RESEARCH MACHINES

ZASM

ASSEMBLER

For
Disc & Network
Systems

PN 11066

ZASM ASSEMBLER for DISC and NETWORK SYSTEMS

PN 11066

The policy of Research Machines Limited is one of continuous development and improvement of its products and services, and the right is therefore reserved to revise this document, or to make changes in the computer software it describes without notice. Research Machines Limited makes every endeavour to ensure the accuracy of the contents of this document but does not accept liability for the consequences of any error or omission.

The original labelled distribution disc is regarded as the only proof of purchase and must be produced in order to qualify for an update at a reduced rate. Keep it safe and always work from copies.

Additional copies of this publication may be ordered from Research Machines Limited at the address above. Please quote the title as given above.

If you would like to comment on any of our products or services please use the reply paid form provided at the end of this manual.

PREFACE

This publication describes the facilities of the Research Machines ZASM Assembler for Disc and Network Systems.

This manual is divided into seven chapters and two appendices. Chapter 1 describes how to run the ZASM Assembler. Chapter 2 describes the elements of the ZASM Assembly Language. Chapter 3 explains the construction and layout of an Assembly Language program to be processed by the ZASM Assembler. Chapter 4 defines the special pseudo operators available in the ZASM Assembler. Chapter 5 discusses the facility for program relocation provided by the ZASM Assembler. Chapters 6 and 7 describe the use of macros and ZASM Assembler directives respectively.

Appendix A is a listing of the instruction mnemonics (and their alternatives) available in ZASM. Appendix B gives the error codes displayed by ZASM when errors are encountered in assembler language source programs.

Related Publications

Users of the ZASM Assembler should also have access to the following publications available from Research Machines:

380Z/480Z Machine Language Programmers Guide, PN 11068

This publication is an introduction to assembler-language programs and/or Research Machines Disc or Network Systems.

MOSTEK Assembly Language Programming Manual, PN 11069

This publication describes the facilities available to the assembler-level programmer using the Z80A microprocessor.

Digital Research CP/M 2.2 Interface Guide, PN 11092

This publication describes how the operating system facilities of CP/M may be called from a ZASM assembler language program.

This publication also refers to the CP/M Operating System Manual, published by Digital Research and available from Lifeboat Associates Ltd.

CONTENTS

CHAPTER 1

GETTING STARTED

Introduction

This is a reference manual for the ZASM Assembler running on Research Machines microcomputers. It is not intended as a teaching manual, nor does it attempt to describe machine code. The Machine Language Programming Guide For 380Z and 480Z provides an introduction to Z80 programming.

For further instructional books on Z80 machine language programming we would suggest contacting either your local bookseller or:

LP Enterprises
8-11 Cambridge House
Barking
Essex.

The ZASM Z80 Assembler translates programs written in the Zilog mnemonic assembly language for the Z80 into absolute (or relocatable) object code.

The object code is output in industry standard (Intel) hexadecimal format, or in the case of relocatable code, industry standard (Microsoft) relocatable format.

ZASM will also produce an annotated listing of the source program. Either the object output or the listing or both may be suppressed.

The source program input to ZASM is normally prepared using RML's text editor TXED.

The instruction mnemonics and the hexadecimal representation of the binary codes into which the assembly program is converted are listed in Appendix A. For a discussion of each of the Z80 instructions refer to either of the following publications:

Z80 Assembly Language Programming Manual (Zilog Inc.)
Z80 Programming Manual (Mostek Corp.)

and it is suggested that you have one or the other of these available. (Copies can be obtained from Research Machines Limited and elsewhere.)

ZASM directives (most of which specify the format of the source program listing) are described in chapter 7.

The ZASM Macro facility (which permits a section of source code to be reproduced in any area of the program simply by mention of its associated name) is described in chapter 6.

The program relocation facility (permitting any given relocatable program to be used in conjunction with any other relocatable program) is described in chapter 5. Relocatable format allows the program sections to be assembled independently of a specific address and sufficient information is also held to fix them to a particular absolute address by the Linker. Relocatable modules must be linked before they can be executed. Only absolute code can be executed.

Pseudo-opcodes which specify actions required of the assembler and define data, but do not themselves normally generate executable code, are described in chapter 4.

The constituent elements of the ZASM Assembly language are described in chapter 2 and program construction and layout discussed in chapter 3.

'Debugging' or removing the error in an assembled program is a far simpler task when the contents of memory locations and registers can easily be changed and the program can be executed one instruction at a time (known as 'single-stepping'). These facilities are provided by the 'Front Panel' of the computer. The term 'Front Panel' refers to a mode of operation in which the contents of specified memory locations and processor registers are displayed on the screen and can be modified by input from the keyboard. The Front Panel and its operation are described in the Machine Language Programming Guide and other Research Machines publications.

The remainder of this chapter explains how to assemble and run a program using ZASM and uses a short assembly language program DEMO.ZSM, which is supplied on the distribution disc, as an example. You should copy the files ZASM.COM (the assembler) and DEMO.ZSM onto your system disc before working through the section (the file copying procedure is described in the System User Guide).

Assembling an absolute program

A program is assembled by typing ZASM followed by the name of the program (with a space in between). The command:

```
A>ZASM DEMO [RETURN]
```

loads the assembler and instructs it to assemble the contents of the file DEMO.ZSM. If you try the above command you will find, after a period of disc activity during which some messages appear on the console screen, that control returns to CP/M. If you now type DIR, you will find that a new file, DEMO.HEX, has been created. DEMO.HEX contains the output of the assembly process. This file is used in further processes to be described below.

Changing disc units

The above example assumed that the source program file DEMO.ZSM was on the disc in drive A and the object file DEMO.HEX was written onto the same disc. ZASM uses the extension file name (also called 'file type') in a rather special way as commands to the assembler, giving the drive names for the input and output files.

For example:

A>ZASM DEMO.BC RETURN

specifies that the source program DEMO.ZSM is on the disc in drive B and that DEMO.HEX should be written onto the disc in drive C. Note that the primary file name DEMO is common to both files, and that the extension file name of the source must be ZSM. Because the extension file name of the source file is assumed, and the one typed is used for other purposes, it is never necessary to type the extension name .ZSM in this context, and indeed to do so will provoke a BDOS error message.

Producing a listing file

If a third logical disc drive name is specified in the position of the extension file name, for example, if ZASM is invoked by the command:

A>ZASM DEMO.BCD RETURN

an additional file is produced which contains an annotated listing of the program. This file has the extension file name PRN. The example above produces a file called DEMO.PRN on drive D. This file contains a listing and can be printed and kept as a permanent record of the program, and is often useful for checking out programs.

Listing on the Console or Printer

As special cases the logical disc drive names X and P can be used to send the listing file to the console screen or to the printer respectively. Examples of these are:

A>ZASM DEMO.AAX RETURN to produce the listing on the screen
A>ZASM DEMO.AAP RETURN to produce the listing on the printer

In both cases the source file DEMO.ZSM should be on the disc in drive A and the .HEX file is written onto this disc also. Note that a printer option must have been selected in order to send the listing to the printer - see the System Information File for details.

Suppressing Object or Listing Output Files

As further special cases, ZASM allows the symbol 'Z' to be used instead of logical disc drive names in the extension file name, to suppress either or both of these output files. For example:

```
A>ZASM DEMO.AZA [RETURN] to suppress the .HEX file
A>ZASM DEMO.AAZ [RETURN] to suppress the .PRN file
A>ZASM DEMO.AZZ [RETURN] to suppress both files
```

The last .AZZ is useful to obtain a quick check for errors, which are listed on the screen, without producing the other types of output file.

Running the Assembled Program

The object output file produced by the assembler is in a hexadecimal ASCII format (known as Intel Hex) and needs to be converted into a memory image (.COM) file before it can be executed. Programs under CP/M (.COM files) are run in the Transient Program Area (TPA) and have a start address of 100 hex. The CP/M utility LOAD.COM can be used to convert an object file (.HEX) into an executable memory image (.COM file). To convert the example object file DEMO, enter:

```
A>LOAD DEMO [RETURN]
```

Finally enter:

```
A>DEMO [RETURN]
```

To run the program.

An alternative, somewhat more complex, method of converting a .HEX file to a .COM file involves the use of the DDT utility program and the CP/M SAVE command. DDT is a CP/M utility for debugging Machine Code programs and provides similar facilities to RML's Front Panel, but is designed for use with Intel 8080 mnemonics. For example, the file DEMO.HEX can be converted to the file DEMO.COM by the following sequence:

```
A>DDT DEMO.HEX [RETURN]
DDT VERS 1.4
NEXT PC
0130 0100
^C                          (type CTRL/C, without RETURN)

A>SAVE 1 DEMO.COM [RETURN]
```

The advantage of this second method is that it allows a number of .HEX files to be merged together to form a single program, and it allows access to the program via Front Panel after loading but before (or during) running, for debugging purposes. It also allows a start address other than 100 hex.

The LOAD and DDT utilities are described more fully in the CP/M Operating System Manual (Digital Research ™)

Before reading the next chapter, it is recommended that you experiment with the commands described above in order to run the demonstration program DEMO.COM, and produce a listing on your screen or your printer (if you have one).

Assembling a relocatable program

Assembling a relocatable program follows much the same pattern as assembling an absolute program. The assembler is invoked by a command of the form

A>ZASM DEMOR.BCD RETURN

as before. Here, the source (DEMOR.ZSM) is taken from drive B, the assembled object code is sent to drive C and a listing file is sent to drive D. As before, the object file may be suppressed by sending it to 'drive' Z, and the listing file may be redirected or suppressed by specifying 'drives' X, P or Z.

The object output so produced differs from absolute code by being written in the industry standard Microsoft format in a file having .REL as its filename extension.

Running the assembled program

Link the program by running a Microsoft compatible linker program such as L80 (as provided with FORTRAN by RML).

This is done by typing

A>L80 DEMOR, MSG, DEMOR/N/E RETURN

this links the files DEMOR.REL and MSG.REL (the latter obtained by assembling MSG.ZSM) and directs the output to DEMOR.COM. The program may then be run by typing:

A>DEMOR RETURN

CHAPTER 2

ELEMENTS OF THE ZASM ASSEMBLY LANGUAGE

This chapter describes the elements from which assembly language statements are constructed.

An assembly language statement consists of a line of text, terminated by a carriage return character (line feeds are ignored). Within each statement the characters are grouped into 'words' or 'tokens', separated by one or more space or TAB characters or by the punctuation characters colon, comma, and semicolon. The tokens can be classified into:

a) Symbols, which commence with a letter
b) Numbers, which commence with a digit
c) Strings, which are enclosed in single quotes
d) Comments, which commence with a semicolon
e) Local symbols, which commence with a number and end with a dollar sign.

Symbols

A symbol consists of a group of characters drawn from the sets A to Z, a to z, 0 to 9, period (.), underline, question mark and commercial 'at' sign (@). The first character must not be a digit, except in the case of local symbols (described below). Examples are:

```
START LOOP1 .OUTC LDIR  INC
```

Symbols may be of any length. However, the assembler only takes into account the first seventeen characters, and long symbols such as RIDICULOUSLYLONGNAME1 and RIDICULOUSLYLONGNAME2, which differ only after the 17th character, are considered identical. By default, the upper and lower case character sets are not considered distinct. Thus, START and Start are considered to be identical. This default setting can be changed with the *U directive (see chapter 7).

Symbols are divided into permanent symbols, those known to ZASM before it starts to process a program, and user defined symbols, which are made known during the course of assembly by their appearance in a symbol definition statement or as a label.

Permanent symbols and most user defined symbols may not be redefined. This means, for example, that a register name cannot be used as a label or in an EQU statement (q.v.). The statement:

```
HL:   ADD   HL,DE
```

gives an 'R' error, indicating the improper use of the register name HL as a label. It would be perfectly proper to use the label .HL .

It is possible to redefine some user symbols. If a symbol is assigned a value with either the ':=' or DEFL pseudo-operators, rather than = or EQU or QUERY, it may be redefined by further := or DEFL statements.

User-Defined Symbols

User-defined symbols are useful because they allow the programmer to refer to a numeric quantity in a meaningful way. For example, the symbol CR can be used to stand for the ASCII carriage return character whose value is 0D hex. The statement LD A,CR is much clearer that LD A,0DH, or LD A, 13D.

In addition a numeric quantity which is likely to change during the course of program development can be defined once at the start of the program and thereafter referred to symbolically. When the time comes to alter the value, only the definition needs to be changed, rather than having to hunt through the program source wondering whether an occurrence of that value should be changed or not.

User symbols are defined by their appearance in an EQU, QUERY, DEFL, =, or := statement. EQU and = (which are synonymous) define a constant value, which may not be changed thereafter. DEFL and := define a variable, which may be redefined as often as desired. QUERY is similar to EQU except that it takes a value from the console keyboard, after issuing a prompt (see chapter 4). Examples are:

```
CR          =           0DH
LF          EQU         0AH
SYMBOL      :=          4
SYMBOL      DEFL        SYMBOL + 1
VERSION     QUERY       'Which version'
```

It is also possible to introduce user symbols which aid in relocation and macros. These are described in chapters 5 and 6 respectively.

If, on the second pass through the assembly process, a symbol does not have a value associated with it, it will be flagged as undefined, giving a >U error.

Labels

A label is a special case of a user-defined symbol which becomes defined by its appearance at the beginning of a statement, followed by a colon. Labels may be preceded in a statement only by another label. The value assigned to a label is the address of the first byte of code or data generated by the statement, or, in the case of statements which generate no code or data, the address of the next such byte. Labels associated with the pseudo-operators ORG and DEFS are exceptions, being assigned the address of the next byte which would be generated had there not been an ORG or DEFS.

Local Symbols

Local symbols are a special form of symbol which take the form of a (decimal) number in the range 1 to 255 followed by a dollar ($) character. Local symbols are used as labels. Unlike normal labels, however, local symbols have limited scope and may not be referred to past the definition of a conventional label. Thus, the fragment:

```
        JP      1$
FRED:   LD      A,1
1$:     LD      B,A
```

may not behave as expected because FRED is defined between the use and definition of the symbol 1$. This means that it is possible to give 1$ a completely different value elsewhere in the program, provided that a non-local label occurs in between.

The main use of local symbols in practice is as targets for jumps within a routine, with a non-local symbol used for the entry point to the routine.

Non-local symbols may be called global symbols in other assemblers. There are, unfortunately at least two other meanings associated with the word global often used in assemblers, associated with relocation. The use of global to mean non-local is therefore somewhat ambiguous, and perhaps should be avoided.

Numbers

A number starts with a digit and consists solely of the digits 0 to 9 and, if expressed in hexadecimal notation, the letters A to F. Note that a number must begin with a decimal digit to be recognized as such (a zero will suffice), even though some hexadecimal numbers have a letter as the leading digit, e.g. the hexadecimal number FFFF should be expressed as 0FFFFH. A single character added to the end of the number is used to indicate the 'base' or 'radix' in which it should be evaluated. Valid characters are:

B	base 2 (binary)
O or Q	base 8 (octal)
D or .	base 10 (decimal)
H	base 16 (hexadecimal)

If there is no trailing indicator,the number is evaluated according to the current default radix. This is initially set to 10 but may be reset to 2, 8, 16, or back to 10 by the pseudo-operators RADB, RADO, RADH, and RADD respectively. However, there is a problem with using RADH: some 'valid' hexadecimal numbers end with the characters B and D, so, when the default radix is set to 16, binary and decimal numbers (indicated by a trailing B or D) will be treated erroneously as hexadecimal.

One widely held belief is that the default radix used by the assembler should be the same as that used by the person whose task it is to

understand the program. Since almost all people use decimal much more fluently than any other radix, the default radix should therefore be decimal, and any quantity which is expressed in some other radix should be flagged with an appropriate trailing character. This overcomes the problem with RADH, which is only included to maintain compatibility with previous versions.

The Dollar Symbol

The dollar symbol is a special numeric symbol which represents the current assembly program counter. The assembler treats it as a number whose value is the address of the first byte of code or data to be generated by the current statement. If the statement does not generate any code or data, the address of the next such byte is assigned. The dollar symbol is often useful in working out the length of a table. For example:

```
TBL:    DEFB    1
        .
        .
        DEFB    7
LEN     EQU     $-TBL
```

LEN becomes the difference between the address of the next byte to be generated and the address of the start of the table and is thus equal to the length of the table in bytes.

$ takes on the mode of the current program counter. This is of relevance only when ZASM is generating relocatable code. See chapter 5 for details.

String and Character Constants

A character string consists of a series of characters enclosed in single quotation marks ('). Strings of more than one character can only be used as the argument to the DEFM or QUERY pseudo-operators (see chapter 4). Within the string, the character up arrow (↑) has special significance. Rather than generating a data byte it governs what will happen to the following character:

↑' generates '
↑↑ generates ↑

In all other cases, the most significant bit of the next character byte is set. Thus, 'A' generates 41 hex, while '↑A' generates C1 hex. This can be useful for marking the end of a string.

A string consisting of a single character is called a character constant. It generates a single data byte and can be used wherever an 8-bit number is allowed. The 'up arrow' construction can be used in a character constant.

ZASM does not support 16-bit (double character) character constants.

Comments

A comment can form part of any statement. Comments begin with a semicolon and continue with any sequence of characters up to the terminating carriage-return. Comments appear on the listing, but are otherwise completely ignored by the assembler. Their only use is to aid human understanding of the program.

Much has been written about the use of comments within an assembly language program. Certainly, they should be used frequently, with about as much (or more) commentary as program text. A good plan is to give a block of comment preceding each block of code. The 'one comment per line' school of thought is not perhaps the best, since many of the comments so generated convey no useful information and merely serve to clutter up the program. Comments should serve to describe what the program is doing (or rather should be doing according to the author) and how it works. A comment such as:

```
LD A,0     ;Load A register with 0
```

is useless, while:

```
LD A,0     ;CLEAR PAGE COUNTER
```

is of far greater value.

Expressions

It is often useful to be able to combine the various numeric and symbolic elements. User-defined symbols (including labels), numbers, character constants, and the dollar symbol can be combined in an expression which consists of any reasonable number of such items separated by the operators shown in the table below.

Operators of highest priority are evaluated first, with operators of equal priority being evaluated from left to right. This order of operations may be altered using parentheses in the conventional way. However, an expression may not begin with a left parenthesis, (except in data definitions such as "DEFB expression" and "expression, expression") since parentheses are also used to represent indirection within the assembler.

The operators of priority 3 (=, EQ, NE, >, GT, <, LT, GE, LE) produce a Boolean result, 0 for FALSE and -1 (0FFH) for TRUE.

The arithmetic operators (*, /, +, -, MOD) treat their operands as signed 16-bit numbers. Overflow is ignored.

The logical operators (SHR, SHL, AND, OR, XOR) view their operands as 16-bit quantities.

The mode of an expression depends both on the modes of the operands and on the operators within the expression. The rules are discussed in chapter 5, since the mode of an object is only of concern when ZASM is generating relocatable code.

Note that in the case of "COND expression" any non zero value of the expression is taken as true.

Operator	Priority	Effect
+	6	Unary Plus
-	6	Unary Minus
*	5	Multiplication
/	5	Division
MOD	5	Remainder
SHR	5	Logical shift right
SHL	5	Logical shift left
+	4	Addition
-	4	Subtraction
=	3	Equal
EQ	3	Equal
NE	3	Not equal
>	3	Greater than
GT	3	Greater than
<	3	Less than
LT	3	Less than
GE	3	Greater than or equal
LE	3	Less than or equal
AND	2	Logical and
OR	1	Logical or
XOR	1	Logical exclusive or

CHAPTER 3

PROGRAM CONSTRUCTION AND LAYOUT

This Chapter describes the construction of an assembly language program. It is based on a 'guided tour' of the program DEMO.ZSM, which is supplied on the distribution disc. The program is listed below, although if you have followed the suggestions in chapter 1, and have a printer, you will have a copy of this already.

Full details of all the constructs mentioned here are given in chapters 4 and 7.

```
                        0001*H ZASM DEMONSTRATION
                        0002 ; DEMO.ZSM - Print a message on the
                        0003 ; console to validate zasm
                        0004
0005 =                  0005 BDOS    EQU    5
0009 =                  0006 PRBUF   EQU    9
000D =                  0007 CR      EQU    0DH
000A =                  0008 LF      EQU    0AH
                        0009
0100                    0010         ORG    100H      ;Transient area
                        0011
0100   0E09             0012 START:  LD     C,PRBUF   ;Print buf function
0102   110901           0013         LD     DE,MSG    ;Address of msg
0105   CD0500           0014         CALL   BDOS      ;Output message
0108   C9               0015         RET              ;Return to CCP
                        0016
0109   5A41534D         0017 MSG:    DEFM   'ZASM welcomes you'
011A   20746F20         0018         DEFM   ' to Z80 programming'
012D   0D0A24           0019         DEFB   CR,LF, '$'
                        0020
0100                    0021         END    START

Symbols:

BDOS   0005 CR          000D    LF    000A    MSG    0109    PRBUF    0009
START  0100

No errors detected
```

Elements of a ZASM Assembly Language Program

Input to the assembler consists of a series of lines of text of the general form:

```
label:    instruction          ;comment
```

Any or all of these elements may be absent, and indeed a completely blank line is a valid statement. However, a label (if present) must come first, and the comment (if any) last. For example, line 12 of DEMO is:

```
label     instruction          comment

START:    LD   C,PRBUF         ;Print buf function
```

This statement causes an instruction to be assembled to load the C register with the value of the symbol PRBUF (which has already been defined, in line 6 of DEMO, to be equal to 9). The line appears on the assembly listing as:

```
address   generated      line
          code           number

0100      0E09           0012 START: LD   C,PRBUF    ;Print buf function
```

Code bytes 0E and 09 are generated and will eventually be stored in memory at addresses 100 and 101 hex. The label START is assigned the address of the first code byte, in this case 100 hex (the normal start address of CP/M programs). The comment is listed, but otherwise ignored. Note that the values the assembler assigns to symbols and labels, such as PRBUF and START, are listed in a table at the end of the assembly listing and, in the case of symbols, at the start of the line on which they are defined.

The next line of the program, (LD DE,MSG), causes the three byte instruction 'load the register pair DE with the address of the label MSG' to be assembled (01 09 01). Notice that the second and third bytes which represent the address of MSG (109 hex) are stored in reverse order. This is a consequence of the way the Z80 processor deals with 16-bit quantities: such quantities are stored with the low-order byte first, then the high-order byte.

The rest of the program consists of the CALL instruction in line 14 (again using a 16-bit address) and finally the RET instruction in line 15. Taken as a whole, the program loads the C register with the function code for CP/M to print the message whose address is in DE, calls the operating system at address 5 to perform the function, and finally returns to where it was called from. In this example the RET instruction returns control to CP/M without reloading it.

Take extreme care in using the stack and then attempting to return from a subroutine to an address stored on it. In any non-trivial program a better technique is to save the original stack pointer in memory and to load the stack pointer with the top of your own stack area. The stack should have at least 64 bytes of free space more than that required within your code and more still if EMT's and escape sequences are to be used.

Example:

```
START:      LD (SYSTEM.STACK.POINTER),SP      ;Preserve System
            LD SP,TOP.MY.STACK                ;Stack Pointer
             .
             .  User Code
             .
EXIT:       LD  SP, (SYSTEM.STACK.POINTER)    ;Restore System
            RET                               ;Stack Pointer

STACK:      DEFS My requirements + 64         ;Preserve enough
                                              ;Stack Space
TOP.MY.STACK:
            END
```

At the end of your program you can either perform a jump (JP 0) to address zero or restore the original system stack pointer and return to the CCP. The problems caused by overflow, i.e. using more stack than is available, are often very intermittent and difficult to track down. Hence the importance of reserving a large enough stack cannot be over-emphasised.

The instructions DEFM and DEFB in lines 17 to 19 of the example program generate data rather than Z80 machine code instructions. DEFM (define message) generates an ASCII code byte for each character of its argument string. Only the first four bytes are shown on the listing but you can verify from the address of the next line that all the characters between the quotes have been processed. DEFB (define byte) generates a byte for each of its arguments, multiple arguments being separated by commas. Thus line 19 assembles to 0D 0A 24, corresponding to the values of the symbols CR and LF and the ASCII code for the dollar character. (The BDOS print function expects a dollar character to terminate the message). Note that to the processor these data bytes are indistinguishable from the executable code produced by the assembly of opcodes.

Lines 5 to 8 define the symbols used in the program. Note their form:

```
symbol    EQU  value
```

and in particular that there is no colon after the symbol name, as in:

```
PRBUF     EQU  9
```

The value which is to be assigned to the symbol is often a number, although it can also be an expression and may contain other symbols. Numbers start with a digit. If the number is in hexadecimal form (base 16), this is indicated by following it with 'H', as in:

```
CR        EQU       0DH       ;Equivalent to 13 decimal
```

The ORG statement (line 10) sets the first address of the program (the program 'origin') to its argument, 100 hex. The END statement (line 21) marks the end of the program. The END statement may also include a label ('START' in the example program). Some programs, such as DDT, use the label to specify the object code execution start address.

The instruction mnemonics in the program can be divided into those which produce executable code (Z80 instructions), referred to as opcodes, and those which generate data or tell the assembler what to do, which are called pseudo-opcodes. In fact, the only statement in DEMO which neither contains an operator or pseudo-opcode, nor consists solely of a blank line or comment, is line 1. Lines such as this, which start with an asterisk, are called assembler directives. The directive *H (*HEADING) supplies a running title for the listing, and in fact most of the ZASM directives control some aspect of how the listing is produced (see Chapter 7).

Program layout

ZASM permits a free-format organization of the source program, so it is worth establishing a convention and keeping to it. It is relatively easy to lay out an assembly language program in a way which is clear and readable even when returning to the program after a long period. Two conventions will be suggested. The first, which is that used for DEMO, is to separate the label, operator, and argument with TAB characters, and the argument and comment with two tab characters unless the argument is longer than eight characters. (If your keyboard has no TAB key, a tab can usually be generated by typing CTRL/I). Thus, line 12 of DEMO could have been keyed in as:

```
START: [TAB] LD [TAB] C,PRBUF [TAB] [TAB] ;Print buf function
```

where [TAB] stands for the CTRL/I key combination. This convention is particularly suitable if you have a printer or 80-column screen. Where listings must often be viewed on the console, you may prefer the shorter lines which result from replacing the TAB that follows the operator by a space, and keying only one TAB before the comment (although in this case listings containing labels of more than 6 characters are extremely untidy).

The purpose of either convention is to line up the various fields of each statement. The labels stand out clearly, as do the comments, and the code can be followed in a logical way.

The proposed behaviour of a program should be adequately described by means of comments. Even a well-written assembly language program can be somewhat obscure, especially if the algorithm is complex, so comment-free programs are almost impossible to follow. However, comments should be meaningful. For example, the comment in:

```
        SCF         ;Set carry flag
```

adds nothing useful and obscures any meaningful comments, whereas:

```
        SCF         ;Signal an error
```

says what setting the carry flag actually means. It is certainly not necessary to include a comment on every line.

CHAPTER 4

ZASM PSEUDO-OPCODES

This section describes all of the pseudo-opcodes that ZASM recognizes, although some, those concerned with relocation and macros, are given only very brief mention, with full explanations to be found in chapters 5 and 6 respectively. The use of pseud-ops in the definition of data and when reserving storage space is described at the end of this section.

Items enclosed in angle brackets in this text (e.g. <expression>, <symbol>) represent variable quantities. Items enclosed in square brackets represent optional parameters (see DEFB). Most of these items are defined in chapter 2, although those concerned with relocation are described in chapter 5.

ASEG ASEG

ASEG sets the location counter to the absolute memory segment. See chapter 5 for details.

COM COM

COM indicates that the output file is to be generated in absolute machine readable code, instead of HEX digits. It should appear before any code is generated. An origin of at least 100H should be specified and the code should never go backwards in origin, otherwise an incorrect COM file could be generated and a >O error will result.

Sections where no code is generated but the assembly program counter is advanced by, for example, DEFSs and ORGs, will be filled with zeros, except for DEFSs at the end of the program which will not be filled.

COMMON COMMON /[<block name>]/

COMMON sets the location counter to the specified COMMON block. See chapter 5 for details.

COND COND <expression>

COND starts a conditional assembly block. The expression is evaluated to yield a truth value (True<>0, False=0) If the expression is false, assembly is suspended. Assembly is resumed when either an ELSE or ENDC pseudo-opcode is encountered. If the expression is true, assembly continues, but an ELSE will cause the suspension of assembly until the matching ENDC. ENDC terminates the conditional segment. The expression must be defined before it is used and must not contain any forward references.

While assembly is suspended ZASM makes no checks with regard to syntax, and the skipped text need not form a legal block of assembly language. There may be at most one ELSE at any given conditional level. CONDs may be nested, to a maximum depth of 10 levels.

CSEG CSEG

CSEG sets the location counter to the code-relocatable memory segment. See chapter 5 for details.

DEFB DEFB <expression>[,<expression>...]

The arguments to DEFB (define byte) are expressions, which must be absolute and lie in the range -128 to 255. DEFB stores the values in successive memory locations. Adjacent commas result in a zero byte being generated, while a trailing comma is ignored. Examples are:

```
DEFB  -1        ;Generates FF hex
DEFB  3,4,,5,   ;Generates 03 04 00 05
```

The rather curious convention concerning stray commas arises from a historical accident involving automatically-generated programs. It is suggested that hand-written programs should ignore this feature.

DEFL <symbol> DEFL <expression>

DEFL is similar to EQU (q.v.) except that no error is generated if the symbol is later redefined. This pseudo-opcode is sometimes called SET in other assemblers. The := pseudo-opcode is synonymous with DEFL.

DEFM DEFM <string>

DEFM generates a sequence of bytes, similar to a DEFB, containing the characters of the string.

DEFS DEFS <expression>

DEFS reserves an area of memory. The value of the argument (which must be absolute) is added to the current location counter, reserving that many bytes of storage. The reserved storage is not initialized in any way unless the COM pseudo-opcode has been executed, in which case null bytes are written to the output file.

Note that the expression MUST be defined before it is used (and must not contain any forward references), or any subsequent labels will cause phase errors. A phase error occours when a label or variable has different values on each of the two passes made through the source code by the ZASM assembler when generating the object code.

DEFW DEFW <expression>[,<expression>...]

DEFW is similar to DEFB, except that it deals with 16-bit values, not 8-bit values. DEFW stores the values of the expressions, which are its arguments, in successive memory locations. Each word is stored with the low-order byte first.

DSEG DSEG

DSEG sets the location counter to the data-relocatable memory segment. See chapter 5 for details.

ELSE ELSE

ELSE resets the conditional assembly state. See COND.

END END [<expression>]

The END statement specifies the end of the program. The argument, or zero if it is absent, is used as the start address for the program.

ENDC ENDC

ENDC terminates a conditional assembly segment. See COND.

ENDM ENDM

ENDM terminates a macro definition. See chapter 6 for details.

EQU <symbol> EQU <expression>

EQU assigns to the symbol on the left of the EQU the value of the expression on the right. The expression can be of any mode, (absolute, code relocatable, etc) and the symbol takes that mode. The symbol may not subsequently be redefined to another value or mode. The trivial exception to this rule is that it is legal to 'redefine' it if the new value and mode are the same as the old.

The symbol "=" may be used instead of EQU.

EXTERNAL EXTERNAL <symbol>[,<symbol>...]

EXTERNAL tells the assembler that the specified symbols are defined in some other module (file) by the GLOBAL pseudo-opcode. The linker will patch in the addresses concerned. An external symbol may not be redefined (except as external).

GLOBAL GLOBAL <symbol>[,<symbol>...]

A symbol defined by GLOBAL is made available to other program modules at link time, to be used as EXTERNALs. GLOBAL symbols may be given values via EQU etc. or by their use as labels. See chapter 5 for details.

LIBRARY LIBRARY <symbol>[,<symbol>...]

The LIBRARY pseudo-opcode generates signals to the linker to search the specified libraries for definitions of undefined global symbols. For example, the statement:

```
LIBRARY GLIB
```

causes the linker to search the file GLIB.REL for global symbol definitions. See chapter 5 for details.

MACRO <symbol> MACRO <argument list>
MACRO <symbol> <any list>

MACRO defines a macro. See chapter 6 for details.

NAME NAME <symbol>

The NAME pseudo-operator sets the name of the current module, for use by the linking loader. By default the first six characters only are used, and if unspecified will be the first six characters of the file name.

ORG ORG <expression>

ORG is used to set the program origin. The current program counter is set to the value of the expression. The expression must be absolute or of the same mode as the current program counter. The expression must be defined before use and must not contain any forward references.

QUERY <symbol> QUERY <string>

QUERY is used to assign a value to the symbol from the keyboard. The string is displayed followed by a colon. ZASM then waits for an expression to be entered from the keyboard terminated by a carriage-return character. The expression is evaluated and, if legal, assigned to the symbol. If an invalid expression is entered, the prompt:

```
Bad input, try again:
```

is given and another expression should be entered. The statement is skipped if the symbol has already been defined, and the symbol may not be redefined in a subsequent EQU or DEFL statement.

RADB RADB

RADB sets the default radix to binary. Any number subsequently encountered not ending with an explicit radix indicator (B,O,Q,D, ., or H) is treated as a binary constant. The default radix is set to decimal at the start of each pass.

RADD RADD

RADD sets the default radix to decimal. Any number subsequently encountered not ending with an explicit radix indicator (B,O,Q,D, . or H) is treated as a decimal constant. The default radix is set to decimal at the start of each pass.

RADO RADO

RADO sets the default radix to octal. Any number subsequently encountered not ending with an explicit radix indicator (B,O,Q,D, ., or H) is treated as an octal constant. The default radix is set to decimal at the start of each pass.

RADH RADH

RADH sets the default radix to hexadecimal (Base Sixteen). Any number subsequently encountered not ending with one of the explicit radix indicators O, Q or H is treated as a hexadecimal constant. Any number encountered ending with B or D is treated, in its entirety, as a hexadecimal constant when RADH is in effect. The default radix is set to decimal at the start of each pass.

Note

When RADH is in effect the letters B or D at the end of a number are treated as part of the number and not as trailing redix indicators. (unlike previous versions of ZASM). Considerable care should be taken when using RADH.

Data Definition

Data may be defined by the pseudo-ops DEFB, DEFW and DEFM which store 8 bit, 16 bit and character string quantities respectively:

```
Byte(s)    Assembled  Pseudo-op  Expression

01                    DEFB       1
0002                  DEFW       0200H
53545249              DEFM       'STRING'
```

DEFB and DEFW (but not DEFM) may define several constants, if they are separated by commas:

```
01020304              DEFB       1, 2, 3, 4
04000300              DEFW       4, 3, 2, 1
```

Strings defined by DEFM may contain the 'escape' character UPARROW (not the same as ASCII ESC). This causes the most significant bit of the next character to be set:

```
41C1                  DEFM       'A↑A'
```

Exceptions:

1) Two uparrows assemble an uparrow:

```
5E                    DEFM       '↑↑'
```

2) Uparrow single quote assembles a single quote:

```
27                    DEFM       '↑''
```

An ASCII CHARACTER CONSTANT consists of a character string which assembles to a single byte. It may contain the uparrow construction:

```
41C15E27        DEFB       'A','↑A','↑↑','↑''
```

Free Format Data Definition

ZASM also allows data to be defined in FREE FORMAT. If the first non-blank character of a statement is one of:

0 - 9	Digit
+	Plus
-	Minus
'	Single Quote
$	Dollar
#	Hash
(	Left Parenthesis

the statement is considered to be defining data and will be assembled as though the character was preceded by DEFB, DEFW or DEFM. Hash (#) signals that the next constant is a 16 bit one and single quote (') that it is a character string. Several such quantities may appear in one statement and their types may be mixed:

```
        01          1           ;Assemble in 8 bits
        0100        #1          ;Assemble in 16 bits
        31          '1'         ;Assemble ASCII constant
        01010031    1,#1,'1'    ;Do all three
        ----        #$          ;Current Address (----) as word

000A=       TEN EQU OAH         ;Hexadecimal Constant
        0A  +TEN                ;Assemble Symbolic Byte
        0A  (TEN)               ;Ditto
```

The last two forms are necessary because free format cannot start with a letter. It may continue with one however:

```
        0A0A        10,TEN
```

Free format is often useful for table generation.

Reserving Storage

Storage space is reserved by the pseudo-op DEFS:

Address	Pseudo-op	Expression
0056	DEFS	100H
0156	NOP	

Note

DEFS and ORG may be followed by a symbol or symbolic expression. However this must be fully defined before the DEFS or ORG is encountered on pass 1. If not (and it is subsequently defined), the assembly program counter will

take on a different value for pass 1 and pass 2 and this will result in incorrect assembly, indicated by P errors for any subsequent labels. If the symbol or expression remains undefined during pass 2 it will give a U error. Error Codes are described in Appendix B.

CHAPTER 5

PROGRAM RELOCATION

Relocation is the name given to the process whereby the exact address of a section of assembly language code or data is deferred to the so-called linking phase. The benefit of this approach is that a large assembly language program can be divided into modules, each relatively autonomous. This reduces the complexity of the program very substantially. The number of errors should therefore be reduced.

A separate program, variously called a linker, linkage editor, or loader (and a variety of other names), is used to convert the output of the assembler (.REL files) to executable (.COM) files.

Simple Relocation

Simple relocation is conceptually straightforward. Contained within a typical assembly language program are various addresses, which must be changed if the program origin is changed. For example, the code:

```
        ORG         0
LAB:    .
        .
        JP          LAB
```

generates the code C3 00 00 for the JP instruction, but moving the origin to 2345H changes the code to C3 45 23. If the assembler is generating relocatable code, it flags every address which needs altering. The example therefore generates the code:

```
C3 (flag) 00 00
```

where (flag) indicates that the following address should have the program origin added to it. The output is now completely independent of the program origin. The linker makes a note of the program origin and adds that to any flagged address.

Note that the use of relative jumps can reduce substantially the number of such flagged addresses. In the limit there may be none, and such a program is described as position-independent. On many computers it is easy to write position-independent code, but the architecture of the Z80 makes it hard to write and often much less efficient than position-dependent code.

By default, ZASM generates absolute (non-relocatable) code. It can be made to generate relocatable code by specifying the pseudo-opcode:

```
CSEG
```

CSEG is a contraction of 'code-relocatable segment'. There are other types

of relocatable segment, which we'll cover later. It is possible to convert back to absolute by the 'absolute segment' pseudo-opcode:

```
ASEG
```

Note that the pseudo-opcode COM cannot be used in a program that contains a relocatable segment.

So far, we haven't actually made any progress, as all that we have done is complicate the assembler. However, there is a big advantage. Suppose that we have several modules each assembled with an origin of zero. The linker can merge these modules, adjusting the origin of each one to be just after the end of its predecessor. By this technique, independent sections of program can be loaded together, and a change means only that one section needs to be reassembled and then the program relinked. Under CP/M, the first module is normally loaded at 100 hex.

Globals and Externals

The simple relocation scheme outlined above is not tremendously useful, since the individual modules are completely separate, with no intermodule communication. The way round this is to specify in each module that certain symbols are GLOBAL or EXTERNAL. A global symbol is an otherwise normal symbol (which must be defined at some time during pass 1) which is flagged by the GLOBAL statement:

```
GLOBAL symbol [,symbol...]
```

ZASM outputs information to the object code file which includes the first six characters of the name of the symbol, and its value (which may be relocatable). This information is made available to any module which requires it by saying:

```
EXTERNAL symbol [,symbol...]
```

in that module. The symbols must not be otherwise defined in that module, but must be defined by a GLOBAL statement in some other module.

For example, suppose there exists in one module a routine called MYSUB which other modules need to access. We can say:

```
            GLOBAL MYSUB
            .
MYSUB:      .
            RET
```

to define it, and :

```
            EXTERNAL MYSUB
            .
            CALL     MYSUB
```

to access it.

Incidentally all relocating assemblers have some technique available which amounts to the same thing as the GLOBAL and EXTERNAL pseudo-opcodes described above. Unfortunately, there is little agreement as to what to call them, and one particularly widespread convention is to use PUBLIC for what we call GLOBAL, and GLOBAL for our EXTERNAL.

The Data-Relocatable Segment

Besides the code-relocatable and absolute segments described above, ZASM provides a data-relocatable segment. This is invoked by:

```
DSEG
```

and behaves just like CSEG in all respects. However, the advantage of having a separate segment in which to put data, rather than code, is that the linker can be directed to separate them. This is useful when the code is to be placed in read-only memory (ROM) and the data in read/write memory (RAM).

ZASM maintains completely independent program counters for the code and data-relocatable segments (and indeed for the absolute segment). If you leave a segment, and later return to it, the relevant code program counter will remain unchanged and further code will join on to the end.

Common Segments

The other type of segment allowed by ZASM is the COMMON segment, which, as its name implies, owes much to FORTRAN. It is, nevertheless, useful in assembly language programs. A common segment is entered by the statement:

```
COMMON /symbol/     or     COMMON //
```

and is left by another segment-defining pseudo-opcode.

The latter form (the slashes are mandatory) is called 'blank common.' The symbol (which, as in FORTRAN, is limited in length to six characters) is used to define which common segment to use. This means that COMMON is not just one segment, but many, each identified by name. (ZASM imposes the restriction that there can be no more than 255 COMMON segments in any module. However, in practice, it is very rare to use more than about two.) A major difference between COMMON and other segments is that each occurrence of the word COMMON sets the COMMON program counter to zero, and so multiple references to the same COMMON block overlay each other.

COMMON blocks are available freely to every module which references them. However, unlike EXTERNAL/GLOBAL, the individual variables within a block are accessed by their offsets from the base of the block, rather than by name.

An example might clarify this point:

Suppose that in module A we define:

```
        COMMON      //      ;Blank COMMON
WORD:   DEFS        2       ;Leave space for a 16-bit number
BYTE:   DEFS        1       ;Only 8 bits here
```

and in module B we define:

```
        COMMON      //      ;Blank COMMON
B1:     DEFS        1       ;First byte
B2:     DEFS        1       ;Second byte
B3:     DEFS        1       ;Third byte
```

In module A the blank COMMON area is divided into two areas, accessed by WORD and BYTE respectively. In module B, it is divided into three, accessed by B1, B2, and B3. B1 and B2 can be used to access the same storage as WORD, and B3 the same byte as BYTE. The possibilities for confusion with this type of construct are enormous. However, used carefully, COMMON is an exceedingly useful technique.

A given COMMON block need not be defined to be the same size in each of the modules which define it. However, linkers generally impose the restriction that the second and subsequent references to a COMMON block must not be to blocks larger than the first (defining) reference. Thus, the module with the largest size definition for each COMMON block must be linked first.

Expression Restrictions

There are several restrictions imposed on expressions which contain relocatable quantities. These are largely due to limitations in the linker. However, in practice, these restrictions largely forbid operations which are very rare (e.g. multiplying two labels together). Attempts to use a forbidden expression result in an E error.

An absolute quantity can be added to a quantity of any mode, the mode of the result being the mode of the non-absolute operand.

An absolute quantity can be subtracted from a quantity of any mode, and the mode of the result is the mode of that operand. Otherwise, the modes of both operands must be the same, and the result is absolute. Subtraction of COMMON items is valid only where both operands are in the same COMMON block, and an external variable may only be subtracted from itself or from a variable whose value is the sum of that external and an absolute.

For any other operator, both operands must be absolute, and the mode of the result absolute.

The linker may add to any address a single offset, if that address is relocatable. Denoting absolute values by A, A', and A'' and relocation offsets by r and r', an expression may therefore deliver either:

```
      A
```

or:

```
      A + r
```

Subtraction of relocatable quantities therefore amounts to

```
(A + r) - (A' + r) => (A - A') + (r - r) => A - A'
```

if they are in the same segment, but

```
(A + r) - (A' + r') => (A - A') + (r - r') => error
```

if they are in different segments. The limitation to one relocation offset applies to sub-expressions as well as to the entire expression, so that for example:

```
(A + r) + (A' + r) - (A'' + r)
```

gives an error, even though the result is A + A' + A'' + r, because after the first addition the result is not representable.

Note that it is not possible in ZASM to compare two relocatable quantities directly, even where they are in the same segment. The quantities must be subtracted and the result compared with zero.

Libraries

A library is a file which contains a number of assembled 'modules'. Each module typically contains some GLOBAL symbols. A linker can search a library for these definitions, and include relevant modules with the executable code. The LIBRARY pseudo-opcode:

```
LIBRARY libfile[,libfile...]
```

specifies one or more library files. The linker will search all of the specified libary files, in order, for definitions of undefined EXTERNAL symbols. It is normally used when the file makes reference to the contents of the library and reduces the amount of typing necessary to produce an executable file from the relocatable binary file.

A typical linker program is the one known as L80, provided by RML with the FORTRAN compiler.

CHAPTER 6

MACROS

The macro (from the Greek for "large") is a very powerful extension to assembly language - or indeed to a high-level language. In many ways, macros are used in a similar way to subroutine calls, in that both can be regarded as adding to the instruction set of the computer. However, there is one important difference: macros work by textual substitution, whereas a jump to a subroutine implies a dynamic linkage. Macros tend to be faster and more flexible, being assembled at each point they are invoked, whereas subroutines offer savings in program size, since they are assembled in the place they are defined.

For beginners, the macro presents a degree of complexity which can be confusing. It is therefore suggested that novices defer reading this chapter until they have a thorough grasp of the rest of this manual. However the experienced user will find the added benefits of macros an invaluable aid to program simplification and readability.

One of the problems with macros (or at least with descriptions of macros) is that realistic examples tend to use many of the features of macros and therefore tend to be somewhat complicated. This description attempts to explain macros in an incremental manner, covering the simplest features first, and going on to more complex matters later. This approach has the drawback that the examples tend to be unrealistic and useless in the early part of the chapter, and it is only towards the end that macros of any real benefit start to appear.

Parameter-less Macros

The simplest type of macro has no parameters. It is defined by a sequence of statements such as:

```
mymac   MACRO
        .
        .
        .
        ENDM
```

where the dots represent any piece of text (which need not form valid assembly code, although of course the final product must be valid to avoid errors), and mymac is the name of the macro. The code between the MACRO and ENDM statements is known as the macro body. The term 'replacement text' is also occasionally used, although in this document it is given slightly different meaning which we will come to later.

The body of the macro is inserted into the source stream more or less unmodified. However, any macro invocations within the macro body are

expanded. The advantage of this scheme, as opposed simply to copying the macro body with the aid of a text editor, is that a macro is much easier to maintain, in that a change in a single place in the source file will change all occurrences of the macro body uniformly. Attempting to change multiple copies of a single piece of source text is extremely error-prone, even with a powerful text editor.

The use of a macro can also make the program much easier to follow, as the messy details of the macro body are hidden, and do not clutter up the main program. Injudicious use of macros can serve to confuse, but in many cases the macro can clarify assembly language routines.

Parameter Passing

The simple macro scheme outlined in the previous section lacks the power necessary to perform even moderately complicated activities. The addition of parameters to macros greatly enhances their utility. A macro with parameters is defined by:

```
mymac   MACRO   dum1,dum2,...,dumn
        .
        .
        .
        ENDM
```

where dum1, dum2, ..., dumn are the n 'dummy parameters' to the macro. Each dummy parameter takes the form of a symbol which should be distinct from the others, but may clash with built-in names such as register names if it is desired that all occurrences of that name be substituted by something else. An invocation of the above macro becomes:

```
mymac   act1,act2,...,actn
```

where act1, act2, ..., actn are the 'actual parameters'. Each actual parameter takes the form of an arbitrary text string, separated from the others by commas or spaces. To pass a text string containing commas, spaces or characters that must not be folded to upper case, it can be enclosed by braces ({ }) causing the enclosed string to be passed "as is".

Note that although a quoted string passed as a parameter will not be folded, strings within the macro body may be, depending on the setting of *U at the time of the macro definition.

When the macro body is inserted into the source stream, occurrences of each dummy parameter within the body are replaced by the corresponding actual parameter. If there are more actual parameters than dummies, the extra ones are ignored. If there are more dummies than actual parameters, the extra dummies are replaced by null strings, i.e. by nothing. The modified macro body then becomes the 'replacement text'.

For example, it is possible to emulate the EMT instruction with the aid of

an appropriate macro. We may define a macro EMT by:

```
EMT     MACRO   CODE
        RST     30H             ;Execute a restart instruction
        DEFB    CODE            ;The EMT code
        ENDM
```

Suppose we call this by:

```
        EMT     KBDWF
```

Wherever the symbol CODE appears in the macro body, the symbol KBDWF is substituted. Thus, the replacement text becomes:

```
        RST     30H             ;Execute a restart instruction
        DEFB    KBDWF           ;The EMT code
```

It is important to realise that it is only when macro substitution is complete that any expression evaluation takes place. All macro substitution is entirely at the textual level. Thus, if KBDWF in the above example has the value 34, the replacement text does not contain the line:

```
        DEFB 34
```

but still refers to KBDWF.

Concatenation

Normally, substitution of macro parameters only occurs when a complete symbol matches the dummy parameter. Thus, the macro:

```
labgen  MACRO   num
labnum:
        ENDM
```

which might be intended to generate a label of the form lab3, does not do so; it generates 'labnum': instead. The concatenation operator '&' can be used to avoid this. If we redefine labgen as:

```
labgen  MACRO   num
lab&num:
        ENDM
```

then all is well, and a call to labgen of the form:

```
        labgen      3
```

now generates:

```
lab3:
```

in the desired manner.

Parameter Value Passing

The % operator can be used in front of an actual parameter to cause that parameter to be evaluated before substitution into the macro body. For example, suppose we invoke the above labgen macro by:

```
labno   EQU     3
        labgen  labno
```

This has the replacement text:

```
lablabno:
```

whereas:

```
labno   EQU     3
        labgen  %labno
```

is replaced by:

```
lab3:
```

It is therefore possible to write the following pair of macros to generate a new label every time one of them is called, once the label number (labno) has been initialized using := or DEFL

```
nextlab     MACRO
            labgen  %labno
labno       :=      labno + 1
            ENDM

labgen      MACRO   num
lab&num:
            ENDM
```

Further Examples

One of the major uses of macros is to confine details of the program, such as the representation of data, into a single place within the program listing (subroutines can also perform this function). Invoking macros can be expensive in terms of the space required.

A very useful trick to overcome this is to redefine a macro within the macro's invocation.

Consider the macro:

```
error   MACRO   errnum              ;Define error


        ld      a,'0'+errnum        ;Get the error number in A
        call    errout              ;Print the error number
        jr      errend              ;Jump round the output routine
```

```
errout: emt     outc            ;Print the character
        ld      a,CR            ;Print a carriage-return
        emt     outc
        ret
errend:


error   MACRO   err             ;Redefine the macro
        ld      a, '0'+err      ;Just get the error
        call    errout          ;and print it

        ENDM

        ENDM
```

This macro defines and uses a short subroutine (labelled errout) which simply prints out a number on the screen, followed by a carriage-return. It also redefines itself so that further invocations simply call errout, rather than bring in the entire routine. Thus, error can be called at any time, at a cost of five bytes per call. Coding the macro in line each time would cost eight bytes per call.

An example which illustrates most of the points in this chapter follows, in which a set of macros are used to generate code, and labels, to simulate the high level construct IF...THEN...[ELSE...]ENDIF.

A number of lines in this example begin with an asterisk. These form commands known as "directives", and are described in Chapter 7.

```
*H   'IF' macro definitions

;IFNOT....[IFTRUE....]ENDIF


;IFNOT  Generate new label number
;       Update nesting level
;       Code jump to block end if 'cc' not true
;       Store jump label

IFNOT   MACRO   cc
*L OFF

nxtlab  :=      nxtlab+1
iflevel :=      iflevel+1
        JUMP    %nxtlab,{cc,}
        DESTS   %iflevel

*L ON
        ENDM
```

```
;IFTRUE Generate new label number
;       Code jump to ENDIF
;       Code destination label forIFNOT
;       Store jump label

IFTRUE  MACRO
*L OFF

nxtlab  :=      nxtlab+1
        JUMP    %nxtlab
        DESTLAB %iflevel
        DESTS   %iflevel

*L ON
        ENDM


;ENDIF  Code destination for IFTRUE or IFNOT
;       Update nesting level

ENDIF   MACRO
*L OFF

        DESTLAB %iflevel
iflevel :=      iflevel-1

*L ON
        ENDM


JUMP    MACRO   labnum,optcc
        JR      optcc iflab&labnum
        ENDM


DESTS   MACRO   level
dest&level :=   nxtlab
        ENDM


DESTLAB MACRO   level
        LABGEN  %dest&level
        ENDM


LABGEN  MACRO   label
iflab&label:
        ENDM


nxtlab  DEFL    0               ; Initial label number
iflevel DEFL    0               ; Initial if nesting level
```

IFNOT is used at the start of the construct, causing the assembly of a jump to the end of the subsequent block (marked by IFTRUE, if present, or else by ENDIF) when the condition is true. If the condition is false the block will be executed up to the occurrence of the matching IFTRUE, when execution passes to ENDIF.

The use of these constructs is illustrated in the following program segment:

```
*LM OFF
        OR      A
        IFNOT Z
                LD      B,-1
                LD      C,(HL)
        IFTRUE
                LD      B,0
                IFNOT C
                        LD      C,0
                ENDIF
        ENDIF
*LM ON
```

which, expanding the macros by hand, would be written thus:

```
                OR      A
                JR      Z,IFLAB1
                LD      B,-1
                LD      C,(HL)
                JR      IFLAB2
IFLAB1:         LD      B,0
                JR      C,IFLAB3
                LD      C,0
IFLAB3:
IFLAB2:
```

and that, to the eyes of those more used to high level languages, is not as easy to read.

CHAPTER 7

DIRECTIVES

Directives are mostly concerned with the layout of an assembly listing. ZASM supports a number of directives identified by an asterisk (*) as the first non-blank item in a statment:

```
*INCLUDE filespec
*REQUEST ON/OFF (off)
*HEADING string
*FORMFEED ON/OFF (on)
*EJECT
*LIST ON/OFF (on)
*NUMBERING ON/OFF
*UPPER.CASE ON/OFF (on)
*WIDTH
*XREF
*SYMBOLS ON/OFF (on)
*PRINT
```

All of these may be abbreviated to their initial letter - thus *E and *EJECT have the same action. Certain directives in the above list may be followed by either ON or OFF. If the argument is omitted, the action is the same as ON. The argument in parenthesis is the default setting.

*INCLUDE filespec

The *INCLUDE directive allows ZASM input to be taken from an additional source file, where 'filespec' is a valid unambiguous CP/M file specification. Disc units A: to P: are allowed: if omitted, the disc defaults to the currently logged in disk. If the secondary filename is omitted, '.ZSM' is used.

Examples:

```
*I B:TABLES.ZSM
*I TABLES
*I A:SYSTEM.DEF
```

If logged in to disc B: the first two examples have the same action.

Note — If there is no carriage return at the end of an included file the next line of the file into which it has been included will become a continuation of its last line at assembly time and will therefore often appear to be ignored

*REQUEST

The *REQUEST directive causes the assembler to pause, waiting for the user to press a key, after it has reported each error on the console.

*HEADING string

The *HEADING directive causes 'string' to appear on the title line of each page. The first *H directive turns on paging and line numbering and sets the page number to 1. Subsequent *H's change the title string and cause a page throw. If used, *H is usually the first statement of a program, but is often followed by other *H's. The string is optional and is truncated to 28 characters.

*FORMFEED OFF

The *FORMFEED OFF directive informs ZASM that hardware formfeeds are not available on the listing device. If you have a 'Teletype' or similar device, *FORMFEED OFF should be the first statement of your program.

*FORMFEED [ON]

The *FORMFEED [ON] directive restores the use of hardware formfeeds (default setting).

*EJECT

The *EJECT directive causes a page throw. This directive is only active if there has been at least one *H. A formfeed character in the source file has a similar action.

*LIST OFF

The *LIST OFF directive turns off the listing. The statement containing *L OFF is listed, but the subsequent statements are not.

*LIST [ON]

The *LIST ON directive restores listing after *L OFF (the default setting is ON).

Note *L OFF/ON may be nested.

Global List Control

It is possible to disable the effect of the normal *L directive by using a Global listing directive.

*L+ turns on the Global listing facility
*L- turns off the Global listing facility

*L+ and *L- take the argument ON or OFF, which turns the listing on or off at that point.

It is not possible to nest global listing directives. All normal *L directives are ignored when the global listing facility is active.

Listing of Macros and Conditionals

It is possible to prevent the expanded code for a macro, or code conditioned out, from being output to the listing.

*LM ON indicates that macro expansions are to be listed
*LM OFF indicates that macro expansions are not to be listed

The default is *LM ON. The macro call is listed, and the line numbers are advanced by the correct amount.

*LC ON indicates that code conditioned out is to be listed
*LC OFF indicates that code conditioned out is not to be listed

The default is *LC ON. If *LC OFF is active, only the COND and ENDC statements are listed, but the line numbers are advanced as if the lines were listed.

*NUMBERING OFF

The *NUMBERING OFF directive turns off the line number field of the source listing. Line numbering is off by default but turned on by *H .

*NUMBERING [ON]

The *NUMBERING [ON] directive starts or restarts line numbering.

*UPPERCASE

The *UPPERCASE directive takes a single ON or OFF argument which enables or disables conversion of lower case (small) letters to upper case (capitals) before further assembly. Such conversion, however, does not apply to string or character constants or within comments, except those within the body of a macro. Case conversion is enabled by default, allowing programs to be entered in lower case.

*WIDTH

*WIDTH takes a numeric argument which controls the width of the program listing and symbol table. The numeric argument must lie in the range 21 to 132; if it does not, it is forced to the nearer limit. The width is counted from the left-most character position. Any character beyond the limit is not listed. The width is initially 132. The number of columns used in the symbol table (if enabled) is controlled by the WIDTH directive - the number of columns will be the largest number that will fit within the specified width.

CROSS-REFERENCE

*XREF takes a single ON or OFF argument which enables or disables cross-referencing. If cross-referencing is enabled, then ZASM makes a note of the line numbers of the definition of, and references to, each symbol. The symbol table output at the end of the listing will show this information, in addition to the value and mode of the symbol. The cross-reference listing does not include local labels. N.B. If *X ON is to be used it must precede the first label or symbol definition.

Occasionally it is useful to have all the references in a cross reference table in HEX, for example when there are more than 9999 lines in the .PRN file.

*XH indicates that the cross reference listing is to be in HEX
*XN indicates that the listing should show line numbers

The default is line numbers.

Warning It is not advisable to mix line numbers and HEX addresses in the same cross reference listing as it may be impossible to distinguish between them.

In a HEX cross reference listing equates are shown as having a reference to the next instruction in the file.

*SYMBOLS ON

The *SYMBOLS ON directive restores the User Symbol table listing (default setting).

*PRINT

*P is used to print messages or variables during assembly.

The message is always prefixed by an indication of which pass of the assembler it is from, for example:

```
Pass 1 : This is a message displayed on the console
```

*P message - prints 'message' on both passes of the assembler
*P1 message - prints 'message' on pass 1 only
*P2 message - prints 'message' on pass 2 only

A maximum of 127 characters can be printed with any one message.

*P= exp - prints out 'exp' followed by '=' and the value of the expression, which can be any normal expression acceptable by ZASM. Values are always output as a 4-digit HEX number, and are only output on pass 2, for example:

```
Pass 2 : $ = 5435
```

APPENDIX A

INSTRUCTION MNEMONICS

The mnemonic instruction set implemented by ZASM is the same as that defined by MOSTEK (see Mostek Z80 Programming Manual) except for:

The 8 bit Arithmetic Instructions (ADD, ADC, SUB, SBC, AND, XOR, OR and CP) can be presented in either the MOSTEK or ZILOG forms. For example, ADD A,7 and ADD 7 are acceptable forms.

Relative Jumps (DJNZ, JR) accept as an argument the branching address and not the displacement.

The IM0, IM1 and IM2 instructions must not include a space between the M and the digit.

Note that the decrement and jump non zero instruction must be written in the form "DJNZ label" with at least one space.

The exchange accumulator instruction EX AF,AF' need not include the terminal prime in ZASM. Note that ZASM ignores any trailing primes on a register name. However trailing primes can be useful in a listing to indicate the currently selected register set.

ZASM also interprets the two instructions EMT and CALR. These are pseudo instructions implemented by the firmware. CALR is a relative call and EMT is the EMulator Trap instruction. For details of these instructions refer to the Firmware Reference Manual.

Two complete lists of ZASM mnemonics are given on the following pages:

a) in alphabetical order
b) in numerical order

Note that in these lists the following values are assumed:

```
d    EQU  05H
n    EQU  20H
nn   EQU  584H
```

and "dis" is a label with an offset of 2EH.

A third list, in function order, may be found in the Machine Language Programmers Guide.

ADC	A,(HL)	8E
ADC	A,(IX+d)	DD8E05
ADC	A,(IY+d)	FD8E05
ADC	A,A	8F
ADC	A,B	88
ADC	A,C	89
ADC	A,D	8A
ADC	A,E	8B
ADC	A,H	8C
ADC	A,L	8D
ADC	A,n	CE20
ADC	HL,BC	ED4A
ADC	HL,DE	ED5A
ADC	HL,HL	ED6A
ADC	HL,SP	ED7A
ADD	A,(HL)	86
ADD	A,(IX+d)	DD8605
ADD	A,(IY+d)	FD8605
ADD	A,A	87
ADD	A,B	80
ADD	A,C	81
ADD	A,D	82
ADD	A,E	83
ADD	A,H	84
ADD	A,L	85
ADD	A,n	C620
ADD	HL,BC	09
ADD	HL,DE	19
ADD	HL,HL	29
ADD	HL,SP	39
ADD	IX,BC	DD09
ADD	IX,DE	DD19
ADD	IX,IX	DD29
ADD	IX,SP	DD39
ADD	IY,BC	FD09
ADD	IY,DE	FD19
ADD	IY,IY	FD29
ADD	IY,SP	FD39
AND	(HL)	A6
AND	(IX+d)	DDA605
AND	(IY+d)	FDA605
AND	A	A7
AND	B	A0
AND	C	A1
AND	D	A2
AND	E	A3
AND	H	A4
AND	L	A5
AND	n	E620
BIT	0,(HL)	CB46
BIT	0,(IX+d)	DDCB0546
BIT	0,(IY+d)	FDCB0546
BIT	0,A	CB47
BIT	0,B	CB40
BIT	0,C	CB41
BIT	0,D	CB42
BIT	0,E	CB43
BIT	0,H	CB44
BIT	0,L	CB45
BIT	1,(HL)	CB4E
BIT	1,(IX+d)	DDCB054E
BIT	1,(IY+d)	FDCB054E
BIT	1,A	CB4F
BIT	1,B	CB48
BIT	1,C	CB49
BIT	1,D	CB4A
BIT	1,E	CB4B
BIT	1,H	CB4C
BIT	1,L	CB4D
BIT	2,(HL)	CB56
BIT	2,(IX+d)	DDCB0556
BIT	2,(IY+d)	FDCB0556
BIT	2,A	CB57
BIT	2,B	CB50
BIT	2,C	CB51
BIT	2,D	CB52
BIT	2,E	CB53
BIT	2,H	CB54
BIT	2,L	CB55
BIT	3,(HL)	CB5E
BIT	3,(IX+d)	DDCB055E
BIT	3,(IY+d)	FDCB055E
BIT	3,A	CB5F
BIT	3,B	CB58
BIT	3,C	CB59
BIT	3,D	CB5A
BIT	3,E	CB5B
BIT	3,H	CB5C
BIT	3,L	CB5D
BIT	4,(HL)	CB66
BIT	4,(IX+d)	DDCB0566
BIT	4,(IY+d)	FDCB0566
BIT	4,A	CB67
BIT	4,B	CB60
BIT	4,C	CB61
BIT	4,D	CB62
BIT	4,E	CB63
BIT	4,H	CB64
BIT	4,L	CB65
BIT	5,(HL)	CB6E
BIT	5,(IX+d)	DDCB056E
BIT	5,(IY+d)	FDCB056E
BIT	5,A	CB6F
BIT	5,B	CB68
BIT	5,C	CB69
BIT	5,D	CB6A
BIT	5,E	CB6B
BIT	5,H	CB6C
BIT	5,L	CB6D
BIT	6,(HL)	CB76
BIT	6,(IX+d)	DDCB0576
BIT	6,(IY+d)	FDCB0576
BIT	6,A	CB77
BIT	6,B	CB70
BIT	6,C	CB71
BIT	6,D	CB72
BIT	6,E	CB73
BIT	6,H	CB74
BIT	6,L	CB75
BIT	7,(HL)	CB7E
BIT	7,(IX+d)	DDCB057E
BIT	7,(IY+d)	FDCB057E
BIT	7,A	CB7F
BIT	7,B	CB78
BIT	7,C	CB79
BIT	7,D	CB7A
BIT	7,E	CB7B
BIT	7,H	CB7C
BIT	7,L	CB7D
CALL	nn	CD8405
CALL	C,nn	DC8405
CALL	M,nn	FC8405
CALL	NC,nn	D48405
CALL	NZ,nn	C48405
CALL	P,nn	F48405
CALL	PE,nn	EC8405
CALL	PO,nn	E48405
CALL	Z,nn	CC8405
CCF		3F
CP	(HL)	BE
CP	(IX+d)	DDBE05
CP	(IY+d)	FDBE05
CP	A	BF
CP	B	B8
CP	C	B9
CP	D	BA
CP	E	BB
CP	H	BC
CP	L	BD
CP	n	FE20
CPD		EDA9
CPDR		EDB9
CPI		EDA1
CPIR		EDB1
CPL		2F
DAA		27
DEC	(HL)	35
DEC	(IX+d)	DD3505
DEC	(IY+d)	FD3505
DEC	A	3D
DEC	B	05
DEC	BC	0B
DEC	C	0D
DEC	D	15
DEC	DE	1B
DEC	E	1D
DEC	H	25
DEC	HL	2B
DEC	IX	DD2B
DEC	IY	FD2B
DEC	L	2D
DEC	SP	3B
DI		F3
DJNZ	dis	102E

EI		FB
EX	(SP),HL	E3
EX	(SP),IX	DDE3
EX	(SP),IY	FDE3
EX	AF,AF'	08
EX	DE,HL	EB
EXX		D9
HALT		76
IM0		ED46
IM1		ED56
IM2		ED5E
IN	A,(C)	ED78
IN	A,(n)	DB20
IN	B,(C)	ED40
IN	C,(C)	ED48
IN	D,(C)	ED50
IN	E,(C)	ED58
IN	H,(C)	ED60
IN	L,(C)	ED68
INC	(HL)	34
INC	(IX+d)	DD3405
INC	(IY+d)	FD3405
INC	A	3C
INC	B	04
INC	BC	03
INC	C	0C
INC	D	14
INC	DE	13
INC	E	1C
INC	H	24
INC	HL	23
INC	IX	DD23
INC	IY	FD23
INC	L	2C
INC	SP	33
IND		EDAA
INDR		EDBA
INI		EDA2
INIR		EDB2
JP	(HL)	E9
JP	(IX)	DDE9
JP	(IY)	FDE9
JP	nn	C38405
JP	C,nn	DA8405
JP	M,nn	FA8405
JP	NC,nn	D28405
JP	NZ,nn	C28405
JP	P,nn	F28405
JP	PE,nn	EA8405
JP	PO,nn	E28405
JP	Z,nn	CA8405
JR	dis	182E
JR	C,dis	382E
JR	NC,dis	302E
JR	NZ,dis	202E
JR	Z,dis	282E
LD	(BC),A	02
LD	(DE),A	12
LD	(HL),A	77
LD	(HL),B	70
LD	(HL),C	71
LD	(HL),D	72
LD	(HL),E	73
LD	(HL),H	74
LD	(HL),L	75
LD	(HL),n	3620
LD	(IX+d),A	DD7705
LD	(IX+d),B	DD7005
LD	(IX+d),C	DD7105
LD	(IX+d),D	DD7205
LD	(IX+d),E	DD7305
LD	(IX+d),H	DD7405
LD	(IX+d),L	DD7505
LD	(IX+d),n	DD360520
LD	(IY+d),A	FD7705
LD	(IY+d),B	FD7005
LD	(IY+d),C	FD7105
LD	(IY+d),D	FD7205
LD	(IY+d),E	FD7305
LD	(IY+d),H	FD7405
LD	(IY+d),L	FD7505
LD	(IY+d),n	FD360520
LD	(nn),A	328405
LD	(nn),BC	ED438405
LD	(nn),DE	ED538405
LD	(nn),HL	228405
LD	(nn),IX	DD228405
LD	(nn),IY	FD228405
LD	(nn),SP	ED738405
LD	A,(BC)	0A
LD	A,(DE)	1A
LD	A,(HL)	7E
LD	A,(IX+d)	DD7E05
LD	A,(IY+d)	FD7E05
LD	A,(nn)	3A8405
LD	A,A	7F
LD	A,B	78
LD	A,C	79
LD	A,D	7A
LD	A,E	7B
LD	A,H	7C
LD	A,I	ED57
LD	A,L	7D
LD	A,n	3E20
LD	A,R	ED5F
LD	B,(HL)	46
LD	B,(IX+d)	DD4605
LD	B,(IY+d)	FD4605
LD	B,A	47
LD	B,B	40
LD	B,C	41
LD	B,D	42
LD	B,E	43
LD	B,H	44
LD	B,L	45
LD	B,n	0620
LD	BC,(nn)	ED4B8405
LD	BC,nn	018405
LD	C,(HL)	4E
LD	C,(IX+d)	DD4E05
LD	C,(IY+d)	FD4E05
LD	C,A	4F
LD	C,B	48
LD	C,C	49
LD	C,D	4A
LD	C,E	4B
LD	C,H	4C
LD	C,L	4D
LD	C,n	0E20
LD	D,(HL)	56
LD	D,(IX+d)	DD5605
LD	D,(IY+d)	FD5605
LD	D,A	57
LD	D,B	50
LD	D,C	51
LD	D,D	52
LD	D,E	53
LD	D,H	54
LD	D,L	55
LD	D,n	1620
LD	DE,(nn)	ED5B8405
LD	DE,nn	118405
LD	E,(HL)	5E
LD	E,(IX+d)	DD5E05
LD	E,(IY+d)	FD5E05
LD	E,A	5F
LD	E,B	58
LD	E,C	59
LD	E,D	5A
LD	E,E	5B
LD	E,H	5C
LD	E,L	5D
LD	E,n	1E20
LD	H,(HL)	66
LD	H,(IX+d)	DD6605
LD	H,(IY+d)	FD6605
LD	H,A	67
LD	H,B	60
LD	H,C	61
LD	H,D	62
LD	H,E	63
LD	H,H	64
LD	H,L	65
LD	H,n	2620
LD	HL,(nn)	2A8405
LD	HL,nn	218405
LD	I,A	ED47
LD	IX,(nn)	DD2A8405
LD	IX,nn	DD218405
LD	IY,(nn)	FD2A8405
LD	IY,nn	FD218405
LD	L,(HL)	6E
LD	L,(IX+d)	DD6E05
LD	L,(IY+d)	FD6E05

LD	L,A	6F	RES	0,A	CB87	RES	6,(IX+d)	DDCB05B6
LD	L,B	68	RES	0,B	CB80	RES	6,(IY+d)	FDCB05B6
LD	L,C	69	RES	0,C	CB81	RES	6,A	CBB7
LD	L,D	6A	RES	0,D	CB82	RES	6,B	CBB0
LD	L,E	6B	RES	0,E	CB83	RES	6,C	CBB1
LD	L,H	6C	RES	0,H	CB84	RES	6,D	CBB2
LD	L,L	6D	RES	0,L	CB85	RES	6,E	CBB3
LD	L,n	2E20	RES	1,(HL)	CB8E	RES	6,H	CBB4
LD	R,A	ED4F	RES	1,(IX+d)	DDCB058E	RES	6,L	CBB5
LD	SP,(nn)	ED7B8405	RES	1,(IY+d)	FDCB058E	RES	7,(HL)	CBBE
LD	SP,HL	F9	RES	1,A	CB8F	RES	7,(IX+d)	DDCB05BE
LD	SP,IX	DDF9	RES	1,B	CB88	RES	7,(IY+d)	FDCB05BE
LD	SP,IY	FDF9	RES	1,C	CB89	RES	7,A	CBBF
LD	SP,nn	318405	RES	1,D	CB8A	RES	7,B	CBB8
LDD		EDA8	RES	1,E	CB8B	RES	7,C	CBB9
LDDR		EDB8	RES	1,H	CB8C	RES	7,D	CBBA
LDI		EDA0	RES	1,L	CB8D	RES	7,E	CBBB
LDIR		EDB0	RES	2,(HL)	CB96	RES	7,H	CBBC
NEG		ED44	RES	2,(IX+d)	DDCB0596	RES	7,L	CBBD
NOP		00	RES	2,(IY+d)	FDCB0596	RET		C9
OR	(HL)	B6	RES	2,A	CB97	RET	C	D8
OR	(IX+d)	DDB605	RES	2,B	CB90	RET	M	F8
OR	(IY+d)	FDB605	RES	2,C	CB91	RET	NC	D0
OR	A	B7	RES	2,D	CB92	RET	NZ	C0
OR	B	B0	RES	2,E	CB93	RET	P	F0
OR	C	B1	RES	2,H	CB94	RET	PE	E8
OR	D	B2	RES	2,L	CB95	RET	PO	E0
OR	E	B3	RES	3,(HL)	CB9E	RET	Z	C8
OR	H	B4	RES	3,(IX+d)	DDCB059E	RETI		ED4D
OR	L	B5	RES	3,(IY+d)	FDCB059E	RETN		ED45
OR	n	F620	RES	3,A	CB9F	RL	(HL)	CB16
OTDR		EDBB	RES	3,B	CB98	RL	(IX+d)	DDCB0516
OTIR		EDB3	RES	3,C	CB99	RL	(IY+d)	FDCB0516
OUT	(C),A	ED79	RES	3,D	CB9A	RL	A	CB17
OUT	(C),B	ED41	RES	3,E	CB9B	RL	B	CB10
OUT	(C),C	ED49	RES	3,H	CB9C	RL	C	CB11
OUT	(C),D	ED51	RES	3,L	CB9D	RL	D	CB12
OUT	(C),E	ED59	RES	4,(HL)	CBA6	RL	E	CB13
OUT	(C),H	ED61	RES	4,(IX+d)	DDCB05A6	RL	H	CB14
OUT	(C),L	ED69	RES	4,(IY+d)	FDCB05A6	RL	L	CB15
OUT	(n),A	D320	RES	4,A	CBA7	RLA		17
OUTD		EDAB	RES	4,B	CBA0	RLC	(HL)	CB06
OUTI		EDA3	RES	4,C	CBA1	RLC	(IX+d)	DDCB0506
POP	AF	F1	RES	4,D	CBA2	RLC	(IY+d)	FDCB0506
POP	BC	C1	RES	4,E	CBA3	RLC	A	CB07
POP	DE	D1	RES	4,H	CBA4	RLC	B	CB00
POP	HL	E1	RES	4,L	CBA5	RLC	C	CB01
POP	IX	DDE1	RES	5,(HL)	CBAE	RLC	D	CB02
POP	IY	FDE1	RES	5,(IX+d)	DDCB05AE	RLC	E	CB03
PUSH	AF	F5	RES	5,(IY+d)	FDCB05AE	RLC	H	CB04
PUSH	BC	C5	RES	5,A	CBAF	RLC	L	CB05
PUSH	DE	D5	RES	5,B	CBA8	RLCA		07
PUSH	HL	E5	RES	5,C	CBA9	RLD		ED6F
PUSH	IX	DDE5	RES	5,D	CBAA	RR	(HL)	CB1E
PUSH	IY	FDE5	RES	5,E	CBAB	RR	(IX+d)	DDCB051E
RES	0,(HL)	CB86	RES	5,H	CBAC	RR	(IY+d)	FDCB051E
RES	0,(IX+d)	DDCB0586	RES	5,L	CBAD	RR	A	CB1F
RES	0,(IY+d)	FDCB0586	RES	6,(HL)	CBB6	RR	B	CB18

RR	C	CB19
RR	D	CB1A
RR	E	CB1B
RR	H	CB1C
RR	L	CB1D
RRA		1F
RRC	(HL)	CB0E
RRC	(IX+d)	DDCB050E
RRC	(IY+d)	FDCB050E
RRC	A	CB0F
RRC	B	CB08
RRC	C	CB09
RRC	D	CB0A
RRC	E	CB0B
RRC	H	CB0C
RRC	L	CB0D
RRCA		0F
RRD		ED67
RST	0H	C7
RST	8H	CF
RST	10H	D7
RST	18H	DF
RST	20H	E7
RST	28H	EF
RST	30H	F7
RST	38H	FF
SBC	A,(HL)	9E
SBC	A,(IX+d)	DD9E05
SBC	A,(IY+d)	FD9E05
SBC	A,A	9F
SBC	A,B	98
SBC	A,C	99
SBC	A,D	9A
SBC	A,E	9B
SBC	A,H	9C
SBC	A,L	9D
SBC	A,n	DE20
SBC	HL,BC	ED42
SBC	HL,DE	ED52
SBC	HL,HL	ED62
SBC	HL,SP	ED72
SCF		37
SET	0,(HL)	CBC6
SET	0,(IX+d)	DDCB05C6
SET	0,(IY+d)	FDCB05C6
SET	0,A	CBC7
SET	0,B	CBC0
SET	0,C	CBC1
SET	0,D	CBC2
SET	0,E	CBC3
SET	0,H	CBC4
SET	0,L	CBC5
SET	1,(HL)	CBCE
SET	1,(IX+d)	DDCB05CE
SET	1,(IY+d)	FDCB05CE
SET	1,A	CBCF
SET	1,B	CBC8
SET	1,C	CBC9
SET	1,D	CBCA
SET	1,E	CBCB
SET	1,H	CBCC
SET	1,L	CBCD
SET	2,(HL)	CBD6
SET	2,(IX+d)	DDCB05D6
SET	2,(IY+d)	FDCB05D6
SET	2,A	CBD7
SET	2,B	CBD0
SET	2,C	CBD1
SET	2,D	CBD2
SET	2,E	CBD3
SET	2,H	CBD4
SET	2,L	CBD5
SET	3,(HL)	CBDE
SET	3,(IX+d)	DDCB05DE
SET	3,(IY+d)	FDCB05DE
SET	3,A	CBDF
SET	3,B	CBD8
SET	3,C	CBD9
SET	3,D	CBDA
SET	3,E	CBDB
SET	3,H	CBDC
SET	3,L	CBDD
SET	4,(HL)	CBE6
SET	4,(IX+d)	DDCB05E6
SET	4,(IY+d)	FDCB05E6
SET	4,A	CBE7
SET	4,B	CBE0
SET	4,C	CBE1
SET	4,D	CBE2
SET	4,E	CBE3
SET	4,H	CBE4
SET	4,L	CBE5
SET	5,(HL)	CBEE
SET	5,(IX+d)	DDCB05EE
SET	5,(IY+d)	FDCB05EE
SET	5,A	CBEF
SET	5,B	CBE8
SET	5,C	CBE9
SET	5,D	CBEA
SET	5,E	CBEB
SET	5,H	CBEC
SET	5,L	CBED
SET	6,(HL)	CBF6
SET	6,(IX+d)	DDCB05F6
SET	6,(IY+d)	FDCB05F6
SET	6,A	CBF7
SET	6,B	CBF0
SET	6,C	CBF1
SET	6,D	CBF2
SET	6,E	CBF3
SET	6,H	CBF4
SET	6,L	CBF5
SET	7,(HL)	CBFE
SET	7,(IX+d)	DDCB05FE
SET	7,(IY+d)	FDCB05FE
SET	7,A	CBFF
SET	7,B	CBF8
SET	7,C	CBF9
SET	7,D	CBFA
SET	7,E	CBFB
SET	7,H	CBFC
SET	7,L	CBFD
SLA	(HL)	CB26
SLA	(IX+d)	DDCB0526
SLA	(IY+d)	FDCB0526
SLA	A	CB27
SLA	B	CB20
SLA	C	CB21
SLA	D	CB22
SLA	E	CB23
SLA	H	CB24
SLA	L	CB25
SRA	(HL)	CB2E
SRA	(IX+d)	DDCB052E
SRA	(IY+d)	FDCB052E
SRA	A	CB2F
SRA	B	CB28
SRA	C	CB29
SRA	D	CB2A
SRA	E	CB2B
SRA	H	CB2C
SRA	L	CB2D
SRL	(HL)	CB3E
SRL	(IX+d)	DDCB053E
SRL	(IY+d)	FDCB053E
SRL	A	CB3F
SRL	B	CB38
SRL	C	CB39
SRL	D	CB3A
SRL	E	CB3B
SRL	H	CB3C
SRL	L	CB3D
SUB	(HL)	96
SUB	(IX+d)	DD9605
SUB	(IY+d)	FD9605
SUB	A	97
SUB	B	90
SUB	C	91
SUB	D	92
SUB	E	93
SUB	H	94
SUB	L	95
SUB	n	D620
XOR	(HL)	AE
XOR	(IX+d)	DDAE05
XOR	(IY+d)	FDAE05
XOR	A	AF
XOR	B	A8
XOR	C	A9
XOR	D	AA
XOR	E	AB
XOR	H	AC
XOR	L	AD
XOR	n	EE20

00	NOP	
018405	LD	BC,nn
02	LD	(BC),A
03	INC	BC
04	INC	B
05	DEC	B
0620	LD	B,n
07	RLCA	
08	EX	AF,AF'
09	ADD	HL,BC
0A	LD	A,(BC)
0B	DEC	BC
0C	INC	C
0D	DEC	C
0E20	LD	C,n
0F	RRCA	
102E	DJNZ	dis
118405	LD	DE,nn
12	LD	(DE),A
13	INC	DE
14	INC	D
15	DEC	D
1620	LD	D,n
17	RLA	
182E	JR	dis
19	ADD	HL,DE
1A	LD	A,(DE)
1B	DEC	DE
1C	INC	E
1D	DEC	E
1E20	LD	E,n
1F	RRA	
202E	JR	NZ,dis
218405	LD	HL,nn
228405	LD	(nn),HL
23	INC	HL
24	INC	H
25	DEC	H
2620	LD	H,n
27	DAA	
282E	JR	Z,dis
29	ADD	HL,HL
2A8405	LD	HL,(nn)
2B	DEC	HL
2C	INC	L
2D	DEC	L
2E20	LD	L,n
2F	CPL	
302E	JR	NC,dis
318405	LD	SP,nn
328405	LD	(nn),A
33	INC	SP
34	INC	(HL)
35	DEC	(HL)
3620	LD	(HL),n
37	SCF	
382E	JR	C,dis
39	ADD	HL,SP
3A8405	LD	A,(nn)
3B	DEC	SP
3C	INC	A
3D	DEC	A
3E20	LD	A,n
3F	CCF	
40	LD	B,B
41	LD	B,C
42	LD	B,D
43	LD	B,E
44	LD	B,H
45	LD	B,L
46	LD	B,(HL)
47	LD	B,A
48	LD	C,B
49	LD	C,C
4A	LD	C,D
4B	LD	C,E
4C	LD	C,H
4D	LD	C,L
4E	LD	C,(HL)
4F	LD	C,A
50	LD	D,B
51	LD	D,C
52	LD	D,D
53	LD	D,E
54	LD	D,H
55	LD	D,L
56	LD	D,(HL)
57	LD	D,A
58	LD	E,B
59	LD	E,C
5A	LD	E,D
5B	LD	E,E
5C	LD	E,H
5D	LD	E,L
5E	LD	E,(HL)
5F	LD	E,A
60	LD	H,B
61	LD	H,C
62	LD	H,D
63	LD	H,E
64	LD	H,H
65	LD	H,L
66	LD	H,(HL)
67	LD	H,A
68	LD	L,B
69	LD	L,C
6A	LD	L,D
6B	LD	L,E
6C	LD	L,H
6D	LD	L,L
6E	LD	L,(HL)
6F	LD	L,A
70	LD	(HL),B
71	LD	(HL),C
72	LD	(HL),D
73	LD	(HL),E
74	LD	(HL),H
75	LD	(HL),L
76	HALT	
77	LD	(HL),A
78	LD	A,B
79	LD	A,C
7A	LD	A,D
7B	LD	A,E
7C	LD	A,H
7D	LD	A,L
7E	LD	A,(HL)
7F	LD	A,A
80	ADD	A,B
81	ADD	A,C
82	ADD	A,D
83	ADD	A,E
84	ADD	A,H
85	ADD	A,L
86	ADD	A,(HL)
87	ADD	A,A
88	ADC	A,B
89	ADC	A,C
8A	ADC	A,D
8B	ADC	A,E
8C	ADC	A,H
8D	ADC	A,L
8E	ADC	A,(HL)
8F	ADC	A,A
90	SUB	B
91	SUB	C
92	SUB	D
93	SUB	E
94	SUB	H
95	SUB	L
96	SUB	(HL)
97	SUB	A
98	SBC	A,B
99	SBC	A,C
9A	SBC	A,D
9B	SBC	A,E
9C	SBC	A,H
9D	SBC	A,L
9E	SBC	A,(HL)
9F	SBC	A,A
A0	AND	B
A1	AND	C
A2	AND	D
A3	AND	E
A4	AND	H
A5	AND	L
A6	AND	(HL)
A7	AND	A
A8	XOR	B
A9	XOR	C
AA	XOR	D
AB	XOR	E
AC	XOR	H
AD	XOR	L

AE	XOR	(HL)	CB1D	RR	L	CB5F	BIT	3,A
AF	XOR	A	CB1E	RR	(HL)	CB60	BIT	4,B
B0	OR	B	CB1F	RR	A	CB61	BIT	4,C
B1	OR	C	CB20	SLA	B	CB62	BIT	4,D
B2	OR	D	CB21	SLA	C	CB63	BIT	4,E
B3	OR	E	CB22	SLA	D	CB64	BIT	4,H
B4	OR	H	CB23	SLA	E	CB65	BIT	4,L
B5	OR	L	CB24	SLA	H	CB66	BIT	4,(HL)
B6	OR	(HL)	CB25	SLA	L	CB67	BIT	4,A
B7	OR	A	CB26	SLA	(HL)	CB68	BIT	5,B
B8	CP	B	CB27	SLA	A	CB69	BIT	5,C
B9	CP	C	CB28	SRA	B	CB6A	BIT	5,D
BA	CP	D	CB29	SRA	C	CB6B	BIT	5,E
BB	CP	E	CB2A	SRA	D	CB6C	BIT	5,H
BC	CP	H	CB2B	SRA	E	CB6D	BIT	5,L
BD	CP	L	CB2C	SRA	H	CB6E	BIT	5,(HL)
BE	CP	(HL)	CB2D	SRA	L	CB6F	BIT	5,A
BF	CP	A	CB2E	SRA	(HL)	CB70	BIT	6,B
C0	RET	NZ	CB2F	SRA	A	CB71	BIT	6,C
C1	POP	BC	CB38	SRL	B	CB72	BIT	6,D
C28405	JP	NZ,nn	CB39	SRL	C	CB73	BIT	6,E
C38405	JP	nn	CB3A	SRL	D	CB74	BIT	6,H
C48405	CALL	NZ,nn	CB3B	SRL	E	CB75	BIT	6,L
C5	PUSH	BC	CB3C	SRL	H	CB76	BIT	6,(HL)
C620	ADD	A,n	CB3D	SRL	L	CB77	BIT	6,A
C7	RST	00H	CB3E	SRL	(HL)	CB78	BIT	7,B
C8	RET	Z	CB3F	SRL	A	CB79	BIT	7,C
C9	RET		CB40	BIT	0,B	CB7A	BIT	7,D
CA8405	JP	Z,nn	CB41	BIT	0,C	CB7B	BIT	7,E
CB00	RLC	B	CB42	BIT	0,D	CB7C	BIT	7,H
CB01	RLC	C	CB43	BIT	0,E	CB7D	BIT	7,L
CB02	RLC	D	CB44	BIT	0,H	CB7E	BIT	7,(HL)
CB03	RLC	E	CB45	BIT	0,L	CB7F	BIT	7,A
CB04	RLC	H	CB46	BIT	0,(HL)	CB80	RES	0,B
CB05	RLC	L	CB47	BIT	0,A	CB81	RES	0,C
CB06	RLC	(HL)	CB48	BIT	1,B	CB82	RES	0,D
CB07	RLC	A	CB49	BIT	1,C	CB83	RES	0,E
CB08	RRC	B	CB4A	BIT	1,D	CB84	RES	0,H
CB09	RRC	C	CB4B	BIT	1,E	CB85	RES	0,L
CB0A	RRC	D	CB4C	BIT	1,H	CB86	RES	0,(HL)
CB0B	RRC	E	CB4D	BIT	1,L	CB87	RES	0,A
CB0C	RRC	H	CB4E	BIT	1,(HL)	CB88	RES	1,B
CB0D	RRC	L	CB4F	BIT	1,A	CB89	RES	1,C
CB0E	RRC	(HL)	CB50	BIT	2,B	CB8A	RES	1,D
CB0F	RRC	A	CB51	BIT	2,C	CB8B	RES	1,E
CB10	RL	B	CB52	BIT	2,D	CB8C	RES	1,H
CB11	RL	C	CB53	BIT	2,E	CB8D	RES	1,L
CB12	RL	D	CB54	BIT	2,H	CB8E	RES	1,(HL)
CB13	RL	E	CB55	BIT	2,L	CB8F	RES	1,A
CB14	RL	H	CB56	BIT	2,(HL)	CB90	RES	2,B
CB15	RL	L	CB57	BIT	2,A	CB91	RES	2,C
CB16	RL	(HL)	CB58	BIT	3,B	CB92	RES	2,D
CB17	RL	A	CB59	BIT	3,C	CB93	RES	2,E
CB18	RR	B	CB5A	BIT	3,D	CB94	RES	2,H
CB19	RR	C	CB5B	BIT	3,E	CB95	RES	2,L
CB1A	RR	D	CB5C	BIT	3,H	CB96	RES	2,(HL)
CB1B	RR	E	CB5D	BIT	3,L	CB97	RES	2,A
CB1C	RR	H	CB5E	BIT	3,(HL)	CB98	RES	3,B

CB99	RES	3,C
CB9A	RES	3,D
CB9B	RES	3,E
CB9C	RES	3,H
CB9D	RES	3,L
CB9E	RES	3,(HL)
CB9F	RES	3,A
CBA0	RES	4,B
CBA1	RES	4,C
CBA2	RES	4,D
CBA3	RES	4,E
CBA4	RES	4,H
CBA5	RES	4,L
CBA6	RES	4,(HL)
CBA7	RES	4,A
CBA8	RES	5,B
CBA9	RES	5,C
CBAA	RES	5,D
CBAB	RES	5,E
CBAC	RES	5,H
CBAD	RES	5,L
CBAE	RES	5,(HL)
CBAF	RES	5,A
CBB0	RES	6,B
CBB1	RES	6,C
CBB2	RES	6,D
CBB3	RES	6,E
CBB4	RES	6,H
CBB5	RES	6,L
CBB6	RES	6,(HL)
CBB7	RES	6,A
CBB8	RES	7,B
CBB9	RES	7,C
CBBA	RES	7,D
CBBB	RES	7,E
CBBC	RES	7,H
CBBD	RES	7,L
CBBE	RES	7,(HL)
CBBF	RES	7,A
CBC0	SET	0,B
CBC1	SET	0,C
CBC2	SET	0,D
CBC3	SET	0,E
CBC4	SET	0,H
CBC5	SET	0,L
CBC6	SET	0,(HL)
CBC7	SET	0,A
CBC8	SET	1,B
CBC9	SET	1,C
CBCA	SET	1,D
CBCB	SET	1,E
CBCC	SET	1,H
CBCD	SET	1,L
CBCE	SET	1,(HL)
CBCF	SET	1,A
CBD0	SET	2,B
CBD1	SET	2,C
CBD2	SET	2,D
CBD3	SET	2,E
CBD4	SET	2,H
CBD5	SET	2,L
CBD6	SET	2,(HL)
CBD7	SET	2,A
CBD8	SET	3,B
CBD9	SET	3,C
CBDA	SET	3,D
CBDB	SET	3,E
CBDC	SET	3,H
CBDD	SET	3,L
CBDE	SET	3,(HL)
CBDF	SET	3,A
CBE0	SET	4,B
CBE1	SET	4,C
CBE2	SET	4,D
CBE3	SET	4,E
CBE4	SET	4,H
CBE5	SET	4,L
CBE6	SET	4,(HL)
CBE7	SET	4,A
CBE8	SET	5,B
CBE9	SET	5,C
CBEA	SET	5,D
CBEB	SET	5,E
CBEC	SET	5,H
CBED	SET	5,L
CBEE	SET	5,(HL)
CBEF	SET	5,A
CBF0	SET	6,B
CBF1	SET	6,C
CBF2	SET	6,D
CBF3	SET	6,E
CBF4	SET	6,H
CBF5	SET	6,L
CBF6	SET	6,(HL)
CBF7	SET	6,A
CBF8	SET	7,B
CBF9	SET	7,C
CBFA	SET	7,D
CBFB	SET	7,E
CBFC	SET	7,H
CBFD	SET	7,L
CBFE	SET	7,(HL)
CBFF	SET	7,A
CC8405	CALL	Z,nn
CD8405	CALL	nn
CE20	ADC	A,n
CF	RST	08H
D0	RET	NC
D1	POP	DE
D28405	JP	NC,nn
D320	OUT	(n),A
D48405	CALL	NC,nn
D5	PUSH	DE
D620	SUB	n
D7	RST	10H
D8	RET	C
D9	EXX	
DA8405	JP	C,nn
DB20	IN	A,(n)
DC8405	CALL	C,nn
DD09	ADD	IX,BC
DD19	ADD	IX,DE
DD218405	LD	IX,nn
DD228405	LD	(nn),IX
DD23	INC	IX
DD29	ADD	IX,IX
DD2A8405	LD	IX,(nn)
DD2B	DEC	IX
DD3405	INC	(IX+d)
DD3505	DEC	(IX+d)
DD360520	LD	(IX+d),n
DD39	ADD	IX,SP
DD4605	LD	B,(IX+d)
DD4E05	LD	C,(IX+d)
DD5605	LD	D,(IX+d)
DD5E05	LD	E,(IX+d)
DD6605	LD	H,(IX+d)
DD6E05	LD	L,(IX+d)
DD7005	LD	(IX+d),B
DD7105	LD	(IX+d),C
DD7205	LD	(IX+d),D
DD7305	LD	(IX+d),E
DD7405	LD	(IX+d),H
DD7505	LD	(IX+d),L
DD7705	LD	(IX+d),A
DD7E05	LD	A,(IX+d)
DD8605	ADD	A,(IX+d)
DD8E05	ADC	A,(IX+d)
DD9605	SUB	(IX+d)
DD9E05	SBC	A,(IX+d)
DDA605	AND	(IX+d)
DDAE05	XOR	(IX+d)
DDB605	OR	(IX+d)
DDBE05	CP	(IX+d)
DDCB0506	RLC	(IX+d)
DDCB050E	RRC	(IX+d)
DDCB0516	RL	(IX+d)
DDCB051E	RR	(IX+d)
DDCB0526	SLA	(IX+d)
DDCB052E	SRA	(IX+d)
DDCB053E	SRL	(IX+d)
DDCB0546	BIT	0,(IX+d)
DDCB054E	BIT	1,(IX+d)
DDCB0556	BIT	2,(IX+d)
DDCB055E	BIT	3,(IX+d)
DDCB0566	BIT	4,(IX+d)
DDCB056E	BIT	5,(IX+d)
DDCB0576	BIT	6,(IX+d)
DDCB057E	BIT	7,(IX+d)
DDCB0586	RES	0,(IX+d)
DDCB058E	RES	1,(IX+d)
DDCB0596	RES	2,(IX+d)
DDCB059E	RES	3,(IX+d)
DDCB05A6	RES	4,(IX+d)

```
DDCB05AE  RES  5,(IX+d)
DDCB05B6  RES  6,(IX+d)
DDCB05BE  RES  7,(IX+d)
DDCB05C6  SET  0,(IX+d)
DDCB05CE  SET  1,(IX+d)
DDCB05D6  SET  2,(IX+d)
DDCB05DE  SET  3,(IX+d)
DDCB05E6  SET  4,(IX+d)
DDCB05EE  SET  5,(IX+d)
DDCB05F6  SET  6,(IX+d)
DDCB05FE  SET  7,(IX+d)
DDE1      POP  IX
DDE3      EX   (SP),IX
DDE5      PUSH IX
DDE9      JP   (IX)
DDF9      LD   SP,IX
DE20      SBC  A,n
DF        RST  18H
E0        RET  PO
E1        POP  HL
E28405    JP   PO,nn
E3        EX   (SP),HL
E48405    CALL PO,nn
E5        PUSH HL
E620      AND  n
E7        RST  20H
E8        RET  PE
E9        JP   (HL)
EA8405    JP   PE,nn
EB        EX   DE,HL
EC8405    CALL PE,nn
ED40      IN   B,(C)
ED41      OUT  (C),B
ED42      SBC  HL,BC
ED438405  LD   (nn),BC
ED44      NEG
ED45      RETN
ED46      IM0
ED47      LD   I,A
ED48      IN   C,(C)
ED49      OUT  (C),C
ED4A      ADC  HL,BC
ED4B8405  LD   BC,(nn)
ED4D      RETI
ED4F      LD   R,A
ED50      IN   D,(C)
ED51      OUT  (C),D
ED52      SBC  HL,DE
ED538405  LD   (nn),DE
ED56      IM1
ED57      LD   A,I
ED58      IN   E,(C)
ED59      OUT  (C),E
ED5A      ADC  HL,DE
ED5B8405  LD   DE,(nn)
ED5E      IM2
ED5F      LD   A,R
ED60      IN   H,(C)
ED61      OUT  (C),H
ED62      SBC  HL,HL
ED67      RRD
ED68      IN   L,(C)
ED69      OUT  (C),L
ED6A      ADC  HL,HL
ED6F      RLD
ED72      SBC  HL,SP
ED738405  LD   (nn),SP
ED78      IN   A,(C)
ED79      OUT  (C),A
ED7A      ADC  HL,SP
ED7B8405  LD   SP,(nn)
EDA0      LDI
EDA1      CPI
EDA2      INI
EDA3      OUTI
EDA8      LDD
EDA9      CPD
EDAA      IND
EDAB      OUTD
EDB0      LDIR
EDB1      CPIR
EDB2      INIR
EDB3      OTIR
EDB8      LDDR
EDB9      CPDR
EDBA      INDR
EDBB      OTDR
EE20      XOR  n
EF        RST  28H
F0        RET  P
F1        POP  AF
F28405    JP   P,nn
F3        DI
F48405    CALL P,nn
F5        PUSH AF
F620      OR   n
F7        RST  30H
F8        RET  M
F9        LD   SP,HL
FA8405    JP   M,nn
FB        EI
FC8405    CALL M,nn
FD09      ADD  IY,BC
FD19      ADD  IY,DE
FD218405  LD   IY,nn
FD228405  LD   (nn),IY
FD23      INC  IY
FD29      ADD  IY,IY
FD2A8405  LD   IY,(nn)
FD2B      DEC  IY
FD3405    INC  (IY+d)
FD3505    DEC  (IY+d)
FD360520  LD   (IY+d),n
FD39      ADD  IY,SP
FD4605    LD   B,(IY+d)
FD4E05    LD   C,(IY+d)
FD5605    LD   D,(IY+d)
FD5E05    LD   E,(IY+d)
FD6605    LD   H,(IY+d)
FD6E05    LD   L,(IY+d)
FD7005    LD   (IY+d),B
FD7105    LD   (IY+d),C
FD7205    LD   (IY+d),D
FD7305    LD   (IY+d),E
FD7405    LD   (IY+d),H
FD7505    LD   (IY+d),L
FD7705    LD   (IY+d),A
FD7E05    LD   A,(IY+d)
FD8605    ADD  A,(IY+d)
FD8E05    ADC  A,(IY+d)
FD9605    SUB  (IY+d)
FD9E05    SBC  A,(IY+d)
FDA605    AND  (IY+d)
FDAE05    XOR  (IY+d)
FDB605    OR   (IY+d)
FDBE05    CP   (IY+d)
FDCB0506  RLC  (IY+d)
FDCB050E  RRC  (IY+d)
FDCB0516  RL   (IY+d)
FDCB051E  RR   (IY+d)
FDCB0526  SLA  (IY+d)
FDCB052E  SRA  (IY+d)
FDCB053E  SRL  (IY+d)
FDCB0546  BIT  0,(IY+d)
FDCB054E  BIT  1,(IY+d)
FDCB0556  BIT  2,(IY+d)
FDCB055E  BIT  3,(IY+d)
FDCB0566  BIT  4,(IY+d)
FDCB056E  BIT  5,(IY+d)
FDCB0576  BIT  6,(IY+d)
FDCB057E  BIT  7,(IY+d)
FDCB0586  RES  0,(IY+d)
FDCB058E  RES  1,(IY+d)
FDCB0596  RES  2,(IY+d)
FDCB059E  RES  3,(IY+d)
FDCB05A6  RES  4,(IY+d)
FDCB05AE  RES  5,(IY+d)
FDCB05B6  RES  6,(IY+d)
FDCB05BE  RES  7,(IY+d)
FDCB05C6  SET  0,(IY+d)
FDCB05CE  SET  1,(IY+d)
FDCB05D6  SET  2,(IY+d)
FDCB05DE  SET  3,(IY+d)
FDCB05E6  SET  4,(IY+d)
FDCB05EE  SET  5,(IY+d)
FDCB05F6  SET  6,(IY+d)
FDCB05FE  SET  7,(IY+d)
FDE1      POP  IY
FDE3      EX   (SP),IY
FDE5      PUSH IY
FDE9      JP   (IY)
FDF9      LD   SP,IY
FE20      CP   n
FF        RST  38H
```

Alternative Mnemonics

ZASM allows several alternative forms of the Zilog mnemonics as a programming convenience:

8-bit arithmetic instructions need not specify the first argument (which is always the A register):

```
bytes assembled          Instruction
     80                  ADD         A,B
     80                  ADD         B
     89                  ADC         A,C
     89                  ADC         C
     A2                  AND         A,D
     A2                  AND         D
     FE20                CP          A,2OH
     FE20                CP          2OH
     B3                  OR          A,E
     B3                  OR          E
     9C                  SBC         A,H
     9C                  SBC         H
     95                  SUB         A,L
     95                  SUB         L
     AE                  XOR         A,(HL)
     AE                  XOR         (HL)
```

With EX AF,AF' the single quote may be omitted from either, both or neither operand:

```
     08                  EX          AF,AF'
     08                  EX          AF,AF
     08                  EX          AF',AF
     08                  EX          AF',AF'
```

For RELATIVE jump instructions, the argument is taken as the absolute target to jump to - just as for ABSOLUTE jumps:

```
                    LOOP:  DEFS  2OH
     C32007                JP    LOOP    ;ABSOLUTE
     18DB                  JR    LOOP    ;RELATIVE
```

ZASM syntax differs in this respect from some of the examples found in Zilog publications, where the form JR LOOP-$ is used. In the Zilog case the argument is the offset to, rather then the address of, the target.

For 'DJNZ' both Zilog and Mostek forms are accepted:

```
     10D9      DJNZ      LOOP      ;ZILOG
     10D7      DJNZ,     LOOP      ;MOSTEK
but:
     00        DJ        NZ,LOOP
```

is not accepted.

With DEFB, DEFW and FREE-FORMAT a trailing comma is ignored:

```
0102            DEFB      1,2,
01000200        DEFW      1,2,
0102            1,2,                ;TREATED AS DEFB
```

An embedded comma generates zero:

```
010002          DEFB      1,,2
01000000        DEFW      1,,2
010002          1,,2            ; ZERO BYTE
01000002        1,,,2           ; ZERO WORD
```

Two additional instructions are available with ZASM:

```
EMT expr
```

generates the hex code F7 (RST 30H) followed by the 8 bit value of expr.;

```
CALR label
```

generates a relative procedure call to label, with the hex code EF (RST 28H) followed by the relative distance to label, computed as for the JR instruction.

Examples:

```
F705            EMT       5     ;OUTPUT TO LST:
EF00            CALR      $+2   ;PUSH PC
```

APPENDIX B

ERROR CODES

ZASM flags errors with >X, where X is an error code. This is to allow a search of a listing file for all errors, by looking for all occurences of the characters 'RETURN, LINEFEED,>' with a text editor.

The meanings of the codes are:

B	bracket (parenthesis) error
C	conditional error
E	expression error
I	illegal character in context
M	multiple definition of symbol
N	number error
O	origin redefined backwards
P	phase error
R	register name misused
S	bad syntax
U	undefined symbol
V	value error

Examples:

Bracket Error

```
>B0766 3E07          LD A,1 + (2 *3
>B                   LD HL),A
>B                   LD (HL,A
```

Conditional Error (no COND)

```
>C                   ELSE
>C                   ENDC
```

Illegal Character

```
>I0768 78            LD A,B,
>I0769 3E00          LD A,Z**4
>I076B EDB0          LDIR DE ;BAD ARG
```

Multiple Definition

```
  0005 =             BDOS   EQU 5
>M0006 =             BDOS   EQU 6
```

Number Error

```
>N076D 00          DEFB 13B  ;NOT BINARY
>N076E 00          DEFB 8Q   ;NOT OCTAL
>N076F 00          DEFB 0A   ;NOT DECIMAL
```

Origin Redefined

```
       COM
       ORG  300H
>O     ORG  100H
```

Phase Error

```
  0770 78          TEST: LD  A,B
>P0771 41          TEST: LD  B,C
```

Note that in this case error message will have been preceded by a message of the form: Multiple def at line xxxx.

Register Error

```
>R                 BC: LD  BC,5
```

Bad Syntax

Most errors fall into this class.

```
>S                          * LIST LAB  ;BAD ARG
>U0772 00          FINAL    LD  A,O  ;EQU EXPECTED
>S                          LD (HL+ 3),0
>S                          DEFM AB  ; NO STRING
>S0773 4142093B             DEFM 'AB  ;MISSING QUOTE
>S                          LD A,'AB' ; STRING TOO LONG
```

Undefined Symbol

```
>U0784 210000      LD HL,GAP
>U0787 00          DEFB TRANS + 5
```

An undefined condition is treated as false

```
>U                 COND FDS
                   DEFB 0
                   ELSE
0106 01            DEFB 1
                   ENDC
```

Value Error

The value is illegal or too large for its context;

```
>V0788 00          DEFB 3/0
>V0789 0000        DEFW BEGIN SHR 17
>V078B 00          DEFB 100H
>V078C 3E00        LD   A,200H
>V078E CB47        BIT  9,A
```

Console Messages

The following messages may appear on the console:

Label: multiple def at line xxxx

N.B. If a label is a local label then the $ is not printed.
Thus 1$ produces
1: multiple def.....

Bad input, try again:
ZASM could not evaluate the imput to QUERY

ZASM: aborted
User typed CONTROL C

ZASM:symbol table overflow
Assembly is aborted - too many symbols defined

ZASM:cannot open source file
ZASM:cannot open included file
Probably results from incorrect file specification

ZASM:read error on source file
ZASM:write error, files closed
ZASM:cannot close files
Fatal errors. May be insufficient space on disc

ZASM:directory full
ZASM:Macro phase error
ZASM:COND/ENDC too deeply nested

INDEX

USER'S COMMENTS

To help Research Machines to produce the highest quality microcomputers, supporting software, and technical publications, we like to hear from users about their experiences with our products.

Do share your thoughts with us by jotting them down on the tear-off form on the next page. You can leave out your personal details, if you want to. Fold the form in two, seal it with a piece of adhesive tape, and put it in the post. No stamp is needed if you post it within the United Kingdom.

If you would like to give more information than we have allowed room for on the form, we will be very pleased to receive a separate letter from you. You can even use the form to ask for a post-paid envelope, if you wish.

Additional information will be most useful, if you give us as much detail as possible about your hardware configuration, software version number, or manual title, so that we can relate your comments to the correct products.

Seal with self-adhesive tape (not staples) along this edge.

RESEARCH MACHINES

MICROCOMPUTER SYSTEMS

Fold along this line.

Postage will be paid by licensee

Do not affix Postage Stamps if posted in Gt Britain, Channel Islands, N Ireland or the Isle of Man

1

BUSINESS REPLY SERVICE
Licence No OF32.

TECHNICAL PUBLICATIONS DEPT
RESEARCH MACHINES LTD
PO BOX 75 OXFORD
OX2 0BR

User's comments help us to improve our products. If you would like to make any comments, please use this reply-paid form.

Your comments:

Research Machines may use this information in any way believed to be appropriate and without obligation.

Although it is not essential, it would be helpful if you gave the following information:

Name.....

Organization.....

Address.....

..... Post Code.....

System: 380Z / 480Z / Network ☐ Cassette / 5.25" discs / 8" discs
(Delete as necessary)

RESEARCH MACHINES

MICROCOMPUTER SYSTEMS