register |
| RL7 | byte register |
| RLB | mnemonic |
| RLC | mnemonic |
| RLCB | mnemonic |
| RLDB | mnemonic |
| RQ0 | quad register |
| RQ12 | quad register |
| RQ4 | quad register |

## RESERVED WORDS

| Reserved Word | Use |
|---|---|
| RQ8 | quad register |
| RR | mnemonic |
| RR0 | long register |
| RR10 | long register |
| RR12 | long register |
| RR14 | long register |
| RR2 | long register |
| RR4 | long register |
| RR6 | long register |
| RR8 | long register |
| RRB | mnemonic |
| RRC | mnemonic |
| RRCB | mnemonic |
| RRDB | mnemonic |
| S | flag |
| SBC | mnemonic |
| SBCB | mnemonic |
| SC | mnemonic |
| SDA | mnemonic |
| SDAB | mnemonic |
| SDAL | mnemonic |
| SDL | mnemonic |
| SDLB | mnemonic |
| SDLL | mnemonic |
| SECTION | directive |
| SEGMENTED | module type |
| SET | mnemonic |
| SETB | mnemonic |
| SETFLG | mnemonic |
| SIN | mnemonic |
| SINB | mnemonic |
| SIND | mnemonic |
| SINDB | mnemonic |
| SINDR | mnemonic |
| SINDRB | mnemonic |
| SINI | mnemonic |
| SINIB | mnemonic |
| SINIR | mnemonic |
| SINIRB | mnemonic |
| SLA | mnemonic |
| SLAB | mnemonic |
| SLAL | mnemonic |
| SLL | mnemonic |
| SLLB | mnemonic |
| SLLL | mnemonic |
| SOTDR | mnemonic |
| SOTDRB | mnemonic |
| SOTIR | mnemonic |
| SOTIRB | mnemonic |
| SOUT | mnemonic |
| SOUTB | mnemonic |
| SOUTD | mnemonic |
| SOUTI | mnemonic |
| SOUTIB | mnemonic |

| Reserved Word | Use |
|---|---|
| SRA | mnemonic |
| SRAB | mnemonic |
| SRAL | mnemonic |
| SRL | mnemonic |
| SRLB | mnemonic |
| SRLL | mnemonic |
| SUB | mnemonic |
| SUBB | mnemonic |
| SUBL | mnemonic |
| TCC | mnemonic |
| TCCB | mnemonic |
| TEMPLATE | directive |
| TEST | mnemonic |
| TESTB | mnemonic |
| TESTL | mnemonic |
| TITLE | directive |
| TRDB | mnemonic |
| TRDRB | mnemonic |
| TRIB | mnemonic |
| TRIRB | mnemonic |
| TRTDB | mnemonic |
| TRTDRB | mnemonic |
| TRTIB | mnemonic |
| TRTIRB | mnemonic |
| TRUE | condition code |
| TSET | mnemonic |
| TSETB | mnemonic |
| UGE | condition code |
| UGT | condition code |
| ULE | condition code |
| ULT | condition code |
| V | flag |
| VI | interrupt |
| XOR | mnemonic |
| XORB | mnemonic |
| Z | condition code |

## B. ASM ERRORS

## ASM ERRORS

The following is a complete list of errors that can be returned by the ASM command, during assembly time. The errors refer to source file line numbers in the context of the program.

**Bad statement errors:**

| Error Number | Meaning |
|---|---|
| 1 | Bad Line |
| 2 | Bad Label/Mnemonic Field or Context |
| 3 | Bad IF/ENDIF |
| 4 | Bad Directive or Context |
| 5 | Bad Labelled Directive or Context |
| 6 | Bad Module/Section or Context |
| 7 | Bad Argument or Context |
| 8 | Bad Byte-Data Context |
| 9 | Bad Word-Data Context |
| 10 | Bad Long-Data Context |
| 11 | Source line truncated |
| 12 | IF not terminated by ENDIF |
| 13 | ENDIF with no matching IF |

**Bad Character, Identifier or Constant:**

| Error Number | Meaning |
|---|---|
| 14 | Identifier Too Long |
| 15 | Single-Quoted Text Too Long or bad use of % |
| 16 | Quote Not Closed or bad use of % |
| 17 | Illegal Number |
| 18 | Illegal Character |
| 19 | Illegal base specification in number |
| 20 | Illegal Keyword |
| 21 | End of Line in number |
| 22 | Keyword in label field |
| 23 | Mnemonic not in mnemonic field |

**Bad Expression:**

| Error Number | Meaning |
|---|---|
| 24 | Bad Parenthesis Use |
| 25 | Segment/Section/External mismatch in relational expression |
| 26 | Illegal relational operands |
| 27 | Illegal Type Combination in Expression |
| 28 | Illegal operator in address term |
| 29 | Segment/Section/External mismatch in additive term |
| 30 | Illegal additive operand |
| 31 | Address type mismatch in additive term |
| 32 | Illegal multiplicative operand |
| 33 | Illegal unary operand |
| 34 | Absolute segment number out of range |
| 35 | Illegal type in segment/offset of absolute address |

**Bad Operand:**

| Error Number | Meaning |
|---|---|
| 38 | Bad Use of Short Address |
| 39 | Bad Argument or Context |
| 40 | Invalid Address Register |
| 41 | DD, DDB or DDL operand, Wrong Size |
| 42 | Index Register is Invalid |
| 43 | DD Repeat Nesting Error |
| 44 | Wrong Register Type |
| 45 | Indirect Register is Zero |
| 46 | Immediate Operand Wrong Size |
| 47 | Base Register zero not allowed |
| 48 | Index Register zero not allowed |
| 49 | Even address required |
| 50 | Invalid Relative Address |
| 51 | Relative out of Range |
| 52 | Invalid Short Extraction |
| 53 | Absolute Address too Large for Short Extraction |
| 54 | Invalid Segment or Offset Extraction |
| 55 | Invalid Small Immediate |
| 56 | Extra Operands Ignored |
| 57 | Illegal Operand |
| 58 | Truncation Warning |

ASM ERRORS

| | |
|---|---|
| 59 | Section or Module Name out of place |
| 60 | Invalid Addrress |
| 61 | DD overflows 64K |
| 62 | No prior Section for SECTION * |
| 63 | Page size specified is too small |
| 64 | Undefined or non-numeric page size |
| 65 | Unexpected end of line |
| 66 | Bad Operator/Value |

**Undefined Symbol:**

| Error Number | Meaning |
|---|---|
| 70 | Undefined Symbol (Second pass only) |

**Bad Location or Definition:**

| Error Number | Meaning |
|---|---|
| 72 | Symbol not defined until second pass |
| 73 | Symbol redefined in second pass |
| 74 | Location Counter overflowed 64K |
| 75 | Warning: Address incremented to even value |

First Pass Errors:

| Error Number | Meaning |
|---|---|
| 86 | Multiple Definition |
| 88 | IF Value Not Defined |
| 89 | Invalid ATparm, DSparm or DD Repeat Count |
| 90 | Undefined ATparm, DSparm Template Base or DD |

**Fatal Errors:**

| Error Number | Meaning |
|---|---|
| 93 | Symbol Table Full - Terminate |
| 94 | Unknown Character in file |
| 95 | Internal Object Table full |
| 96 | Internal Object Table full |
| 97 | Too many INCLUDEs |
| 98 | Binary data file absent or improper |

## C. FUNCTIONAL LIST OF SYSTEM CALLS

FUNCTIONAL LIST OF SYSTEM CALLS

## 1. BYTESTREAM CALLS

| Name | System Call | Parameter | Register |
|---|---|---|---|
| LookByte | 9 | DID | R8 |
| | | returned byte | RL7 |
| | | buffer status(00 or FF) | RH7 |
| | | error status | R5 |
| GetByte | 10 | DID | R8 |
| | | returned byte | R7 |
| | | error status | R5 |
| PutByte | 11 | DID | R8 |
| | | input byte value | RL7 |
| | | error status | R5 |
| ReadBytes | 12 | DID | R8 |
| | | input count | R9 |
| | | input ptr to memory | RR10 |
| | | returned count | R7 |
| | | error status | R5 |
| WriteBytes | 13 | DID | R8 |
| | | input count | R9 |
| | | input ptr to memory | RR10 |
| | | returned count | R7 |
| | | error status | R5 |
| ReadLine | 14 | DID | R8 |
| | | input count | R9 |
| | | input ptr to memory | RR10 |
| | | count returned | R6 |
| | | error status | R5 |
| Eof | 16 | DID | R8 |
| | | returned status | R9 |
| | | error status | R5 |
| ResetByte | 18 | DID | R8 |
| | | error status | R5 |
| Close | 19 | DID | R8 |
| | | error status | R5 |
| SetControlByte | 20 | DID | R8 |
| | | input word number | R9 |
| | | input word | R10 |
| | | error status | R5 |

| Name | System Call | Parameter | Register |
|---|---|---|---|
| GetStatusByte | 21 | DID | R8 |
| | | input word number | R9 |
| | | returned word read | R10 |
| | | error status | R5 |
| OpenFile (files) | 22 | DID | R8 |
| | | input extent length | R6 |
| | | input mode | R7 |
| | | input file id. length | R9 |
| | | input ptr to addr | RR10 |
| | | error status | R5 |
| OpenFile (RS-232) | 22 | DID | R8 |
| | | error status | R5 |
| DSeek | 23 | DID | R8 |
| | | input position | RR10 |
| | | error status | R5 |
| DGetLen (files) | 24 | DID | R8 |
| | | returned length | RR10 |
| | | error status | R5 |
| DGetLen (RS-232) | 24 | DID | R8 |
| | | returned zero status | R10 |
| | | returned number of bytes | R11 |
| | | error status | R5 |
| DGetPosition | 25 | DID | R8 |
| | | returned position | RR10 |
| | | error status | R5 |
| DRemove | 26 | input length | R9 |
| | | input ptr to name | RR10 |
| | | error status | R5 |
| DRename | 27 | input old address | RR6 |
| | | input old length | R8 |
| | | input new address | RR10 |
| | | input new length | R9 |
| | | error status | R5 |
| DDirectory | 28 | input file id. length | R9 |
| | | input address | RR10 |
| | | error status | R5 |

## 2. BLOCK TRANSFER CALLS

| Name | System Call | Parameter | Register |
|---|---|---|---|
| BSet | 29 | input n (byte value) | RL7 |
| | | input ptr to memcry | RR8 |
| | | input length | R10 |
| | | error status | R5 |
| BWSet | 30 | input n (word value) | R7 |
| | | input ptr to memory | RR8 |
| | | input length | R10 |
| | | error status | R5 |
| BClear | 31 | input ptr to memory | RR8 |
| | | input length | R10 |
| | | error status | R5 |
| BMove | 32 | input length | R7 |
| | | input ptr to old memory | RR8 |
| | | input ptr to new memory | RR10 |
| | | error status | R5 |

## 3. STORAGE ALLOCATION CALLS

| Name | System Call | Parameter | Register |
|---|---|---|---|
| NewSameSegment | 33 | address of block pointer | RR8 |
| | | input length | R10 |
| | | error status | R5 |
| | | returned block pointer | @RR8 |
| Dispose | 34 | address of block pointer | RR8 |
| | | input length | R10 |
| | | error status | R5 |
| | | Hex FFFFFFFF | @RR8 |
| MaxSize | 99 | returned size | R8 |
| | | error status | R5 |
| NewAbsolute | 104 | address of block pointer | RR8 |
| | | input length | R10 |
| | | input block pointer | @RR8 |
| | | error status | R5 |

| Name | System Call | Parameter | Register |
|---|---|---|---|
| New | 120 | address of block pointer | RR8 |
| | | input length | R10 |
| | | error status | R5 |
| | | returned block pointer | @RR8 |
| BrandNewAbsolute | 121 | address of block pointer | RR8 |
| | | input length | R10 |
| | | input block pointer | @RR8 |
| | | error status | R5 |
| NewLargestBlock | 122 | address of block pointer | RR8 |
| | | returned block pointer | @RR8 |
| | | returned length | R10 |
| | | error status | R5 |
| StickyNew | 123 | address of block pointer | RR8 |
| | | input length | R10 |
| | | error status | R5 |
| | | returned block pointer | @RR8 |

## 4. GRAPHICS SYSTEM CALLS

| Name | System Call | Parameter | Register |
|---|---|---|---|
| Cls | 35 | (no parameters) | |
| ChgCur0 | 36 | input column | R8 |
| | | input row | R9 |
| | | error status | R5 |
| ChgCur1 | 37 | input x | R8 |
| | | input y | R9 |
| ChgCur2 | 38 | input blink rate | R8 |
| ChgCur3 | 39 | input blink rate | R8 |
| ChgCur4 | 40 | input ptr to array | RR8 |
| ChgCur5 | 41 | input ptr to array | RR8 |
| ReadCur0 | 42 | input ptr to array | RR10 |
| | | output blink rate | R7 |
| | | output column | R8 |
| | | output row | R9 |
| | | error status | R5 |

## FUNCTIONAL LIST OF SYSTEM CALLS

| Name | System Call | Parameter | Register |
|---|---|---|---|
| ReadCur1 | 43 | input ptr to array | RR10 |
| | | output blink rate | R7 |
| | | output x-position | R8 |
| | | output y-position | R9 |
| | | error status | R5 |
| SelectCur | 44 | input select | R8 |
| GrfInit | 45 | output colour flag | R8 |
| | | output ptr to m-box | RR10 |
| PalatteSet | 46 | input colour A | R8 |
| | | input colour B | R9 |
| | | input colour C | R10 |
| | | input colour D | R11 |
| | | error status | R5 |
| DefineWindow | 47 | input quadrant | R8 |
| | | input position | R9 |
| | | input vert-spacing | R10 |
| | | input horz-spacing | R12 |
| | | output window number | R11 |
| | | error status | R5 |
| SelectWindow | 48 | input window number | R8 |
| | | error status | R5 |
| ReadWindow | 49 | output window number | R7 |
| | | output x-size | R8 |
| | | output y-size | R9 |
| | | output foreground colour | R10 |
| | | output background colour | R11 |
| | | error status | R5 |
| ChgWindow | 50 | input foreground colour | R8 |
| | | input background colour | R9 |
| | | error status | R5 |
| CloseWindow | 51 | input window number | R8 |
| ScaleXY | 52 | input x-position | R8 |
| | | input y-position | R9 |
| | | return_value | R10 |
| MapXYC | 53 | input x-position | R8 |
| | | input y-position | R9 |
| MapCXY | 54 | returned x-position | R8 |
| | | returned y-position | R9 |
| FetchC | 55 | returned C-value | RR8 |

| Name | System Call | Parameter | Register |
|---|---|---|---|
| StoreC | 56 | input C-value | RR8 |
| UpC | 57 | (no parameters) | |
| DownC | 58 | (no parameters) | |
| LeftC | 59 | (no parameters) | |
| RightC | 60 | (no parameters) | |
| SetAtr | 61 | input colour | R8 |
| | | error status | R5 |
| SetC | 62 | input operation | R8 |
| ReadC | 63 | returned colour | R8 |
| NSetCX | 64 | input hor. line count | R8 |
| | | input operation | R9 |
| NSetCY | 65 | input vertical line count | R8 |
| | | input operation | R9 |
| NRead | 66 | input width (count) | R8 |
| | | input height (count) | R9 |
| | | input ptr to array | RR10 |
| | | always cleared | R5 |
| | | returned addr. of array | @RR10 |
| NWrite | 67 | input logical function | R7 |
| | | input width (count) | R8 |
| | | input height (count) | R9 |
| | | input ptr to array | RR10 |
| | | always cleared | R5 |
| PntInit | 68 | input paint colour | R8 |
| | | input border colour | R9 |
| | | error status | R5 |
| TDownC | 69 | returned check value | R8 |
| TUpC | 70 | returned check value | R8 |
| ScanL | 71 | returned count-1 | R9 |
| | | returned margin flag | R10 |
| | | returned painted flag | R11 |

| Name | System Call | Parameter | Register |
|---|---|---|---|
| ScanR | 72 | input maxcount | R8 |
| | | returned C-type | RR6 |
| | | returned maxcount | R8 |
| | | returned count-r | R9 |
| | | returned margin flag | R10 |
| | | returned painted flag | R11 |
| CloseAllWindows | 113 | (no parameters) | |
| ClearText | 115 | input column | R10 |
| | | input row | R11 |
| | | input column count | R12 |
| | | input row count | R13 |
| | | error status | R5 |
| ScrollText | 116 | input color plane mask | R6 |
| | | input logical function | R7 |
| | | input source column | R8 |
| | | input source row | R9 |
| | | input destination column | R10 |
| | | input destination row | R11 |
| | | input column count | R12 |
| | | input row count | R13 |
| | | error status | R5 |

## 5. TIME AND DATE CALLS

| Name | System Call | Parameter | Register |
|---|---|---|---|
| SetTime | 73 | input addr of data | RR8 |
| | | input length of string | R10 |
| | | error status | R5 |
| SetDate | 74 | input addr of data | RR8 |
| | | input length of string | R10 |
| | | error status | R5 |
| GetTime | 75 | input addr of data | RR8 |
| | | input length of string | R10 |
| | | error status | R5 |
| GetDate | 76 | input addr of data | RR8 |
| | | input length of string | R10 |
| | | error status | R5 |

## 6. USER CODE CALLS

| Name | System Call | Parameter | Register |
|---|---|---|---|
| CallUser | 77 | input pointer (system stack has a pointer to 2-character symbol, list of parameter pointers, number of parameters) | RR14 |
| | | error status | R5 |

## 7. IEEE-488 CALLS

| Name | System Call | Parameter | Register |
|---|---|---|---|
| IBSrQ0 | 78 | error status | R5 |
| IBSrQ1 | 79 | error status | R5 |
| IBPoll | 80 | input talker addr | R8 |
| | | returned ptr to status | RR10 |
| | | error status | R5 |
| IBISet | 81 | input operand | R8 |
| | | error status | R5 |
| IBRSet | 82 | error status | R5 |
| IBPrnt | 83 | input buffer addr | RR6 |
| | | input listener addr | R8 |
| | | input buffer length | R9 |
| | | input delimiter | R10 |
| | | error status | R5 |
| IBWByt | 84 | input numval addr | RR6 |
| | | input comlist addr | R8 |
| | | input numval length | R9 |
| | | input comlist addr | RR10 |
| | | error status | R5 |
| IBInpt | 85 | input buffer length | R7 |
| | | input talker addr | R8 |
| | | input listener addr | R9 |
| | | input buffer addr | RR10 |
| | | returned buffer length | R7 |
| | | error status | R5 |

| Name | System Call | Parameter | Register |
|---|---|---|---|
| IBLinpt | 86 | input buffer length | R7 |
| | | input talker addr | R8 |
| | | input listener addr | R9 |
| | | input buffer addr | RR10 |
| | | returned buffer length | R7 |
| | | error status | R5 |
| IBRByt | 87 | input buffer addr | RR6 |
| | | input comlist length | R8 |
| | | input buffer length | R9 |
| | | input comlist addr | RR10 |
| | | error status | R5 |

## 8. MISCELLANEOUS SYSTEM CALLS

| Name | System Call | Parameter | Register |
|---|---|---|---|
| Error | 88 | input parameter num | RH5 |
| | | input error code | RL5 |
| DString | 89 | input addr of string | RR12 |
| | | error status | R5 |
| CrLf | 90 | error status | R5 |
| DHexByte | 91 | input byte | R12 |
| | | error status | R5 |
| DHex | 92 | input word | R12 |
| | | error status | R5 |
| DHexLong | 93 | input long word | RR12 |
| | | error status | R5 |
| DNumW | 94 | input integer | R12 |
| | | input field width | R13 |
| | | error status | R5 |
| DLong | 95 | input long integer | RR12 |
| | | error status | R5 |
| DisectName | 96 | input string length | R9 |
| | | input string addr | RR10 |
| | | input names record addr | RR12 |
| | | error status | R5 |
| | | returned volume number | R7 |
| | | returned names record | @RR12 |

| Name | System Call | Parameter | Register |
|---|---|---|---|
| CheckVolume | 97 | error status | R5 |
| Search | 98 | input drive | R6 |
| | | input search mode | R7 |
| | | input length | R9 |
| | | input file pointer | RR10 |
| | | input file name pointer | RR12 |
| | | returned length | R9 |
| | | returned file pointer | RR10 |
| | | modified | RR12 |
| | | error status | R5 |
| SetVol | 102 | input volume number | R7 |
| | | error status | R5 |
| StringLen | 105 | input pointer | RR12 |
| | | returned length | R7 |
| | | error status | R5 |
| DiskFree | 106 | input volume number | R7 |
| | | returned num of sectors | RR10 |
| | | error status | R5 |
| BootSystem | 107 | error status | R5 |
| SetSysSeg | 108 | error status | R5 |
| SearchDevTab | 109 | input ptr device name | RR10 |
| | | input dev name length | R9 |
| | | returned entry number | RL5 |
| | | returned device type | RH5 |
| | | returned ptr table entry | RR8 |
| | | error status | R5 |
| KbSetLock | 114 | input integer flag | R6 |
| | | returned previous state | R7 |
| | | error status | R5 |

## D. DEVICE ID (DID) ASSIGNMENTS

## DEVICE ID (DID) ASSIGNMENTS

The following table describes the allocation of DID's to various functions. Some of these DID's represent devices which are always open; others are assigned to files or screen windows by system calls.

| | |
|---|---|
| 1 | BASIC files |
| 2 | . |
| . | . |
| . | . |
| 15 | . |
| 17 | Console |
| 18 | Printer |
| 19 | Communications RS-232-C |
| 20 | System Disk Files (Not accessible to BASIC) |
| . | . |
| . | . |
| . | . |
| 24 | . |
| 25 | Com1 (RS-232-C) |
| 26 | Com2 (RS-232-C) |

## E. SYSTEM ERRORS

## SYSTEM ERRORS

| ERROR CODE (Decimal) | ERROR Description | Error code in hexadecimal (returned in R5) |
|---|---|---|
| 0 | no error | 00 |
| 3 | invalid termination of input bytestream | 03 |
| 7 | out of memory | 07 |
| 9 | invalid listener or talker address | 09 |
| 10 | no IEEE board | 0A |
| 11 | time out error | 0B |
| 13 | bad data type | 0D |
| 35 | window does not exist | 23 |
| 36 | window create error | 24 |
| 53 | file not found | 35 |
| 54 | bad file open mode | 36 |
| 55 | file already open | 37 |
| 57 | disk i/o | 39 |
| 58 | file already exists | 3A |
| 60 | disk not initialized | 3C |
| 61 | disk filled | 3D |
| 62 | end of file | 3E |

|  |  |  |
|---|---|---|
| 63 | bad record number | 3F |
| 64 | bad file name | 40 |
| 71 | volume name not found | 47 |
| 73 | invalid volume number | 49 |
| 75 | volume not enabled | 4B |
| 76 | password not valid | 4C |
| 77 | illegal disk change | 4D |
| 78 | write protected file | 4E |
| 79 | copy protected file | 4F |
| 90 | error in parameter | 5A |
| 91 | too may parameters | 5B |
| 92 | command not found | 5C |
| 96 | file not open | 60 |
| 99 | bad load file | 63 |
| 101 | time or date | 65 |
| 106 | function key already exists | 6A |
| 108 | call-user | 6C |
| 110 | time-out | 6E |
| 111 | invalid device | 6F |

## F. M20 I/O PORT ADDRESSES

M20 I/O PORT ADDRESSES

Port Addresses are here listed in 4 groups:

1. Main Motherboard Ports
2. IEEE Expansion Board Ports
3. Hard Disk Unit Expansion Board Ports
4. RS-232-C Twin Expansion Board Ports

**MAIN MOTHERBOARD PORTS:**

| DEVICE | ADDRESS | COMMENT |
|---|---|---|
| FDC | %001 | Status/Command |
| | %003 | Track |
| | %005 | Sector |
| | %007 | Data |
| TTL Latch | %021 | |
| CRTC (Video) | %061 | Address |
| | %063 | Data |
| 8255A (Centronics Parallel Interface) | %081 | Port A |
| | %083 | Port B |
| | %085 | Port C |
| | %087 | Control |
| 8251 (Keyboard) | %0A1 | Data |
| | %0A3 | Status/Control |
| 8251 (TTY/PRTR) | %0C1 | Data |
| | %0C3 | Status/Control |

|  |  |  |
|---|---|---|
| 8253 | %121<br>%123<br>%125<br>%127 | Ctr 0 (TTY/printer timing)<br>Crt 1 (Keyboard timing)<br>Crt 2 (Real time clock-NVI)<br>Control register |
| 8259 (Master) | %140-1<br>%142-3 |  |
| REG FILE<br>(4 colours) | %181<br>%183<br>%185<br>%187 | Loc 1<br>Loc 2<br>Loc 3<br>Loc 4 |

**IEEE EXPANSION BOARD PORTS:**

| DEVICE | ADDRESS | COMMENT |
|---|---|---|
| 8292<br>(GPIB CTLER) | %101<br>%103 | A0 = 0<br>A0 = 1 |
| 8291<br>(GPIB Talker/<br>Listener) | %161<br>%163<br>%165<br>%167 | Data in / Data out<br>Interrupt status / Mask 1<br>Interrupt status / Mask 2<br>Serial poll status / Mode |
|  | %169<br>%16B<br>%16D<br>%16F | Address status / Mode<br>Cmd pass through / Aux mode<br>Address 0 / Address 0/1<br>Address 1 / EOS |
| 8259 (IEEE) | %1A0<br>%1A2 |  |

M20 I/O PORT ADDRESSES

HARD DISK UNIT EXPANSION BOARD PORTS:

| DEVICE | ADDRESS | COMMENT |
|---|---|---|
| cyl_hi | %1cb | cylinder address high register |
| cyl_lo | %1c9 | cylinder address low register |
| head | %1cd | head select register (also contains drive select and bytes per sector) |
| sector | %1c7 | sector for operation |
| command | %1cf | command status register address |
| error | %1c3 | contains error information |
| wr_prcomp | %1c3 | value * 4 = cylinder to start write precompensation |
| data | %1c1 | data port to the interface board |
| sec_cnt | %1c5 | sector count for the format command |

RS-232-C TWIN EXPANSION BOARD PORTS

| DEVICE | ADDRESS | COMMENT |
|---|---|---|
| | for the modem interface | |
| modem_prt | %881 | modem status port |
| | for the interrupt sub-system | |
| exp_int | %841 | 8259 interrupt command register |
| | %843 | 8259 data register |
| | for the serial ports | |
| tp_0 | %803 | 8251a 0 control port |
| | %801 | 8251a 0 data port |
| tp_1 | %823 | 8251a 1 control port |
| | %821 | 8251a 1 data port |
| exp_baud | %867 | 8253 control port |
| | %861 | 8253 out 0 register |
| | %863 | 8253 out 1 register |
| | %865 | 8253 out 3 register |

## G. MAILBOX

## MAILBOX

A mailbox area (8 bytes), used by the IEEE driver, is declared globally by PCOS. The first 6 bytes comprise the array "IEEE"; the next byte is the flag "srq_488" (see also section on IEEE calls and system calls #78 through #87). The next byte indicates which carriage return key, /S1/, /S2/ or the standard /CR/, was pressed last (it should be noted that a zero is returned for any key except /S1/ or /S2/).

On calling GrfInit (SC 45), the interpreter will be passed the address of this area in RR10.

### Format of Mailbox Area

| bytes | description |
|---|---|
| 0-5 | "IEEE" Array; values set by IEEE driver for use by BASIC interpreter. |
| 6 | " srq_488 " flag; value set by IEEE interrupt service routine " ibsrq92 ", tested by the BASIC interpreter. This indicates that a service request has been received. |
| 7 | S1 and S2 key depression flag. Set in keyboard driver; ( 0 = neither key depressed, 1 =/S1/ depressed, 2 =/S2/ depressed) |

## H. M20 - RS-232-C DEVICE PARAMETER TABLE

M20 - RS-232-C DEVICE PARAMETER TABLE

This appeandix details the structure of the Device Parameter Table used by System Calls 20 and 21. These system calls are used for reading and writing device parameters for devices connected to the RS-232-C interfaces.

A knowledge of the hardware in question is useful for a deeper comprehension of this appendix (see M20 hardware literature).

| WORD NUMBER | DESCRIPTION | |
|---|---|---|
| 0-1 | Ring buffer address | (long word) |
| 2 | Ring buffer input address | (word) |
| 3 | Ring buffer output address | (word) |
| 4 | Ring buffer count | (word) |
| 5 | Ring buffer size | (word) |
| 6 | 75% of ring buffer size | (word) |
| 7 | 50% of ring buffer size | (word) |
| 8 | 8251A USART control port address | (word) |
| 9 | 8251A USART state and error flags | (word) |
| 10 | 8251A USART time out for data output | (word) |
| 11 (high) | 8251A USART mode | (byte) |
| 11 (low) | 8253 timer command | (byte) |
| 12 | 8253 timer control port address | (word) |
| 13 | 8253 timer baudrate data port address | (word) |
| 14 | 8253 timer baud rate count | (word) |
| 15 | 8259A PIC port A address | (word) |
| 16 | 8259A PIC SEOI command word | (word) |
| 17 | 8259A PIC - master interrupt mask bit | (word) |
| 18 | 8259A PIC - slave interrupt mask bit | (word) |

Word numbers 0 to 7 contain the state of the ring buffer. Words 8 to 11 (high) contain information relative to the 8251A (Programmable Communication Interface).

Word 8 contains the control port address. This can assume the following values:

%00C3 : USART motherboard control port.

%0803 : USART expansion board 1 control port.

%0823 : USART expansion board 2 control port.

Word 9 represents the status and the error flags for the 8251 and is organised in the following way:

| STATUS | BIT POSITION | LEGAL VALUES | MEANING |
|---|---|---|---|
| Duplex mode | 15 | 1 | full echoing of all input |
| | | 0 | No echoing of input |
| (reserved) | 14 | 0 | (not used) |
| Framing Error | 13 | 1 | a valid stop bit has not been detected at the end of each character. (Reported from 8251A) |
| | | 0 | No Framing Error |
| Overrun Error | 12 | 1 | a character has not been read before the next one becomes available. (Reported from 8251A) |
| | | 0 | No Overrun Error |
| Parity Error | 11 | 1 | a change in parity value has been detected. (Reported from 8251A) |
| | | 0 | No Parity Error |
| Timeout Error | 10 | 1 | a timeout has occured while waiting for the Transmit Ready line on the 8251A |
| | | 0 | No Timeout Error |
| Memory Error | 9 | 1 | driver failed to open buffer - no Open Port call or insufficient memory. |
| | | 0 | No Memory Error |
| Buffer Error | 8 | 1 | interrupt routine tried to overwrite the buffer. |
| | | 0 | No Buffer Error |

M20 - RS-232-C DEVICE PARAMETER TABLE

| STATUS | BIT POSITION | LEGAL VALUES | MEANING |
|---|---|---|---|
| (reserved) | 7 | 0 | (not used) |
| Free-running protocol | 6 | 1<br>0 | free-running protocol,<br>Handshake protocol using XON/XOFF |
| XOFF/XON Flag (M20 previously acted as trans-mitter) | 5 | 1 | XOFF character, sent in previous trans-mission. Buffer is 75% full. XOFF is sent from M20 i.e. other sender should stop. |
| | | 0 | XON character, sent in previous trans-mission. Input buffer is ready to receive characters (default state.) XON is sent from M20 i.e. other sender should start again. |
| Hardware State | 4 | 1 | hardware present and 8259A passed interrupt mask test. |
| | | 0 | No hardware or failed test |
| XOFF/XON Flag (M20 acts as receiver) | 3 | 1 | XOFF character, de-tected in current reception. XOFF character is received from outside. No characters will be transmitted. |
| | | 0 | XON character, de-tected in current reception. XON received from outside. Characters will be transmitted (default state). |
| (reserved) | 2 | 0 | (not used) |
| (reserved) | 1 | 0 | (not used) |
| (reserved) | 0 | 0 | (not used) |

Word 10 contains the time-out value for the transmission of data.

The high byte in word 11 is the 8251A Mode byte and is described below:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | Ø |
|---|---|---|---|---|---|---|---|
| S2 | S1 | EP | PEN | L2 | L1 | B2 | B1 |

| | | | |
|---|---|---|---|
| Number of Stop Bits: | S2 | S1 | |
| | 0 | 1 | 1 stop bit |
| | 1 | 0 | 1.5 stop bits |
| | 1 | 1 | 2 stop bits (default) |
| | 0 | 0 | ILLEGAL |
| Even Parity/ Parity Enable: | EP | PEN | |
| | 0 | 0 | Disable Parity/Odd Parity (default) |
| | 0 | 1 | Enable Parity/Odd Parity |
| | 1 | 1 | Enable Parity/Even Parity |
| | 1 | 0 | Disable Parity/Even Parity |
| Character Length: | L2 | L1 | |
| | 0 | 0 | 5 Data bits |
| | 0 | 1 | 6 Data bits |
| | 1 | 0 | 7 Data bits (default) |
| | 1 | 1 | 8 Data bits |
| Baud Rate Factor: | B2 | B1 | |
| | 1 | 0 | Asynchronous Mode 16 x (default) |
| | 0 | 0 | Synchronous Mode |
| | 0 | 1 | Asynchronous Mode 1 x |
| | 1 | 1 | Asynchronous Mode 64 x |

The low byte in word 11, and words 12 to 14 concern the 8253 timer (Programmable interval timer). The low byte in word 11 is the 8253 command byte described below:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | Ø |
|---|---|---|---|---|---|---|---|
| SC1 | SCØ | RL1 | RLØ | M2 | M1 | MØ | BCD |

## M20 - RS-232-C DEVICE PARAMETER TABLE

| | | | | |
|---|---|---|---|---|
| Counter Select: | | SC1 | SC0 | |
| | | 0 | 0 | Select Counter 0 |
| | | 0 | 1 | Select Counter 1 |
| | | 1 | 0 | Select Counter 2 |
| | | 1 | 1 | ILLEGAL |
| Read/Load Instruction: | | RL1 | RL0 | |
| | | 0 | 0 | Counter Latching Operation |
| | | 0 | 1 | Read/Load most sig. byte only (msb) |
| | | 1 | 0 | Read/Load least sig. byte only (lsb) |
| | | 1 | 1 | Read/Load lsb first, then msb |
| Mode: | M2 | M1 | M0 | |
| | 0 | 0 | 0 | Mode 0: Interrupt on Terminal Count |
| | 0 | 0 | 1 | Mode 1: Programmable One-Shot |
| | x | 1 | 0 | Mode 2: Rate Generator |
| | x | 1 | 1 | Mode 3: Square Wave Rate Generator |
| | 1 | 0 | 0 | Mode 4: Software Triggered Strobe |
| | 1 | 0 | 1 | Mode 5: Hardware Triggered Strobe |
| 4 BCD's/ Binary Word: | BCD | | | |
| | 0 | | | Binary Counter (16 bits) |
| | 1 | | | BCD Counter (4 decades * 4 bits/ decade ) |

Word 12 contains the 8253 control port address; this can be either

| | |
|---|---|
| %0127 | motherboard timer control port |
| %0867 | expansion board timer control port |

Word 13 contains a channel address of an 8253 timer. The address can be one of the following:

| | |
|---|---|
| %0121 | channel 0 motherboard timer |
| %0123 | channel 1 motherboard timer |
| %0125 | channel 2 motherboard timer |
| %0861 | channel 0 expansion board timer |
| %0863 | channel 1 expansion board timer |
| %0865 | channel 2 expansion board timer |

Word 14 sets the transmission baud rate as follows:

| | |
|---|---|
| 1538 | baud count for baud rate of 50 |
| 699 | baud count for baud rate of 110 |
| 256 | baud count for baud rate of 300 |
| 128 | baud count for baud rate of 600 |
| 64 | baud count for baud rate of 1200 |
| 32 | baud count for baud rate of 2400 |
| 16 | baud count for baud rate of 4800 |
| 8 | baud count for baud rate of 9600 |
| 4 | baud count for baud rate of 19200 |

Word 15 contains the 8259 control port address (Programmable Interrupt Controller (PIC)). These can be:

| | |
|---|---|
| %0140 | mother board PIC control port A address |
| %0840 | expansion board PIC control port A address |

Word 16 contains the SEOI (Specific End Of Interrupt) command to be issued before exiting the interrupt routine. The SEOI is calculated using the formula:

SEOI = %C0 + (2* IRn)

where IRn is an interrupt routine number from 0 - 7.

The RS-232 SEOI's are the following:

| | |
|---|---|
| %00C6 | master 8259A PIC SEOI for IR3 (tty mother) |
| %00CE | master 8259A PIC SEOI for IR7 (expansion) |
| %00C0 | slave 8259A PIC SEOI for IR0 (port 1) |
| %00C4 | slave 8259A PIC SEOI for IR2 (port 2) |

## M20 - RS-232-C DEVICE PARAMETER TABLE

The following table gives all the M20 interrupt assignments.

---

Master 8259A PIC Mother Board Interrupt Assignments:

| | | | |
|---|---|---|---|
| IR0: | Floppy Disc Controller | | |
| IR1: | External Daisy Chain Request | | (potentially a slave 8259A) |
| IR2: | External Daisy Chain Request | | (potentially a slave 8259A) |
| IR3: | RxD: DTE TTY/Remote | 8251A | |
| IR4: | RxD: keyboard | 8251A | |
| IR5: | TxD: DTE/TTY/Remote | 8251A | (not used) |
| IR6: | Parallel 8255A PC0 or PC3 | | |
| IR7: | External Daisy Chain Request | | (used w/ RS-232 Expansion Board) |

Slave 8259A PIC Expansion Board Interrupt Assignments:

| | | | |
|---|---|---|---|
| IR0: | RxD: DTE/TTY port 1/Remote | 8251A | |
| IR1: | TxD: DTE/TTY port 1/Remote | 8251A | (not used) |
| IR2: | RxD DTE/TTY port 2/Remote | 8251A | |
| IR3: | TxD: DTE/TTY port 2/Remote | 8251A | (not used) |
| IR4: | grounded (not used) | | |
| IR5: | grounded (not used) | | |
| IR6: | grounded (not used) | | |

---

Words 17 and 18 contain the masks relative to the interrupt levels. The mask values are the following:

---

8259A PIC Interrrupt Assignments (by bit with data bus shift):

| | |
|---|---|
| %0100 | IR7 interrupt mask |
| %0080 | IR6 interrupt mask |
| %0040 | IR5 interrupt mask |
| %0020 | IR4 interrupt mask |
| %0010 | IR3 interrupt mask |
| %0008 | IR2 interrupt mask |
| %0004 | IR1 interrupt mask |
| %0002 | IR0 interrupt mask |

---

# I ASCII CODE

## ASCII CODE

This table shows decimal, hexadecimal, and binary representation of the ASCII code. (Boxed characters are different on national keyboards.)

| a | b | c | d | a | b | c | d | a | b | c | a | b | c |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 00 | 0000 0000 | NUL | 64 | 40 | 0100 0000 | $\boxed{@}$ | 128 | 80 | 1000 0000 | 192 | C0 | 1100 0000 |
| 1 | 01 | 0000 0001 | SOH | 65 | 41 | 0100 0001 | A | 129 | 81 | 1000 0001 | 193 | C1 | 1100 0001 |
| 2 | 02 | 0000 0010 | STX | 66 | 42 | 0100 0010 | B | 130 | 82 | 1000 0010 | 194 | C2 | 1100 0010 |
| 3 | 03 | 0000 0011 | ETX | 67 | 43 | 0100 0011 | C | 131 | 83 | 1000 0011 | 195 | C3 | 1100 0011 |
| 4 | 04 | 0000 0100 | EQT | 68 | 44 | 0100 0100 | D | 132 | 84 | 1000 0100 | 196 | C4 | 1100 0100 |
| 5 | 05 | 0000 0101 | ENQ | 69 | 45 | 0100 0101 | E | 133 | 85 | 1000 0101 | 197 | C5 | 1100 0101 |
| 6 | 06 | 0000 0110 | ACK | 70 | 46 | 0100 0110 | F | 134 | 86 | 1000 0110 | 198 | C6 | 1100 0110 |
| 7 | 07 | 0000 0111 | BEL | 71 | 47 | 0100 0111 | G | 135 | 87 | 1000 0111 | 199 | C7 | 1100 0111 |
| 8 | 08 | 0000 1000 | BS | 72 | 48 | 0100 1000 | H | 136 | 88 | 1000 1000 | 200 | C8 | 1100 1000 |
| 9 | 09 | 0000 1001 | HT | 73 | 49 | 0100 1001 | I | 137 | 89 | 1000 1001 | 201 | C9 | 1100 1001 |
| 10 | 0A | 0000 1010 | LF | 74 | 4A | 0100 1010 | J | 138 | 8A | 1000 1010 | 202 | CA | 1100 1010 |
| 11 | 0B | 0000 1011 | VT | 75 | 4B | 0100 1011 | K | 139 | 8B | 1000 1011 | 203 | CB | 1100 1011 |
| 12 | 0C | 0000 1100 | FF | 76 | 4C | 0100 1100 | L | 140 | 8C | 1000 1100 | 204 | CC | 1100 1100 |
| 13 | 0D | 0000 1101 | CR | 77 | 4D | 0100 1101 | M | 141 | 8D | 1000 1101 | 205 | CD | 1100 1101 |
| 14 | 0E | 0000 1110 | SO | 78 | 4E | 0100 1110 | N | 142 | 8E | 1000 1110 | 206 | CE | 1100 1110 |
| 15 | 0F | 0000 1111 | SI | 79 | 4F | 0100 1111 | O | 143 | 8F | 1000 1111 | 207 | CF | 1100 1111 |
| 16 | 10 | 0001 0000 | DLE | 80 | 50 | 0101 0000 | P | 144 | 90 | 1001 0000 | 208 | D0 | 1101 0000 |
| 17 | 11 | 0001 0001 | $DC_1$ | 81 | 51 | 0101 0001 | Q | 145 | 91 | 1001 0001 | 209 | D1 | 1101 0001 |
| 18 | 12 | 0001 0010 | $DC_2$ | 82 | 52 | 0101 0010 | R | 146 | 92 | 1001 0010 | 210 | D2 | 1101 0010 |
| 19 | 13 | 0001 0011 | $DC_3$ | 83 | 53 | 0101 0011 | S | 147 | 93 | 1001 0011 | 211 | D3 | 1101 0011 |
| 20 | 14 | 0001 0100 | $DC_4$ | 84 | 54 | 0101 0100 | T | 148 | 94 | 1001 0100 | 212 | D4 | 1101 0100 |
| 21 | 15 | 0001 0101 | NAK | 85 | 55 | 0101 0101 | U | 149 | 95 | 1001 0101 | 213 | D5 | 1101 0101 |
| 22 | 16 | 0001 0110 | SYN | 86 | 56 | 0101 0110 | V | 150 | 96 | 1001 0110 | 214 | D6 | 1101 0110 |
| 23 | 17 | 0001 0111 | ETB | 87 | 57 | 0101 0111 | W | 151 | 97 | 1001 0111 | 215 | D7 | 1101 0111 |
| 24 | 18 | 0001 1000 | CAN | 88 | 58 | 0101 1000 | X | 152 | 98 | 1001 1000 | 216 | D8 | 1101 1000 |
| 25 | 19 | 0001 1001 | EM | 89 | 59 | 0101 1001 | Y | 153 | 99 | 1001 1001 | 217 | D9 | 1101 1001 |
| 26 | 1A | 0001 1010 | SUB | 90 | 5A | 0101 1010 | Z | 154 | 9A | 1001 1010 | 218 | DA | 1101 1010 |
| 27 | 1B | 0001 1011 | ESC | 91 | 5B | 0101 1011 | $\boxed{[}$ | 155 | 9B | 1001 1011 | 219 | DB | 1101 1011 |
| 28 | 1C | 0001 1100 | FS | 92 | 5C | 0101 1100 | $\boxed{\backslash}$ | 156 | 9C | 1001 1100 | 220 | DC | 1101 1100 |
| 29 | 1D | 0001 1101 | GS | 93 | 5D | 0101 1101 | $\boxed{]}$ | 157 | 9D | 1001 1101 | 221 | DD | 1101 1101 |
| 30 | 1E | 0001 1110 | RS | 94 | 5E | 0101 1110 | ↑ | 158 | 9E | 1001 1110 | 222 | DE | 1101 1110 |
| 31 | 1F | 0001 1111 | US | 95 | 5F | 0101 1111 | — | 159 | 9F | 1001 1111 | 223 | DF | 1101 1111 |
| 32 | 20 | 0010 0000 | SPACE | 96 | 60 | 0110 0000 | $\boxed{`}$ | 160 | A0 | 1010 0000 | 224 | E0 | 1110 0000 |
| 33 | 21 | 0010 0001 | ! | 97 | 61 | 0110 0001 | a | 161 | A1 | 1010 0001 | 225 | E1 | 1110 0001 |
| 34 | 22 | 0010 0010 | " | 98 | 62 | 0110 0010 | b | 162 | A2 | 1010 0010 | 226 | E2 | 1110 0010 |
| 35 | 23 | 0010 0011 | $\boxed{\#}$ | 99 | 63 | 0110 0011 | c | 163 | A3 | 1010 0011 | 227 | E3 | 1110 0011 |
| 36 | 24 | 0010 0100 | $\boxed{\$}$ | 100 | 64 | 0110 0100 | d | 164 | A4 | 1010 0100 | 228 | E4 | 1110 0100 |
| 37 | 25 | 0010 0101 | % | 101 | 65 | 0110 0101 | e | 165 | A5 | 1010 0101 | 229 | E5 | 1110 0101 |
| 38 | 26 | 0010 0110 | & | 102 | 66 | 0110 0110 | f | 166 | A6 | 1010 0110 | 230 | E6 | 1110 0110 |
| 39 | 27 | 0010 0111 | ' | 103 | 67 | 0110 0111 | g | 167 | A7 | 1010 0111 | 231 | E7 | 1110 0111 |
| 40 | 28 | 0010 1000 | ( | 104 | 68 | 0110 1000 | h | 168 | A8 | 1010 1000 | 232 | E8 | 1110 1000 |
| 41 | 29 | 0010 1001 | ) | 105 | 69 | 0110 1001 | i | 169 | A9 | 1010 1001 | 233 | E9 | 1110 1001 |
| 42 | 2A | 0010 1010 | * | 106 | 6A | 0110 1010 | j | 170 | AA | 1010 1010 | 234 | EA | 1110 1010 |
| 43 | 2B | 0010 1011 | + | 107 | 6B | 0110 1011 | k | 171 | AB | 1010 1011 | 235 | EB | 1110 1011 |
| 44 | 2C | 0010 1100 | , | 108 | 6C | 0110 1100 | l | 172 | AC | 1010 1100 | 236 | EC | 1110 1100 |
| 45 | 2D | 0010 1101 | - | 109 | 6D | 0110 1101 | m | 173 | AD | 1010 1101 | 237 | ED | 1110 1101 |
| 46 | 2E | 0010 1110 | . | 110 | 6E | 0110 1110 | n | 174 | AE | 1010 1110 | 238 | EE | 1110 1110 |
| 47 | 2F | 0010 1111 | / | 111 | 6F | 0110 1111 | o | 175 | AF | 1010 1111 | 239 | EF | 1110 1111 |
| 48 | 30 | 0011 0000 | 0 | 112 | 70 | 0111 0000 | p | 176 | B0 | 1011 0000 | 240 | F0 | 1111 0000 |
| 49 | 31 | 0011 0001 | 1 | 113 | 71 | 0111 0001 | q | 177 | B1 | 1011 0001 | 241 | F1 | 1111 0001 |
| 50 | 32 | 0011 0010 | 2 | 114 | 72 | 0111 0010 | r | 178 | B2 | 1011 0010 | 242 | F2 | 1111 0010 |
| 51 | 33 | 0011 0011 | 3 | 115 | 73 | 0111 0011 | s | 179 | B3 | 1011 0011 | 243 | F3 | 1111 0011 |
| 52 | 34 | 0011 0100 | 4 | 116 | 74 | 0111 0100 | t | 180 | B4 | 1011 0100 | 244 | F4 | 1111 0100 |
| 53 | 35 | 0011 0101 | 5 | 117 | 75 | 0111 0101 | u | 181 | B5 | 1011 0101 | 245 | F5 | 1111 0101 |
| 54 | 36 | 0011 0110 | 6 | 118 | 76 | 0111 0110 | v | 182 | B6 | 1011 0110 | 246 | F6 | 1111 0110 |
| 55 | 37 | 0011 0111 | 7 | 119 | 77 | 0111 0111 | w | 183 | B7 | 1011 0111 | 247 | F7 | 1111 0111 |
| 56 | 38 | 0011 1000 | 8 | 120 | 78 | 0111 1000 | x | 184 | B8 | 1011 1000 | 248 | F8 | 1111 1000 |
| 57 | 39 | 0011 1001 | 9 | 121 | 79 | 0111 1001 | y | 185 | B9 | 1011 1001 | 249 | F9 | 1111 1001 |
| 58 | 3A | 0011 1010 | : | 122 | 7A | 0111 1010 | z | 186 | BA | 1011 1010 | 250 | FA | 1111 1010 |
| 59 | 3B | 0011 1011 | ; | 123 | 7B | 0111 1011 | $\boxed{\{}$ | 187 | BB | 1011 1011 | 251 | FB | 1111 1011 |
| 60 | 3C | 0011 1100 | < | 124 | 7C | 0111 1100 | $\boxed{\vert}$ | 188 | BC | 1011 1100 | 252 | FC | 1111 1100 |
| 61 | 3D | 0011 1101 | = | 125 | 7D | 0111 1101 | $\boxed{\}}$ | 189 | BD | 1011 1101 | 253 | FD | 1111 1101 |
| 62 | 3E | 0011 1110 | > | 126 | 7E | 0111 1110 | $\boxed{\sim}$ | 190 | BE | 1011 1110 | 254 | FE | 1111 1110 |
| 63 | 3F | 0011 1111 | ? | 127 | 7F | 0111 1111 | DEL | 191 | BF | 1011 1111 | 255 | FF | 1111 1111 |

GR Code 3987670 L (0)

Printed in Italy