```
**************************************************
***                                            ***
***            QSAL Assembler Manual           ***
***                                            ***
**************************************************
```

Manual Revision 1.0

June 1, 1983

Written by

Carl Galletti

CORRECTIONS

Please note the following corrections to this manual.

Throughout this manual "Q" is used instead of "QSAL". They may be thought of as the same. "QSAL" is a trademark of Computer Design Labs and "Q" is a shorthand for "QSAL".

On your disk you have three versions of QSAL, QSAL1.COM,QSAL6.COM, and QSAL7.COM. QSAL1 runs at 0100 hex and assembles programs at 06000 hex. You should hardly ever need this version. QSAL6 runs at 06000 hex and assembles programs directly for 0100 hex to 05FFF hex without using an offset. This is the version you should use most of the time. Rename it to "QSAL" or "Q". QSAL7 is provided to get an extra 4K before having to use an offset. However, since few assembly language programs are (or need to be) greater than 24K, it will most likely never be needed.

On page 8, % and # are not allowed.

On page 9, arithmetic currently evaluates from left to right NOT by the MDAS method mentioned.

On page 12, there is no "CON" data type. Same for section 2.3.2 and 4.7 (page 22).

On page 24, there is no IMAGE function. You must use the SAVE function in the DOS (after exiting QSAL, of course).

On page 25, there is no LOAD function.

On page 32, there is no "S." nor "L." function.

On page 32, section 5.7, there is no apostrophe function.

Sections 7.0, 8.0, and 9.0 have been replaced with new sections that look much different from the rest of the manual.

Section 10.0 has been removed.

In section 12.0, ignore "/BRKS->P" construction. Also, due to hardware limitations, there is no N,S,H,T, nor U command. The "R," command is also not functional.

Section 13 is not fully implemented.

The information provided in section 14 is for advanced programmers. If you wish to use it, be advised that you are on your own. We cannot provide support for this information in the form of tutorial.

NOTE: MOST OF THE ABOVE CHANGES WERE MADE BECAUSE THE MANUAL WAS INITIALLY WRITTEN FOR A SPECIFIC COMPUTER WHERE THE HARDWARE FEATURES WERE DIFFERENT FROM A STANDARD CP/M (R) SYSTEM AND CONSEQUENTLY SOME OF THE FEATURES COULD NOT BE MOVED OVER. OTHER FEATURES WERE NOT IMPLEMENTED IN THE FINAL VERSION BECAUSE OF TIME CONSTRAINTS.

TABLE OF CONTENTS

0.0 INTRODUCTION

Q is a one pass assembler for the Zilog Z80 microprocessor. It has many features which go beyond the conventional assembler's ability to translate mnemonics and pseudo-ops into native machine language instructions. Q is capable of special register handling and block structured notation as well as other features which have the potential of making assembly language easier to write and comprehend. Like any powerful tool it can also be misused to achieve the opposite effect. Since Q allows the programmer a great amount of freedom and power, he is also expected to assume a greater responsibility for its proper use. He is expected to follow the form used in examples and especially the section on Software Standards (15.0).

This documentation assumes enough familiarity with the Z80 registers and mnemonics and assembly language in general to be able to enter, assemble, and run conventional assembly language programs. Those not so oriented should first study an introductory text on the subject. "Z80 Assembly Language Programming" written by Lance A. Leventhal and published by Osborne/McGraw-Hill is an excellent choice and is also useful to the experienced assembly language programmer because of the description of the Z80 instruction set in chapter 3.

0.1 SURVEY OF FEATURES

Q is an extension of the standard Z80 assembler (ASM) which provides several unique features, including:

--Speed of execution.

--Long symbol names (up to 31 characters, upper/lower case distinct).

--Pascal-like "structured" expressions to permit self-documenting control structures.

Five types are introduced:

a. if...then...else
b. repeat...until
c. while...do
d. case...of
e. begin...end
(Note lower case letters.)

See chapter 6.0, "BLOCK STRUCTURED EXPRESSIONS", for details.

--Simple register handling expressions. Two types are introduced; thosethat are computed in 8 bit precision whose values are left in register A (A-expressions); and, those that are computed to 16 bit and whose values are left in the HL register (HL-expressions). In addition, results can optionally be stored in one or more places. For detailed information, refer to section 5.0, "SPECIAL REGISTER HANDLING EXPRESSIONS."

--Command mode, in which commands from the console are executed on a line by line basis.

--Immediate mode, in which statements preceded by "*" or "/" are interpreted as commands and executed immediately without adding to the run time code.

--Direct code generation into memory, with offset capability

--Symbolic debugging locally.

--Multiple instructions per line separated by ";" (Comments are introduced by "..." instead of ";"). For example: `INC DE; INC DE; INC DE ...skip 3`

--Conditional assembly

--Disassembly of code when using versions which include DEASM (optional).

--Features may be temporarily added by using the Extend capability (optional).

0.2 ENTERING Q

To evoke Q, type: A>Q ("A>" is a prompt of TPM)

The system will respond with the prompt "-", awaiting the first line of input from the console. At this point several types of response are possible. A source file may be brought in and assembled, a procedure file may be loaded, a loaded program may be run and debugged, code may be directly entered from the console, etc. This is the most versatile method of evoking Q.

0.3 AUTO ASSEMBLY

A file name may also be given in the command line when evoking Q,

```
A>Q TEST.PGM
```

Q is first loaded by the TPM operating system which in turn loads TEST.PGM. If TEST.PGM is a source file, Q automatically assembles it. If it is a procedure file, Q just loads it. It may also be a MAKE file.

A MAKE file is an ASCII type file which contains commands as they would be entered from the console while in Q. It may contain any instructions which could be entered from the console including directives to assemble/load other files. In fact, there is little difference between a source file and a MAKE file since they may each contain source instructions intermixed with console commands. In practice, convention dictates that files containing mostly source are called source files and those with predominantly console commands are MAKE files.

0.4 WRITING PROGRAMS IN Q

Although instructions may be entered and assembled directly while in Q, this is not the normal mode of operation. Entering code directly is only useful for patching or as a tutorial to examine the object code generated for a given source statement (see section 4.5 /LIST).

The normal sequence of actions for writing Q programs is similar to conventional assemblers. First, the source file is written using an editor to enter the source statements; then, Q is loaded and the source file assembled.

0.5 ERROR HANDLING

Q flags errors by re-writing the source line containing the error with a question mark (?) inserted at the point where it could no longer continue the assembly. Note that this is NOT always the precise point that the error is located. It is merely the point where it is impossible for Q to continue assembly of the remainder of the line/file. If no error can be found in the immediate area, check backwards from the "?". A "begin" without a matching "end" often causes this type of error.

1.0 STANDARD ASSEMBLY USING ZILOG MNEMONICS

It is possible for Q to function similar to a conventional assembler because it assembles standard Zilog Z80 mnemonics. These may be intermixed with more advanced Q statements as they are learned. Covered in this chapter are some basic points to follow when using either the standard assembly or advanced Q features.

1.1 FORMAT

All Q source lines contain up to three fields:

```
LABEL FIELD          STATEMENT FIELD          COMMENT FIELD
```

Each of these fields is optional but if included must be in the left-to-right order indicated.

The LABEL FIELD must start in the first column. It consists of a symbol name which must start with an upper case letter and may contain lower case letters, numbers, or underlines (which on some terminals are back arrows). A trailing colon (:) may be used but is not necessary. For example,

```
LABEL:          and
LABEL           are both treated as the same symbol.
```

Lines which do not have labels are allowed to start in the second column but good programming practice dictates that they are indented more, usually at least one tab (8 spaces) in.

Comments with 3 periods, "..." and may start in any column.

The statement field consists of one or more Q statements separated by semicolons (;). In its simplest form, a Q statement is the mnemonic representation of any Z80 instruction (for example: LD A,C).

Blank lines are allowed.

Z80 instructions and register names must be uppercase only. Some examples:

```
                                 ...This line is a comment only, beginning in the first column.
Label:   LD A,(DATA)             ...labels can be here also.
         CPL
Label_2                          ...the above line was a statement field only.  This label has no ":"
```

1.2 COMMENTS

All comments begin with three periods "..." as:

```
                ...this is a comment
```

Anything from the right of the "..." to the end of the line is part of the comment field.

1.3 MULTIPLE INSTRUCTIONS PER LINE

More than one instruction may be placed on a single line by separating each instruction/statement by a semicolon (;). For example:

```
Test:    LD A,(Status)
         BIT 0,A
         JP Z,Test
         RET
```

may be written as:

```
Test:    LD A,(Status); BIT 0,A; JP Z,Test; RET      ...comment
```

1.4 HEX VERSUS DECIMAL NUMBERS

The rule to distinguish between hex and decimal numbers is:

Any number beginning with a zero (0) is hex and any number beginning with - a non-zero is decimal.

Conventional assemblers usually put a trailing "H" to indicate hex. In Q a trailing "H" is optional and has no effect. The reason it allows this is so programs written in conventional assembly language are less trouble to convert.

```
Some examples:
0F000    ...hex
500      ...decimal
0500H    ...hex
014      ...hex
14       ...decimal
9        ...technically decimal but anything 9 or less is the same for hex or decimal.
```

Numbers representing addresses in JP, CALL, or ORG statements, or the START command, are always considered to be hex, even without the leading "0".

Some extra capabilities were added to Q which enabled it to assemble code having a percent (%) sign preceding binary numbers and a cross-hatch (#) preceding hex numbers.

```
...EXAMPLES:
        LD A,%11000011                    [3EC3]
        LD B,#48                          [0648]
        ...of course #48 and 048 are the same
```

It is strongly recommended to use the preceeding "0" as the means for indicating hex numbers.

1.5 ASCII VALUES

String values used in instructions are indicated by single quotes before and after. For example:

```
LD A,'X'    ...loads register A with the ASCII value for X (058).
CP 'C'                  ...compares register A with ASCII C (043)
```

1.6 NUMERIC VALUES, ARITHMETIC OPERATORS AND ORDER OF EVALUATION

Q allows some compile time evaluation of numeric values as used in expressions like

```
LD HL,2*(VALUE+512)  or  ORG $+0FF&0FF00  or  DEFB 2**3
```

There may be any number of terms and parentheses are used to group values. However, no spaces are allowed.

"My Dear Aunt Sally" rule or Multiplication and Division before Addition and Subtraction (MDAS). In either method Anding (&) and exponentiation were given the same order as multiplication and division.

Therefore:

```
LD BC,1+2*3                         ...[010700]
```

now results in loading BC with a 7 because 2 is multiplied by 3 before 1 is added.

Another might be to use

```
LD BC,(1+2)*3                       ...no good
```

However, this expression does not assemble and the reasoning is as follows. When Q scans the instruction from left to right, it gets to the point where it sees LD BC,(1+2), which is a valid and COMPLETE instruction. According to the assembler rules, there can be nothing else attached to this expression. It may only be followed by a space, semicolon, comment or end of line. When it next sees the *, it must flag this as invalid. Note that a "load BC with the value at address 3" is what the scanner sees as a valid expression. In other words, when Q sees an open paren, "(", in the operand field, it branches into the LD BC,(nn) portion of the evaluation tree which does not allow anything to follow the close paren, ")". The key in the branch is the open paren immediately following the comma. i.e., --->>> LD BC,(<<<---.

Another possibility which at first looks good and assembles without error is:

```
LD BC,((1+2)*3)                     ...[ED4B0900]
```

This expression evaluates to a 9 but note that the instruction generated is NOT a "load BC with 9" but a "load BC WITH THE VALUE AT ADDRESS 9", a big difference.

Alas, Q may be "tricked" into evaluating the LD BC,nn form by inserting some valid character other than the open paren as the character immediately following the comma. Acceptable possibilities are:

```
LD BC,3*(1+2)                                ...[010900]
            ...moving the *3 to the front
LD BC,1*(1+2)*3                              ...[010900]
            ...adding the neutral "1*"
LD BC,0+(1+2)*3                              ...[010900]
            ...literally adding a neutral zero
```

All numeric values are calculated as 2-byte quantities; if only one byte is needed, the value is truncated to the low-order portion. Numeric values may consist of hex or decimal numbers, symbols, single ascii characters in quotes (e.g. 'A'), "2**n" (2 raised to the nth power-where "n" is another numeric value), or the expression "value(------)" (to be described below). The above forms may also be preceded by a "-" sign, or combined and modified using addition (+), subtraction (-), multiplication (*), division (/), or anding (&) operations.

The expression "value(-----)" works as follows: Inside the parentheses is an instruction list which is executed immediately (at compile time) to put a value in R.HL which becomes the numeric value. The instruction list may be any number of instructions separated by semicolons and must not include undefined labels. An example is

```
DEFW value(P.GETVAL)*3
```

which makes a call to the subroutine GETVAL, multiplies the value in R.HL by 3, then stores it as a word variable in the memory.

If numeric values contain an undefined symbol, they must not also contain a modification to the symbol value. As an example, the statement

```
LD HL,NOTKNOWN+3              ...NO GOOD
```

where NOTKNOWN is undefined is an error. (This is because of the way Q handles undefined symbol references). Statements like

```
        LD HL,NOTKNOWN            ...OK
or      JP KNOWN+3                ...OK
```

where KNOWN is defined and NOTKNOWN is not, are fine.

1.7 INSTRUCTIONS NOT IMPLEMENTED

Not all of the Z80 instructions are directly supported by Q. Some instructions which are thought to be used rarely are left out to make the assembler somewhat smaller. If these instructions are needed by a program, there are two ways to implement them. The first and simplest is to substitute a BYTE variable (see section 2.4.1 for details) with the value of the instruction. For example, to create a "LD A,I" which generates 0ED 057,

```
                    BYTE 0ED 057        ...LD A,I
```

The following is a list of all instructions currently unimplemented:

LD A,I	...BYTE 0ED 057		
LD A,R	...BYTE 0ED 05F		
LD I,A	...BYTE 0ED 047	LD R,A	...BYTE 0ED 04F
NOP	...BYTE 00	IM 0	...BYTE 0ED 046
IM 1	...BYTE 0ED 056	IM 2	...BYTE 0ED 05E
IND	...BYTE 0ED 0AA	INDR	...BYTE 0ED 0BA
INI	...BYTE 0ED 0A2	INIR	...BYTE 0ED 0B2
OUTD	...BYTE 0ED 0AB	OTDR	...BYTE 0ED 0BB
RETN	...BYTE 0ED 045	RST 0	...BYTE 0C7
RST 10H	...BYTE 0D7	RST 18H	...BYTE 0DF
RST 20H	...BYTE 0E7	RST 28H	...BYTE 0EF
RST 30H	...BYTE 0F7	RST 38H	...BYTE 0FF

KNOWN BUG: Consider the following,

```
        ADD IX,HL
        ADD IY,HL
        ADD HL,IX
        ADD HL,IY
```

Although they are illegal instructions, Q does NOT flag them as errors. Instead it assembles them as the following LEGAL instructions (in respective order):

```
        ADD IX,IX
        ADD IY,IY
        ADD HL,HL
        ADD HL,HL
```

2.0 SYMBOLS, VARIABLES, AND CONSTANTS

2.1 SYMBOLS

There is only a subtle difference between labels and symbols. Labels are, technically, those symbols which are defined by the assembler as the next available address. All other symbols are assigned a value by the programmer or left unassigned. In Q, as in most assemblers, the difference is mainly academic since they are treated similarly. The syntax is therefore the same.

Symbols must be defined in some part of the progam by using a name which starts in the first column and begins with an upper case letter. The rest of the name may contain lower case letters, numbers, or underlines (which on some terminals are back arrows). A trailing colon (:) may be used but is not necessary.

Conventional assemblers make two passes, one chiefly to pick up symbol definitions and the other to do the actual assembly using those definitions. Q assembler, on the other hand, is a one pass assembler and therefore must handle symbols in a special manner. In Q, before symbols are defined they are called "forward references." If how they are handled is of interest, refer to section 2.5, FORWARD REFERENCES. The only consideration of concern to the user is discussed in the section on "B. and W. expressions", chapter 5.

2.1.1 SYMBOL TABLE

All symbols, whether defined or not, are placed in a buffer referred to as the symbol table. Each symbol is assigned a "type" byte which describes it. The following is a table indicating the significance of each bit in the type byte:

xxxx 0000	if Label:
xxxx 0001	if defined as BYTE (see section 2.4.1)
xxxx 0010	if defined as WORD (see section 2.4.2)
xxxx 0011	if PROCedure definition (see section 6.1)
xxxx 0100	if defined by DEFL (see section 2.3.3)
xxxx 0101	if defined by EQU (see section 2.3.1)
xxxx 0110	if defined by CON (see section 2.3.2)
xxxx 1xxx	if undefined
xx1x xxxx	if symbol reuseable (ZAPed-see section 4.8)
x1xx xxxx	if undefined, indicates forward reference chain exists
	if defined, indicates no references to symbol exist.
1xxx xxxx	if symbol is GLOBAL

The symbol table may be viewed by using the /MAP command, which is described in section 4.7.

2.1.2 UNDEFINED SYMBOLS

Undefined symbols may be detected by using the command:

```
/MAP 08
```

Since Q does not otherwise warn that a symbol is left undefined, this command should always be inserted at the end of the source code to detect those which are not intentional.

There are two symbols in Q which should always be left undefined. They are:

```
USERCC and EXTEND
```

2.2 "HERE" ($) SYMBOL

The "$" symbol (pronounced "here") does not have to be defined with a label. Its value is internally understood to represent the current value of the program counter. For example, a commonly used test mechanism is to insert an infinite loop operation in a program to signify that, "if the program got to 'this' place, something is drastically wrong; so, loop and don't do anything else". This may be done with a jump relative instruction which jumps to itself. That is,

```
JR $                    ...generates an 18 FE
```

In regular programming, a label might be defined with $,

```
Label:    DEFL $
```

which set "Label" equal to the current program counter.

2.3 CONSTANTS

Constants are those symbols representing values which are never changed by the program while it is running. There are three basic types: equates (EQU), constants (CON), and defined labels (DEFL).

2.3.1 EQU

An equate is a constant which, once defined, can not be changed. It is defined by assigning a symbol name in the label field and putting the letters "EQU" followed by the value in the statement field. For example:

```
ESCAPE:           EQU      01B       ...escape character
```

defines the symbol name "ESCAPE" as being equal to the value 01B which is the hexadecimal representation in the ASCII code for the escape character. After this definition, the letters "ESCAPE" can be used in place of the number "01B". Equates should be used in place of the actual number wherever possible to make changing the code much easier.

For instance, in the above example, if the program had to be moved to another computer which did not use the standard ASCII value for escape, the new value could be substituted for "01B" in the definition and the program reassembled. If the actual value, "01B", had been used instead of the equate, every occurrance of "01B" would have to be found and changed before reassembly.

2.3.2 CON

When an EQU is used in a "W." expression, it results in R.HL being loaded with the value pointed to by the EQU rather than the value of the EQU. Observe,

```
1122    EQUATE:  EQU 01122
0111             W.EQUATE                            [2A2211]
                 ...EQUate is used as a pointer to the word to--->^^^^^^ be put into R.HL
```

This is a peculiarity of the way Q handles EQUs.

At some point in Q's evolution, it was decided to have a way to load R.HL with the actual EQU. However, to not require that existing programs be changed, the CON function was added.

A symbol defined with CON acts the same as an EQU in all but "W." expressions; where, R.HL loads the actual value of the symbol rather than the value pointed to. The following demonstrates the difference between CON and EQU:

```
1234    CONSTANT:        CON 01234...defines a constant as opposed to an equate
1122    EQUATE:          EQU 01122
010E                     W.CONSTANT                  [213412]
                                   ...The CONstant is put into R.HL as an address-->^^^^^^
                                   ...i.e., R.HL now has the value of CONSTANT.
0111                     W.EQUATE                    [2A2211]
                                   ...EQUate is used as a pointer to the word to--->^^^^^^
                                   ...be put into R.HL. I.e., R.HL is loaded with the word
                                   ...at the memory location EQUATE is pointing to.
```

The CON function was to have the same effect on "B." expressions. Unfortunately, due to an oversight, it does not. For A-type expressions it works the same as the EQU. This is a bug which will hopefully be corrected in future versions. However, for now, CON defined constants should not be used in A-type expressions such as,

```
LetterA:  CON        041       ...Ascii for letter A
          B.LetterA->@HL       ...DON'T DO THIS !!!!!!!!!
```

2.3.3 DEFL

The "DEFL" or "defined label" is exactly the same as the "EQU" except that the value assigned to the symbol name may be changed by redefining it with another DEFL statement. For instance,

```
ORG 0100 ...current program counter is forced to 0100 (hex).
```

```
HERE:   DEFL $                  ...HERE is defined as the current value of the program counter.
        LD HL,HERE              ...loads register pair HL with 0100
        .------|
        .      |->>>>>>>...some program statements here
        .------|
HERE:   DEFL $                  ...HERE now takes on the new current value of the program
                                ...counter which for this example will be assumed to be 0152.
        LD HL,HERE              ...now loads HL with 0152.
```

2.4 VARIABLES

Variables are defined by the following format:

```
<Symbol name>     <keyword>          <initial values>
```

The symbol name can be any valid symbol name starting in the first column. The keyword may be: BYTE, WORD, DEFB, or DEFW. One or more "initial values" may be assigned. The first value is assigned to the symbol name and the succeeding ones occupy the next available bytes/words.

2.4.1 BYTE

The "BYTE" variable definition reserves one memory location (byte) whose address can be referred to by the symbol name. It initially has the value indicated in the definition. For example,

```
Count:    BYTE 0
```

reserves the memory location at the current program counter for the symbol name "Count" and initializes the location to 0. From then on the value referred to by "Count" may be changed by writing a new value, such as,

```
LD A,(Count)                ...gets present value
INC A                       ...increases it by 1
LD (Count),A                ...puts back the new value
```

A numeric value may be assigned to a BYTE as,

```
StartCount:         EQU 022
Count:              BYTE StartCount     ...initialized to 022 hex
```

Also, more than one byte may be defined. Example:

```
TABLE:  BYTE 5 0 StartCount+3
```

defines TABLE as a 'BYTE' variable and in addition assembles and stores the BYTE expressions which follow into each successive memory location. If "TABLE" was assigned the memory location 5000, then a 05 would be in location 5000, a 0 in location 5001, and a 025 in 5002.

By putting a number enclosed in square brackets, [], after the value, multiple byte locations will be initialized to the value. For example,

```
Buffer:  BYTE 0[22]        ...reserves 22 bytes starting at "Buffer" and initializes them to 0.
Buf:     BYTE [33] ...If the value is omitted, then the bytes are reserved but not initialized to any value.
```

2.4.2 WORD

Word quantities work similarly, e.g.:

```
ADR_TABLE:  WORD 0100 Vector+Index TABLE 0[4] ...reserves 6 words, clears last 4 to 0.
REG_SAVE:   WORD [11]
```

except that instead of one byte, one word (two bytes) is reserved. That is, REG_SAVE is 11 words or 22 bytes long.

2.4.3 DEFB AND DEFW

These are similar to BYTE and WORD respectively. However, they should not be used in new code. Their purpose is to make the conversion from existing ZILOG assembly language to Q easier by not requiring these symbols types to be converted.

2.5 FORWARD REFERENCES

The subject of forward reference must be considered in the development of any assembler. If a symbol name is referred to before it is defined, how does the assembler know what address or value to use? Conventional two pass assemblers handle this by getting all the symbol name definitions in the first pass. In fact the primary purpose of the first pass is to get the symbolic definition. Q is able to define symbols on the fly which is one reason why it is faster than conventional assemblers.

When Q encounters a reference to a symbol name which is undefined, it inserts a zero byte or word (depending on the operation) and marks this location as the beginning of a chain. The next time that symbol is referred to, instead of a zero, the address of the previous reference (or in the case of one byte references, a relative offset) is used. As a result, a chain is built. When Q finally gets to the definition, the chain is followed backward and the correct value substitued until all references have been resolved. Subsequent references to the symbol name get the correct value immediately.

Before a label is defined, all references to it must be of the same type; that is, either all one-byte (relative) references, as in JR instructions, or all two-byte (absolute) references, as with JP or CALL or two-byte loads. This is because, depending on the type of reference, either a one or two-byte chain is set up in order to resolve the references when a symbol becomes defined. (Other than with JR, one-byte references to undefined symbols are not supported.)

3.0 ASSEMBLER DIRECTIVES

Assembler directives are commands given to the assembler to perform some special manipulation.

3.1 ORG

The ORG or "origin" directive instructs Q to set the current value of the program counter to the value following. For example, if a program started assembling and got up to address 01F0, and it was necessary to then insert the next piece of code starting at location 1000:

```
01F0            ORG 01000
1000                                ...program counter now set to 1000
```

3.2 DEFM

DEFM stands for "defined message". It has the form:

```
Label:   DEFM 'This is any "ASCII" text'
 OR
Label:   DEFM "This is to allow 'single quotes' in the message"
```

For each character in the message, it inserts a byte into memory which corresponds to its ASCII code. Note that either single or double quotes may be used. If the message starts with a single quote ('), the body of the message will consist of all characters following, including double quotes ("), until the next single quote is reached. The reverse is true if the message starts with a double quote.

3.3 DEFT

DEFT or "defined text" is essentially the same as DEFM except that it inserts a byte at the beginning of the message which is a count of the number of characters in the message. This is useful for loading a register with the count byte and decrementing until zero for each message character transferred.

```
5000   Msg:      DEFT 'HELLO'
```

Generates the following code at address 5000:

```
5000   05 48 45 4C 4C 4F ...first byte (05) is the count byte.
          H  E  L  L  O  <<<--- ASCII
```

3.4 DEFS

"Defined storage" reserves a specified number of bytes in memory without changing them from their existing values. It is primarily used to reserve space for buffers.

```
Buffer:    DEFS 0100
```

Reserves 100 hex (256 decimal) bytes.

3.5 WRITE

While assembling, it is often convenient to have a message, address, or some other data printed out to the console and an output file if open . The WRITE directive can be used to do this. It should be preceeded with a slash (/) or star (*)--see chapter 4 for more details.

```
-/WRITE "TITLE -- I/O Routines"
```

Will print the message: TITLE -- I/O Routines

Also, byte and word quantities may be output. Byte values are output from the A register by any valid A-expression (see section 5.4) and word values come from the HL register pair by any valid HL-expression (section 5.5).

Strings, A-expressions, and HL-expressions are separated by commas in the WRITE statement:

```
-/WRITE "IMAGE FROM ",^Begin," for ",LD A,(Count)," bytes"
```

If the address of "Begin" was 0100 and a 3F was in "Count", the following would be output:

```
IMAGE FROM 0100 for 3F bytes
```

Any "begin-end" blocks, separated by commas, may also be included within the 'write' statement to do calculations or output data @DE.

The procedures 'WSbgn', 'WSend', 'Copy', 'HLout', and 'Outhex' (which are in Q) must be present (and in the symbol table during compilation) in order to use the 'write' statement.

The WRITE keyword may be substituted with a question mark (?), as in:

```
-/?"Print this string"
```

The write statement may also be used by a program during its execution if Q is resident:

```
                ?"VALUE=",W.VALUE," X=",R.C+2
assuming VALUE=9999 and R.C=030,
then            VALUE=9999 X=32
```

will be written during program execution.

Note that during the execution of a program, if the write statement is used to output the value of register pair DE, special handling is required. Since R.DE is used by the write statement as a pointer to the output buffer, its value is saved in LASTDE before executing the write statement and then restored afterward. If the true value of R.DE is to be written, use LASTDE.

```
?"The true value of DE is ",W.LASTDE
```

This prints the correct value of register pair DE.

Any A- or HL- expression in the write statement will use R.A and R.HL. Therefore, they may easily be overwritten in the write statement. For example,

```
?"BC= ",R.BC," HL=",R.HL
```

is incorrect because HL was destroyed when R.BC was output. The same thing happens with R.A except a little worse. Once an A-expression is written, R.A is no longer valid - even if R.A was the only thing written. Examine:

```
?"R.A=",R.A,"   ",R.A
```

If R.A contains 055, the following is written:

```
R.A=55   35
```

The 35 is the ASCII for the number 5 which was the last thing in R.A (used to output the second 5 of the last A-expression).

3.6 GLOBAL

Symbols are made global throughout an assembly by appearing in a GLOBAL statement of the form

```
GLOBAL symbol1 symbol2 etc.
```

The word GLOBAL starts in any column but the first. For the best speed of assembly, put GLOBAL statements after the symbols in them have been referenced. There is no EXTERNAL statement.

3.7 stop

Placing a "stop" in the source code causes Q to generate a BYTE OFF. This can be used in places where reaching that point indicates something major has gone wrong. It provides a means of stopping the action similar to a breakpoint but without having to enter it repeatedly. It may be in any column other than the first.

4.0 IMMEDIATE MODE

The immediate mode in Q is a command mode with no relationship to the Z80's immediate addressing mode. In Q, immediate commands are those which are executed immediately at assembly time rather than run time.

The following is a list of immediate commands. They are entered from either the console, a MAKE file, or a program file. All must start with either a "/" or a "*", and be the first character on a line. ("/" and "*" are generally interchangeable except during conditional assembly, described in chapters 8). In addition, any instruction or instruction list may be executed immediately (from the console or during compilation) by entering

```
/INSTRUCTION.LIST    ...e.g. /LD B,2; TEST()
```

Some instructions or commands, in particular those needing to do some calculations or immediate execution, may temporarily use a few extra bytes in the code area (where the object code is generated). However, ORG, DEFS, EQU and DEFL statements, and the QBUG commands, do not.

4.1 // [EXIT FROM Q]

While in the "/" or "*" modes, a "//" may be used to exit Q. "//" exits the program, closing files and returning to the TPM operating system.

/EOF (in a file) marks the end of file. (It will be closed at that point.)

4.2 /* [ENTER DEBUG MODE]

Q contains a symbolic debugger. To enter debug mode, from the command mode type "/*" (this can also be in a file; any open input or output files will be closed). The prompt is a "*". From debug mode, any instruction (or instruction list) that is typed will be executed immediately, with registers preserved from one line to the next (including any changes made). All of the commands beginning with a "/" are also valid. In addition, there is a set of special QBUG commands (see chapter 12.0 for info).

4.3 *Q [EXIT DEBUG MODE]

If in debug mode ("*" prompt), type "Q" to return to normal command mode ("-" prompt) or "//" to exit to TPM.

4.4 /DO

The "/DO" command form is used to assemble a file (and/or execute commands from it). Therefore,

```
/DO <source file name>
```

will assemble <source file name>. Note that the file may contain assembler commands which will be executed as if they were entered from the console. It may also contain other files to assemble.

4.5 /LIST

This command has many useful purposes. It can be used for self tutorial by entering the command

```
/LIST      ...without a file name will generate a listing to the console.
```

Instructions entered are immediately assembled and the results displayed on the console. This mode resembles an interpreter in its ability to give instant feedback about the acceptability of the source. However, it is much more powerful in that it also gives a display of the generated code. This feature, if wisely used, accelerates the learning process.

```
LD A,033  ...sample line of code with list display      [3E33]
```

Note the numbers between the brackets which indicate the code generated from the source.

By entering files to be assembled (with /DO command), the /LIST can also be used as a debugging tool, catching spots where code is being overwritten by watching the addresses on the console. Of course an entire listing can be generated and printed to accomplish the same, however, a partial console listing is often adequate and faster.

The "/LIST" command is also used to obtain a source listing with a display of generated object code. While in immediate mode and before entering "/DO", enter:

```
/LIST <listing file name>
```

CAUTION: The <listing file name> should be DIFFERENT from the source file name. When the above command is entered, Q opens a file with the specified file name. If that file name already exists, then Q will erase it prior to opening the new file. Therefore, if the listing and source file names are the same, THE SOURCE FILE WILL BE ERASED!!!

For example, if a listing is needed for the source file named PROG.S,

```
/LIST L.PROG        ...opens the list file "L.PROG" erasing any previous "L.PROG".
/DO PROG.S          ...simultaneously assembles PROG.S and generates the listing to L.PROG.
/LIST END ...turns the listing off and closes L.PROG
```

To get a printed listing, first list to a file as above, then print the file. Note the use of "/LIST END" in the above. There are two other variations:

```
/LIST OFF turns the listing off but leaves the listing file open
```

```
/LIST ON turns the listing back on after a /LIST OFF
/LIST END turns the listing off and closes the listing file
```

Therefore, selective portions of a source file may be listed by using "/LIST OFF" and "/LIST ON". Also multiple list files may be generated by:

```
/LIST L.FILE1
.
.
/LIST END
/LIST L.FILE2
.
.
/LIST END
```

Another list option, /LIST SHORT, gives an easy to read list where the generated code is not required but the addresses are. Without the short list a piece of sample code looks like this:

```
0100     Example: LD A,04                          [3E04]
```

With the short list the above code appears:

```
0100     Example: LD A,04
                         Notice the lack of listed code here    ^^^^^^
```

4.6 /START

/START name or /START 0hhhh causes control to be transfered to the given address, closing files first.

4.7 /MAP

/MAP [mask] or /MAPA [mask] (where mask is optional and will be described below) will display a compact listing of the source symbols and their addresses (or defined values) plus a byte in hex representing the symbol type on the console (and in a file if open for listing). /MAP prints symbols in the order of their occurrence, while /MAPA prints them alphabetically. The bits in the 'type' byte have meaning as follows:

xxxx 0000	if Label:
xxxx 0001	if defined as BYTE (see section 2.4.1)
xxxx 0010	if defined as WORD (see section 2.4.2)
xxxx 0011	if PROCedure definition (see section 6.1)
xxxx 0100	if defined by DEFL (see section 2.3.3)
xxxx 0101	if defined by EQU (see section 2.3.1)
xxxx 0110	if defined by CON (see section 2.3.2)
xxxx 1xxx	if undefined
xx1x xxxx	if symbol reuseable (ZAPed-see section 4.8)
x1xx xxxx	if undefined, indicates forward reference chain exists

```
                                  if defined, indicates no references to symbol exist.
              1xxx xxxx           if symbol is global
```

The above bit definitions are used for the [mask] byte used in the ZAP ALL and PACK ALL commands below.

4.8 /ZAP

```
        /ZAP <list of symbol names>
e.g.
        /ZAP SYM1 SYM2
```

effectively deletes (zaps) those symbols from the table, although they still take up space in memory.

```
        /ZAPALL [mask]            ...zaps all symbols not masked out.
                                  ...space between ZAP and ALL is optional
                                  ...NOTE: the space between the ZAP and ALL is
                                  ...optional.
```

Use the bits as described in section 4.7 (/MAP) to compose the [mask] byte. The bits are active true. For instance, to ZAP all symbols which are in bank 2, use

```
        /ZAP ALL 010              ...all bank 2 symbols are zapped.
```

Note that undefined symbols cannot be zapped.

4.9 /PACK

The "PACK" series of commands usually follow the ZAP command and are used to pack the symbol table into minimum space by saving only undefined symbols, symbols masked out, and, except for /PACK ALL, global symbols.

"/PACK" gets rid of all symbols which are not global or undefined. In other words, it eliminates all defined local symbols.

"/PACK ALL" will pack even the global symbols. The space between PACK and ALL is optional.

"/PACK [mask]" packs the symbol table saving only undefined symbols and global symbols which have not been zapped or masked out.

"/PACK ALL [mask]" packs the symbol table saving all symbols not zapped or masked out.

Note: undefined symbols cannot be zapped or eliminated through packing.

4.10 /CLEAR

/CLEAR clears the entire symbol table.

4.11 /IMAGE

Q is able to IMAGE a "object" file that uses following format:

/IMAGE <filename> $(?(symbol:) addr1 addr2)

where $(--) ...means some no. of' (one or more of), and
?(--) ...means optional'

This command will
--image a file of name <filename>
...between and including memory addresses addr1 and addr2 (up to 16 sets of addr1,addr2 pairs)

Also:
--"symbol:" has to do with associating an offset with addr1 and addr2

this is described
in more detail in chapter 9.0, Assembling With Offset'.)
--addr1,2 may be either a hex number, an HL-expression, or a "value(---)" expression
(as described in section 1.6) of 80H, otherwise the record length is set to 200H

Examples:

```
/IMAGE TASK1 0100 03000 04000 05000
```

images a file named TASK1 with the code from memory location 0100 to 03000 and the code from memory location 04000 to 05000.

```
/IMAGE FILEX ^START ^END
```

images a file named FILEX with the code beginning at the memory location symbolized by START and ending at the location indicated by END.

4.12 [mask] --- not a command

The form and meaning of the mask allowed in some of the commands above is more extensive than indicated. The full form is as follows (where the brackets indicate optional constructions):

mask=[c [m]] [R=[n1][,n2]] ...where c and m are one-byte hex numbers.

c is compared with the 'type' byte while m is used as a mask. A symbol is accepted if the bits of c which are '1' in m agree with the corresponding bits in the 'type' byte. In other words, the mask, m, is anded with c, the result is compared to the type byte, and all symbols whose type byte compares true are accepted.

If m is not given, it is set equal to c; if neither c nor m is given, they are 0 by default (and thus all symbols are accepted).

"R=n1,n2" specifies that the symbol must be within the range n1-n2 (inclusive) to be accepted. n1 and n2 are either HL-expressions or hex numbers, e.g. ~L0+3. (n1 defaults to 0 and n2 to 0FFFF.)

Examples:

```
/MAP 8
```

will produce a map just containing all the undefined symbols.

```
/MAPA 2 7 R=,5000
```

will produce an alphabetic map just containing symbols defined as WORDs with addresses <=05000.

```
/ZAPALL R=^ABC,^XYZ
```

will zap all symbols whose values are between ABC and XYZ.

```
/PACK 0 40
```

will pack the symbol table saving only global symbols which have been referenced (as well as any undefined symbols).

4.13 /LOAD

Q is able to LOAD any file (typically an object file) into RAM memory and it uses following format:

```
/LOAD <filename>
```

This command will:
load a file of name <filename> into memory starting at location 0100.

5.0 SPECIAL REGISTER HANDLING EXPRESSIONS

This chapter deals with special symbolic representations which are peculiar to Q and not found in any other language. In general, they help in visualizing data flow by achieving a degree of graphic representation not realized by conventional assembly language.

There are two types of implicit register operations known as A-expressions and HL-expressions. A-expressions assume that certain operations are performed with the accumulator (register A). Likewise, HL-expressions assume operations with the HL register pair. Both types make heavy use of the special register designations described in section 5.1.

The expressions presented in this chapter make use of the following general form:

```
<LOAD PART> <MODIFIER PART> <STORE PART>
```

where the

LOAD PART causes an implicit load of either the A or HL registers

MODIFIER PART causes a modification to the A or HL registers, usually after it has been loaded

STORE PART causes the value in the A or HL registers to be stored in either memory or another register/register pair. The load part may begin in any column other than 1, however, good form calls for indenting one or more tabs. The modifier and store parts are optional.

Whether A or HL is used as the implied register is usually apparent by the form of the load part. In certain instances where a symbol is used in the load part, the type may not be clear since whether it is a byte or word quantity is unknown. In these cases the modifier and/or store parts make the type of operation clear. However, since these two parts are optional, it is still possible to have a load part who's operation is unclear. At such times it is the responsibility of the programmer to make it obvious by using appropriate comments.

5.1 ADDITIONAL REGISTER DESIGNATIONS

There are expressions in Q which refer to register names in places where it could just as easily be a symbol. For this reason, register designations have been added to Q. In general,

R.r	...refers to register(s) r, where r is A, B, C, D, E, ...H, L, BC, DE, HL, IX, or IY.

That is,

R.A	...the accumulator, register A		
R.B	...register B	R.C	...register C
R.D	...register D	R.E	...register E
R.H	...register H	R.L	...register L
R.BC	...register pair BC	R.DE	...register pair DE
R.HL	...register pair HL		
R.IX	...register pair IX	R.IY	...register pair IY

5.2 @ EXPRESSIONS

@ expressions are actually not expressions at all but parts of A or HL expressions. They may be used as either load parts or store parts and are always used in conjunction with a register pair which is used as a pointer into memory.

By placing an at-sign (@) in front of one of the special register pair designations, that register pair is used as a pointer to a location in memory. If it is used as a load part, then R.A is loaded with the byte pointed to by the register pair. For example,

```
@R.HL     ...generates the same code as LD A,(HL) but is easier to type
          ...and aids in other expressions yet to be covered.
```

There are several variations. First, for @ expressions, the "R." is optional. So,

```
@HL and @R.HL are the same and the former is the preferred form.
```

Also, if an offset is to be added to the register pair first, use

```
@(R.HL+2) ...which increments HL twice before loading R.A with the
            ...byte pointed to by R.HL.
```

Note that in this example the "R.HL" cannot be substituted by the "HL". It is only when the @ and register designation are not separated by the paren, "(", that the "R." may be dropped.

Register pairs can also be used as pointers to word locations where they load R.HL with the word pointed to. These operations are part of HL-expressions which are covered in detail in section 5.5. To specify that a word is to be stored in R.HL rather than a byte in R.A, the @ sign is followed by a 2,

```
@2DE               ...loads R.HL with the word pointed to by R.DE.
@2(R.HL+2)...first increments R.HL by 2, then loads itself with the
                   ...word it is pointing to.
```

When used as store parts the byte in R.A (for A expressions) or the word in R.HL (for HL expressions) are stored into the memory location(s) pointed to by the register pair following the @. For example,

```
R.A -> @HL...stores R.A where HL is pointing to
R.HL -> @2DE       ...stores R.L where DE is pointing to and R.H at DE+1
                   ...R.DE is restored to its starting value.
```

5.3 UP ARROW (^)

When an up arrow precedes a symbol name, it usually indicates that R.HL is loaded with the ADDRESS of that symbol. The exception is when the symbol name is defined as a constant (i.e., CON, EQU, or DEFL) in which case R.HL is loaded with the value of the constant. However, examine a listing of Q and observe that, when the constant is defined, its value is placed in the far left 4 columns; the same place that symbolic addresses are put. Therefore, it is obvious that

THE UP ARROW (^) LOADS THE VALUE THAT A
LISTING WOULD PLACE IN THE ADDRESS COLUMNS.

In the case of byte, word, or label values, it is the address; but for constants, it is the actual value. For example,

```
0765    PROM_ADR:        EQU 0765 ...prom subroutine
0100    COUNT:           BYTE 012  ...count
                         ^PROM_ADR         ...loads R.HL with 0765
                         ^COUNT            ...loads R.HL with 0100
```

5.4 A-EXPRESSIONS

A-expressions deliver a value to R.A and consist of a load part followed by any number of modifier parts and/or store parts. These parts are described as follows:

Load Parts		Mnemonic Equivalent
@RBC or @BC	....	LD A,(BC)
@RDE or @DE	....	LD A,(DE)
@RHL or @HL	....	LD A,(HL)
@RIX or @IX	....	LD A,(IX)
@RIY or @IY	....	LD A,(IY)
@RIX(n) or @IX(n)	....	LD A,(IX+n)
@RIY(n) or @IY(n)	....	LD A,(IY+n)
		where n is an arithmetic expression for a 1-byte quantity (see section 16)
		Examples: (OFF EQU 012 and SS EQU 032)
@IY(OFF)	....	LD A,(IY+012)
also		@hl where "hl" stands for an HL-load part
(see 'HL-Expressions' below)	....	Load HL,--- LD A,(HL)
RB (or RC,D,E,H,L)	....	LD A,B (or LD A,C etc)
RA	....	No code
BVARNAME	....	LD A,(VARNAME)
VARNAME (if defined as a BYTE)	....	LD A,(VARNAME)
BPROCEDURE() (see 'Procedures')		CALL PROCEDURE
00 (this affects the flags)	....	XOR A
99 or other decimal number	....	LD A,99
0C3 or other hex number	....	LD A,0C3H
'Z'	....	LD A,'Z'
CONST (defined by "EQU" or "DEFL")	....	LD A,CONST
-a where "a" stands for an A-load part	....	Code for load part, NEG

Modifier Parts		Mnemonic Equivalent
--------------		-------------------
+carry+ R.B or C, etc........	....	ADC A,B etc
+carry+ (HL) or (IX+n), etc........	....	ADC A,(HL) etc
+carry+ 99 (or other constant)	....	ADC A,99
+carry+ @HL	....	ADC A,(HL)
+carry+ @IX (or @IY)	....	ADC A,(IX); (or ADC A,(IY))
+carry+ @IX(n) (or @IY(n))	....	ADC A,(IX+n); (or ADC A,(IY+n))
+carry+ @^DATA (or other HL-load part)	....	LD HL,DATA (or other HL-load)
		ADC A,(HL); (*see note below)
-carry- (options are same as +carry+)	....	SBC A,---
+1	....	INC A
-1	....	DEC A

Note: the space between the + or - and the 1 is significant Omitting the space generates the single byte increment/decrement whereas, putting it in generates a two byte ADD A,---

+ (options are the same as +carry+)	....	ADD A,---
- (same as +carry+)	....	SUB A, ---
XOR (same as +carry+)	....	XOR A,---
& or AND (same as +carry+)	....	AND ---
| or OR (same as +carry+)	....	OR ---
= or :: (same as +carry+)	....	CP ---
(except note "=0" below)		
= 0 (or :: 0)	....	OR A
*2	....	ADD A,A
/2	....	SRL A

Store Parts		Mnemonic Equivalent
-----------		-------------------
-> BABC	....	LD (ABC),A
-> ABC (where ABC prev classified as BYTE)		LD (ABC),A
-> @RBC or @BC	....	LD (BC),A
-> @RDE or @DE	....	LD (DE),A
-> @RHL or @HL	....	LD (HL),A
-> @RIX or @IX (same with IY)	....	LD (IX),A; (or LD (IY),A)
-> @RIX(n) or @IX(n) (same with IY)	....	LD (IX+n),A; (or LD (IY+n),A)

also

-> @hl where "hl" stands for an HL-load part		
(see 'HL-Expressions' below)	....	Load HL,---
		LD (HL),A; (*see note below)
-> RA	....	No code
-> RB (or RC, D, --- etc)	....	LD B,A (etc)

Additional 8-bit stores may be indicated directly by:

RA -> @HL -> SAVE -> RB	....	LD (HL),A
		LD (SAVE),A; LD B,A

5.5 HL-EXPRESSIONS

HL-expressions deliver a 16-bit value to the HL register. They also consist of a load part followed by any number of modifier and/or store parts.

	Load Parts	Mnemonic Equivalent
	----------	-------------------
	R.DE	LD H,D; LD L,E
	R.BC	LD H,B: LD L,C
	R.HL	No code
	R.IX	PUSH IX; POP HL
	R.IY	PUSH IY; POP HL
	^VARNAME	LD HL,VARNAME
	W.VARNAME	LD HL,(VARNAME)
	VARNAME (where VARNAME previously defined as WORD)	LD HL,(VARNAME)
	W.PROCEDURE() (see 'Procedures' section below)	CALL PROCEDURE
	W.999 (or other decimal number)	LD HL,999
	W.0EFF (or other hex number)	LD HL,0EFF
	@2HL goes 1 level indirect	LD A,(HL); INC HL
		LD H,(HL); LD L,A
		(*note use of R.A)
	@2BC or @2R.BC	LD H,B; LD L,C
		LD A,(HL)
		INC HL; LD H,(HL)
		LD L,A
		(*note use of R.A)
	@2DE or @2R.DE	EX DE,HL; LD E,(HL)
		INC HL
		LD D,(HL); DEC HL
		EX DE,HL
	@2IX or @2R.IX	LD L,(IX); LD H,(IX+1)
	@2IY or @2R.IY	LD L,(IY); LD H,(IY+1)
	@2IX(n) or @2R.IX(n)	where n is an expression for a 1-byte numeric value
	@2IY(n) or @2R.IY(n)	(see 'Numeric Values' below)
		LD L,(IX+n)
		LD H,(IX+n+1)
		[or LD L,(IY+n); LD H,(IY+n+1)]
also		
	@2hl	Load HL,---, where "hl" stands for an HL-load part
		@2HL (as above)
	^"string"	CALL #1; DEFT 'string'
		(effectively points HL to in-line assembled DEFT)
		#1: POP HL
	(R.HL+R.BC)*2	whatever is generated by HL-expr

Modifier Parts		Mnemonic Equivalent
+carry+ R.BC (or DE or HL or SP)	....	ADC HL,BC etc.
-carry- R.BC (or DE or HL or SP)	....	SBC HL,BC etc.
+ R.BC (or DE, HL, SP)	....	ADD HL,BC etc.
+ 1	....	INC HL
+ 2	....	INC HL; INC HL

Note: a + or - with a number less than 7 is not very efficient since 7
bytes of code are generated whereas, using +2 and/or +1 several times
will save bytes.

R.HL+2+2+2	....	only generates 6 bytes, all INC HL.
and R.HL+6	....	generates 7 bytes.
- R.BC (or DE, HL, SP)	....	OR A ,Reset Carry, SBC HL,BC (etc.)
- 1	....	DEC HL
- 2	....	DEC HL; DEC HL
+ R.[B] or other A-load part or +(R.A&0F)	....	LD A,[B] etc. (Note: A-expression may reuse R.HL) ADD A,L; LD L,A; JR NC,$+3; INC H
		- R.[B] or other A-load part or
(A-expression)	....	LD A,[B] etc. CPL; ADD A,L; LD L,A; JR C,$+3; DEC H; INC HL
*2	....	ADD HL,HL
/2	....	SRL H; RR L
= 0 (or :: 0)	....	LD A,H; OR L; (*note use of R.A)
= R.BC (or R.DE)	....	OR A; SBC HL,BC (or DE); ADD HL,BC

Store Parts		Mnemonic Equivalent
-> R.BC (or R.DE)	....	LD B,H LD C,L
-> R.IX (or R.IY)	....	PUSH HL; POP IX
-> @2BC or @2R.BC	....	LD A,L; LD (BC),A; INC BC; LD A,H; LD (BC),A; DEC BC; ...(*note use of R.A)
-> @2DE or @2R.DE	....	EX DE,HL; LD (HL),E; INC HL LD (HL),D; DEC HL; EX DE,HL
-> @2IX or @2R.IX (or IY)	....	LD (IX),L; LD (IX+1),H
-> @2IX(n) or @2R.IX(n) (same with IY)	....	LD (IX+n),L; LD (IX+n+1),H
-> W.VARNAME	....	LD (VARNAME),HL
-> VARNAME (if defined as WORD)	....	LD (VARNAME),HL
-> R.HL	....	No code

Note: Incrementing and decrementing R.HL does not set the flags; thus, the expressions: (HL-load part) +1, +2, -1, or -2 cannot be used to set the flags for a relation test.

Note: Some HL-expressions can modify R.A; thus R.A+@(@2HL) or R.A+@(^DATA+32) don't do the addition on the original R.A.

5.6 "S." AND "L." EXPRESSIONS

These expressions are used as a shorthand for @ expressions. "L." can mnemonically be referred to as "load the <?> register with @HL" and "S." can mnemonically be referred to as "store the <?> register @HL". For example,

```
0114            L.B                                                   [46]
                ...load R.B with (HL), or LD B,(HL)
0115            S.B                                                   [70]
                ...store R.B at (HL), or LD (HL),B
```

"+" or "-" used to the right of a register in an "L." or "S."

expression means "INCREMENT" or "DECREMENT" R.HL respectively. For example,

```
0116            L.C+B                                           [4E2346]
                ...generates        LD C,(HL)
                ...                 INC HL
                ...                 LD B,(HL)
                ...an often used sequence, also...
0119            S.E+D-                                        [7323722B]
                ...generates        LD (HL),E
                ...                 INC HL
                ...                 LD (HL),D
                ...                 DEC HL
                ...preserves R.HL
```

"&" causes a logical AND between R.A and the byte @HL and "|" causes a logical OR between R.A and the byte @HL. For example,

```
011D            L.A+|+&-                          [7E23B623A62B]
0123            S.A+|+&-                          [7723B623A62B]
                ...notice that the &,|,-, and + have the same function in both
                ...the "L." and "S." expressions.
```

5.7 INCREMENT BY APOSTROPHE (')

The appostrophe (') may be used to indicate increment immediately following an @RR expression where RR is a 16-bit register pair (e.g., DE). For example,

```
0129            @DE'                                                [1A13]
012B            @HL'                                                [7E23]
                ...Note that a double ' (e.g., @DE") does not work
```

5.8 B. AND W. EXPRESSIONS

The "B." and "W." prefix must be used in circumstances where there is an ambiguous forward reference. Until a symbol name is defined, whether it is a byte or word quantity is ambiguous. Q is able to handle this only if the programmer removes the ambiguity with the "B." or "W." prefix.

Placing the "B." in front of the symbol name identifies it as a proposed byte quantity. Q will properly build a single byte relative chain until the symbol is defined. The "W." has a similar effect in identifying word quantities. Examples:

```
        B.BSYM->R.A          ...= LD A,BSYM
        W.WSYM->R.HL         ...= LD HL,WSYM
```

Another type of expression using these prefixes has to do with procedure calls. By placing a prefix in front of the call, as,

```
        B.GETBYTE()->R.B    ...=CALL GETBYTE; LD B,A
or      W.GETWORD()->R.DE ...=CALL GETWORD; LD D,H; LD E,L
```

the register to be used (either R.A or R.HL only) is specified. In other words, for the "B." example, the prefix notifies Q that it is an A expression and upon returning, R.A is to be used in the data transfer. The "W." prefix has the same effect in signaling an expression with R.HL to be used.

6.0 BLOCK STRUCTURED (PASCAL-LIKE) EXPRESSIONS

Q supports a set of relational expressions grouped as "if", "while", "repeat", and "case" statements which take the following forms:

[Brackets indicate optional constructions.]

"Keywords are in quotes"

```
*****************************************************************************

"if" RELATION.BLOCK "then" INSTRUCTION ["else" INSTRUCTION] ["endif"]

*****************************************************************************
*****************************************************************************

            "while" RELATION.BLOCK "do" INSTRUCTION

*****************************************************************************
*****************************************************************************

        "repeat" INSTRUCTION.LIST "until" RELATION.BLOCK

*****************************************************************************
*****************************************************************************

"case"   INSTRUCTION "of"
         CP.OPERAND or PROCEDURE or BEGIN.END ":" INSTRUCTION
         CP.OPERAND or PROCEDURE or BEGIN.END ":" INSTRUCTION
         |
         etc.
         ["else" INSTRUCTION]
"end"

*****************************************************************************
```

where:

INSTRUCTION is any single Z-80 assembler or Q statement. In RELATION.BLOCK it is used to set the zero and carry flags. In "case" statements it is used to place a value in R.A which will be compared.

Examples:

```
LD A,Var   ...Z80 instruction
R.A->Var   ...Q instruction
```

INSTRUCTION.LIST is any number of instructions separated by ";"s or on different lines and may also include labels (at the beginning of a line) or comments. (Remember that comments must start with "...".)

Examples:

```
^Start; LD DE,End; LD BC,Count; LDIR ...instruction list
```

is same as:

```
^Start                    ...start of same list put on separate
LD DE,End                 ...lines
LD BC,Count
LDIR                      ...end of instruction list
```

BEGIN.END is the sequence "begin" INSTRUCTION.LIST "end". In the "case" statement, it is used the same as a PROCEDURE to set the zero flag. Note: The BEGIN.END construction is considered an INSTRUCTION. In other words wherever an INSTRUCTION is called for in the above relational expressions, an INSTRUCTION.LIST set off by "begin" and "end" may be used. Examples:

```
begin LD A,Var; R.A->@HL; INC HL end  ...= an INSTRUCTION
```

CP.OPERAND is any single operand which may be compared to R.A. That is, a numeric value such as 01F or 28, an 8 bit register such as R.A (also B,C,D,E,H, or L), or an "@" term (including @IX(n), or IY). In general, it is anything that works with:

```
R.A = CP.OPERAND
```

T.OPERAND defines a class of operands which are used in tests. It is only used to define RELATION and TEST, described below.

T.OPERAND depends on the preceding instruction. If the instruction is NOT an HL expression, then a T.OPERAND may be a CP.OPERAND, a "zero", or a "carry". If the instruction IS an HL expression, then it may be a 0, an R.BC, or an R.DE.

The default for T.OPERAND is "zero". Therefore, in expressions which call for T.OPERAND but don't have one, an operand of "zero" is assumed.

A "zero" causes a test of the zero flag (bit 6 of the flag register). A branch will then occur based on the result. This is useful when the zero flag contains the information for the branch but R.A does not. For example,

```
DEC B               ...decrements R.B and sets zero flag (among others)
if R.C zero then    ...loads R.A with R.C which does not set any flags;
INC A               ...therefore, the branch is based on the zero flag condition
else DEC A          ...set by the DEC B rather than the existing value of R.A.
```

A "0" or any numeric value, causes a compare of the value with R.A before any flags are tested and is used when the branch needs to be based on a comparison between a numeric value and the existing value of R.A.

The "carry" causes a test of the carry flag (bit 0 of flag register). The resulting branch will be based on the condition of this flag rather than the zero flag.

RELATION is an operator which specifies the type of test to be made on the test operand (T.OPERAND). The following is a list of the valid relation operators:

=	...equal to
<>	...NOT equal to
not	...same as <>
<	...less than
>	...greater than
<=	...less than or equal to

```
>=        ...greater than or equal to
[none]    ...default of no operator is the same as equal to
```

Note that most relational operators cause a single jump relative to be generated in a TEST. However, ">" and "<=" each cause two jump relatives. Therefore, consideration should be given to avoiding these when convenient. TEST is of the form:

```
INSTRUCTION RELATION T.OPERAND
```

When INSTRUCTION is omitted, it is assumed to be R.A. Recalling that an omitted RELATION and T.OPERAND assume = and "zero" respectively, observe the following expressions containing tests which are equivalent:

```
if then...
if zero then...
if = then...            [all these statements are equivalent]
if = zero then...
if R.A then...
if R.A zero then...
```

Note that two possibilities are left out. These two are special cases which are NOT allowed:

```
if R.A=zero then...    [these two are NOT VALID ]
if R.A= then...
```

Also note that if the = is changed to any of the other relational operators, such as < or even <=, the above two statements ARE VALID.

RELATION.BLOCK is a group of logical TEST's "and"ed and "or"ed together without parentheses, i.e.: TEST.1 and TEST.2 or TEST.3 . Higher priority is given to "and"s. For example,

```
if R.A>='0' and <='9' or >='A' and <='F' then RET;
```

first tests the character in R.A to see if it is greater than or equal to ASCII 0. If it is, then the test is said to be "true" and it then checks to see if it is also less than or equal to ASCII 9. If this test is also true, the RET will be executed. If either of these two tests fails, the test to the right of the "or" would be tried. If either of the two tests to the right fails, the statement after this if statement is executed next. And, if both tests to the right of the "or" are true, the RET is executed. Note that in the 2nd, 3rd, and 4th tests the R.A was implied.

PROCEDURE, as used in the "case" statement, refers to Q-type procedure call; that is, P.PROCNAME or PROCNAME(optional args.). When this is used in the case of a CP.OPERAND compared with R.A (see above), the zero flag is tested upon return from the procedure; if set, it is the same as if a match is found.

NOTE: End-of-lines (including possible comments) may go after INSTRUCTION,RELATION, "then", "else", "do", "until", or "of".

6.1 PROCEEDURE

A procedure in Q is essentially a subroutine and has the same significance. Any symbol name may be used as a subroutine/procedure call. The only difference between a symbol name used for a procedure call and one used for a subroutine call is that the former is defined in the symbol table as a PROC type and the other is not. Of the three methods of indicating a subroutine/procedure call, only one requires the name to be declared as a PROC type.

The first method of procedure call is to put an open and close paren after the procedure name:

```
                    Pname()
```

A variation of this allows that within the parenthesis, INSTRUCTIONS, separated by semicolons, may be added:

`Pname(^Start; R.B->R.D)` will generate a LD HL,Start; LD A,B; and LD D,A before the call to Pname.

A second method is to precede the procedure call with a "P.":

```
                    P.Pname
```

And lastly, is to merely state the name of the procedure:

`Pname` This last method requires that the procedure name be defined as type PROC. This is accomplished by putting "PROC" in the statement field immediately after the symbol name. For example,

```
     Pname:    PROC      ...this declares Pname to be a procedure.
```

Note that only this last method REQUIRES the label name to be declared a procedure. The "P." and "()" forms not only do not require this but can even be defined after reference to them has been made.

As a statement of proper software form, it is strongly recommended that the "()" form of procedure call be used. It is not only the most versatile but also the most readable.

Some examples:

```
     PROCEDURE()                    ..........   CALL PROCEDURE
     PPROC1(RH)                     ..........   LD A,H; CALL PROC1
     PROC2(BVARNAME->RB; 99->RA) ..........   LD A,VARNAME; LD B,A
                          LD A,99; CALL PROC2
     PROC3(WVARNAME)                ..........   LD HL,(VARNAME); CALL PROC3
```

Also, when procedures return values to be used in expressions, the type of value returned can be specified by the W. or B. prefix. For example,

```
     W.Pname()->R.BC      ...word returned in R.HL by Pname is placed in R.BC
```

```
or        B.Pname()->@DE         ...byte returned in R.A by Pname is placed in memory
                                 ...location pointed to by R.DE
```

A subroutine is defined as a section of code beginning with a label (so it can be referred to) and ending with a return (RET). Some subroutines have an implied return. That is, since the instruction prior to the RET is another subroutine call, the RET is dropped and the call is changed to a jump to the subroutine. Consequently, a superfluous RET is eliminated. For example,

```
Sub1:     R.B->R.C; Sub2(); RET         ...subroutine #1
Sub2:     R.HL->R.BC; RET               ...subroutine #2
```

Sub1 may be changed to:

```
Sub1:     R.B->R.C; JP Sub2             ...new subroutine #1
```

When Sub1 is called from wherever, the return address is pushed onto the stack. Sub1 then jumps to Sub2 with "JP Sub2" which then executes up to the RET. The RET pops the stack which contains the return address in the routine which called Sub1.

Q contains an optimizer which is trained to spot these occurrances, substitute the jump for the call and drop the RET. The source statement will still show the call and RET but the code in the [] brackets will show the optimized version.

6.2 if...then...else...endif

```
*******************************************************************************

  "if" RELATION.BLOCK "then" INSTRUCTION ["else" INSTRUCTION] ["endif"]

*******************************************************************************
```

If the RELATION.BLOCK test is true, the "then" INSTRUCTION is executed. If it is false, the "else" INSTRUCTION is executed. At the end of execution, the program continues at the statement following the "if" statement. If the RELATION.BLOCK tests false and there is no "else", the program also continues at the statement following the "if" statement.

The "endif" is not normally needed. Its main use is to make the end of an if statement clearer in the midst of several "begin...end" constructions.

EXAMPLES:

```
if R.A=0 then INC A                          ...insure that R.A is non zero
if R.A>='0' and <='9' or >='A' and <='F' then RET ...test for Hex digit
if begin LD A,Var; R.A->@HL; INC HL end =0 then INC A ...notice how the
                                                    ...begin-end equals one instruction.
if R.A=0D then begin                         ...notice use of multiple lines to improve
          LD C,0                                    ...readability and
          Put1(R.A)                          ...allow comments
          0A                                        ...on each line.
```

```
                end
        else begin
                DEC C
                if R.A<>8 then begin            ...backspace?
                        INC C; INC C                    ...no
                  if R.A<' ' or R.A=07F then '#' ...non-character
                end
                endif                           ...not really needed but makes the end of the if
                                                ...statement clearer.  Also gives an additional check
                                                ...on the begin-end pairs matching.

                end
        endif                                   ...ditto reason for previous endif.
Get1if: PROC            ...get 1 char. from console->R.A if ready; ret. Z=0 iff got one
        if begin IN A,(CONRDY); BIT INRDY,A end <> then begin ...char waiting?
                IN A,(CONIO)                    ...yes, get the char.
                RES 7,A                         ...reset bit 7 to mask off parity
                end
        RET
```

6.3 begin...end

In the definitions of the block structures at the beginning of this chapter, wherever there is INSTRUCTION indicated, a multiple instruction INSTRUCTION.LIST may be substituted if it is preceded by "begin" and followed by "end". Example,

```
begin LD A,Var; R.A->@HL; INC HL end  ...= an INSTRUCTION
```

Errors caused by a "begin" without a matching "end" are fairly common and sometimes difficult to locate. So, particular attention should be given to using proper form with them. Notice the examples in section 5.2. The "end" should be in vertical line with the preceding code. For more details on proper structuring of begin...end expressions, consult chapter 15, Software Notes.

There are some common indications which help to locate errors caused by lack of a matching "end". Understanding the following description will provide some insight into locating this type of error.

If a "/MAP 8" is placed at the end of the source file to catch undefined labels (EXTEND & USERCC are OK as undefined labels), it or some other illegal statement will be flagged as an error. If some other error is in the source file, the /MAP 8 will work and show an undefined label with a symbol name that is a two digit number followed by a #, such as 00#. The address associated with this undefined label will give a clue of where to start looking. It will be at a "begin" but not necessarily the one without the matching "end". If it isn't, look for another nested inside.

There is a limit of 255 bytes coming between the "begin" and "end". It is not necessary to be concerned with the exact length when coding because an error message during assembly signals this condition. It is easily handled by creating a subroutine from some of the code and replacing it with a subroutine call within the begin-end bracket.

6.4 while...do

```
******************************************************************************

                 "while" RELATION.BLOCK "do" INSTRUCTION

******************************************************************************
```

There are two basic types of loops in Q. The while-do loop which makes a relational test before executing the conditional instruction and the repeat- until loop which executes the instruction(s) first and then exits based the result of a relational test at the end. The repeat-until loop is covered in the next section.

Note that after the "do" is INSTRUCTION. Since an INSTRUCTION.LIST is more likely to be needed, a "begin" INSTRUCTION.LIST "end" construction which is equivalent to an INSTRUCTION will often be needed. For example,

```
while R.A>0 do begin
          LD (HL),B
          INC HL
          DEC A
          end  ...from "begin" to here = an INSTRUCTION
```

6.5 repeat...until

```
******************************************************************************

               "repeat" INSTRUCTION.LIST "until" RELATION.BLOCK

******************************************************************************
```

In this type of loop the INSTRUCTION.LIST is executed first. Therefore, the loop code will always be executed at least once. This is in contrast to the "while" loop which may not execute any of its code should the RELATION.BLOCK test false the first time through.

Special code utilizing the Z80 DJNZ instruction is generated for "repeat" loops ending with "until DEC B zero;" ("zero" is optional). (The ";" must be present; if left off, the normal DEC B; JR NZ,$-n code will be generated.) Be aware that the zero flag is not automatically set in the DJNZ form of the loop. Example,

```
LD B,(MaxCount); LD HL,(Pointer1); LD DE,(Pointer2)
repeat
          @HL                          ...get char
          if R.A=0D then LD B,1        ...quick exit if carriage return
          R.A->@DE                     ...transfer to line buffer
          INC HL; INC DE
until DEC B zero;
```

Some further examples:

```
Get1:   PROC                    ...get 1 char. from console->R.A (wait if one not ready)
        repeat Get1if() until not zero
        RET
Echo:                           ...echo char. & keep track of col. in R.C
        if R.A=9 then begin
                repeat Put1(' '); INC C until R.C&7=0
                RET
                end
        if R.A=0D then begin LD C,0; Put1(R.A); 0A end
        else begin
                DEC C
                if R.A<>8 then begin
                        INC C; INC C
                        if R.A<' ' or R.A=07F then '#'
                        end
                end
Put1:   PROC                    ...put out 1 char. (in R.A) to console
        PUSH AF
        repeat IN A,(CONRDY) until BIT OUTRDY,A not zero  ...check console rdy.
        POP AF; OUT (CONIO),A; RET      ...print char.
```

6.6 case...of...else...end

```
*******************************************************************************

"case" INSTRUCTION "of"
        CP.OPERAND or PROCEDURE or BEGIN.END ":" INSTRUCTION
        CP.OPERAND or PROCEDURE or BEGIN.END ":" INSTRUCTION
        |
        etc.
        ["else" INSTRUCTION]
        "end"

*******************************************************************************
```

First the INSTRUCTION between the words "case" and "of" is executed. This usually loads R.A with a value which is to be compared. Following the "of" is a list of possibilities. In its simplest form when CP.OPERAND is used, R.A is compared to the operand. If the comparison is true, that case is selected and the following INSTRUCTION is executed; otherwise, it is skipped and the next case is checked. After execution of the selected INSTRUCTION, program flow continues with the statement following the "end". For instance,

```
case GetChar() of               ...GetChar returns a char from console in R.A
        'Y':    EraseFile()
        end
^Buffer                         ...program continues here
```

If the character from the console is Y, EraseFile is executed after which program flow continues at ~Buffer. If it is any other character EraseFile is not executed and program continues at ~Buffer. This example is obviously oversimplified. If a few lines are added,

```
case GetChar() of              ...GetChar returns a char from console in R.A
          'Y':      EraseFile()
          'N':      begin ^FilePointer; NextFile() end ...skips to next
          'A':      repeat EraseFile() until not zero ...erase all
          else      DisplayErrorMsg()
          end
^Buffer                        ...program continues here
```

Then one of three choices is given plus a default (else). If R.A contains either a Y, N, or A, the INSTRUCTION to the right of the colon (:) is executed after which flow continues at ~Buffer. Otherwise (else), DisplayErrorMsg is executed; then, flow continues at ~Buffer. Although the code generated by the case statement is less efficient than using straight assembler techniques, it can be written and modified with relative ease and is easy to understand at a glance.

When using the "begin...end" or PROCEDURE options, upon completion of the code between the begin...end or alternately upon return from the PROCEDURE call, the zero flag (bit 6 of register F) is tested. If true (=1, =zero), that line is selected and the corresponding INSTRUCTION is executed after which program flow continues skips to the end of the case statement.

Note that the INSTRUCTION on the compare line MUST begin on that line.

HOW THE CASE STATEMENT WORKS:

The following listing (modified with an editor to put in explanations) examines the mechanism of a simple case statement. All case statements, regardless of complexity, have the same structure. Only the number of choices is increased.

```
5000    Subroutine: EQU 05000        ...a hypothetical subroutine call
5100    ChoiceA: EQU 05100  ...a hypothetical INSTRUCTION call
5200    ChoiceB: EQU 05200  ...ditto

0100     case Subroutine() of                 [CD0050  ...call to Subroutine
                                               E5      ...push HL because...
                                               211014  ...load HL with adr of code
                                                       ...following the end of case
                                                       ...statemtent (Next:), then
                                               E3]     ...exchange stack pointer with
                                                       ...HL; this puts the adr of
                                                       ..."Next:" on the stack so RET
                                                       ...will cause exectution to
                                                       ...continue after end of case.
0108               'Y':      ChoiceA() [FE59  ...compare with 'Y'
                                               2004    ...No?, jump to next choice.
                                               CD0051 ...Yes, call ChoiceA, then
                                               C9]     ...returns to Next
0110               else      ChoiceB() [CD0052]        ...default-call to ChoiceB
```

```
0113          end                          [C9]     ...must clean up stack
0114     Next:               ...this is the next statement to be executed after the case
                             ...statement has made a selection.
```

The case statement uses the stack to store an address while it is executing. Thus, if jumping out of the case statement (without returning), the stack should be popped. Observe:

```
case GetChar() of
          CarriageReturn : begin POP AF; JP Continue end ...Exit
          |
          etc.
          end
```

The various cases may alternatively be separated by ";"s instead of end-of-lines.

7.0 RE-ASSEMBLY FOR ACCURATE OBJECT CODE LISTING

The hexadecimal code (between the [] brackets) in the listing generated by Q does not always correspond to that which ultimately resides in memory. Specifically, all printed code associated with forward references is inaccurate because Q is a one-pass assembler which lists the values deposited in memory before the forward references are resolved.

It is possible to make a second assembly run, once all forward references have been resolved on the first pass, to obtain a more accurate listing. This is done by ORGing to the original starting address and re-assembling the same file once more (with the symbol table now starting fully populated). To prevent all of the REDEFINED LABEL error messages which would normally occur on the second pass, Q provides a special BYTE variable call REDEFSW, whose normal value of Ø prevents redefinition of labels. A non-zero value permits redefinition. Thus, the second pass should begin with the immediate command:

```
     /1->REDEFSW   (executed in bank 1).
```

The only inaccuracies which will remain in the listing will be due to the fact that, in the translation of certain control expressions, Q generates and then ZAPs some temporary labels. These references will still appear as zeroes.

8.0 CONDITIONAL ASSEMBLY

An assembler having conditional assembly features is able to ignore (skip past) certain subsets of statements in a source file when corresponding user-specified parameters are set properly. This permits different versions of the same program to be incorporated into a single source file. Q is capable of a form of conditional assembly via use of its global symbol Skip. ("Skip" is defined as a BYTE variable. Each memory bank has its own "Skip" byte.) The value of this variable affects whether or not Q skips any statements in the source file and, if so, which. The following rules apply:

Skip value	Q's response
Ø	Normal assembly. No statements skipped.
1	Any statement beginning with * will be skipped (not compiled).
2	Only statements beginning with / will be taken. All others will be skipped.

Notice especially that an immediate statement such as

```
*DO FILE2
```

can materially change a program, if skipped. It is assumed that a source program taking advantage of this conditional assembly feature will include one or more immediate commands (possibly initially entered from the keyboard) to set the Skip value (which starts at Ø). For example,

```
/2->Skip   or   /if CONDITION then 1->Skip
```

may be used. Normal assembly may be restored via /Ø->Skip, for example.

It is also possible to make a numeric value within a statement conditional on certain immediate parameter values. Whenever a two-byte numeric value may be used, you may also use the form:

```
value(INSTRUCTION)
```

where INSTRUCTION, as defined earlier, is one or more statements separated by semicolons. Whenever Q encounters this form, it immediately executes the statements inside the parentheses and uses the resulting value left in R.HL as the two-byte number. As an example,

```
DEFW value (P.GETVAL)*3
```

assembles the two-byte number which is three times the value in R.HL returned by immediate execution of the subroutine GETVAL.

9.0 ASSEMBLY WITH AN OFFSET

Q includes the capability to assemble code at one location for execution (after relocation) at another. Q maintains a special global variable (with attribute BYTE) whose name is OFFSET and whose value is the signed difference, in number of pages (where one page is 256 or Ø1ØØ bytes), between the location of the assembly area and the location of the ultimate execution area. The default value for OFFSET is, of course, zero. For example,

```
/-Ø44->OFFSET
```

at the beginning of a run, will assemble code which should ultimately be loaded at location zero before execution begins.

Q assigns the offset value (corresponding to the location at execution time) to all symbols (including the special "$" symbol) except those defined by EQU or DEFL. An EQU or DEFL symbol may be offset, if desired, by equating it to an offset label. Since the ORG statement is meant to control the value of the location counter, which is an assembly time parameter, if the first item following "ORG" is a symbol (or "$"), the offset value is subtracted from it, thus cancelling the offset. (The values of any items after the first are taken as is.)

One warning: During assembly with offset, no error checking for out-of-range JR instructions is done.

Note: Pages 47 + 48 have been removed.

11.0 SYMBOL TABLE

The symbol information is stored in a 'symbol table' in memory. (This table initially contains global symbols from Q which may be needed for extensions, debugging, etc.) The symbol table can be altered with the following commands:

/ZAP <list of symbol names separated by a space>

effectively deletes (zaps) the specified symbols from the table (although they still take up space in memory).

/ZAPALL [mask]

zaps all symbols not masked out. Use the bits as described in section 4.7 (/MAP) to compose the [mask] byte. The bits are active true.

The "PACK" series of commands usually follow the ZAP command and are used to pack the symbol table into minimum space by saving only undefined symbols, symbols masked out, and, except for /PACK ALL, global symbols.

/PACK

gets rid of all symbols which are not global or undefined. In other words, it eliminates all defined local symbols.

/PACK ALL

will pack even the global symbols.

/PACK [mask]

packs the symbol table saving only undefined symbols and global symbols which have not been zapped or masked out.

/PACK ALL [mask]

packs the symbol table saving all symbols not zapped or masked out.

****** Note: undefined symbols cannot be zapped or eliminated through packing.

/CLEAR

clears the entire symbol table.

MOVING THE SYMBOL TABLE

It is possible to move the symbol table elsewhere in memory or change the amount of space allocated for it. The word at TABBSE is the starting address of the symbol table and at TABBSE+2 is the address of the location after the end of the table.

The first word in the table is its current size.

Simply move the table as desired and patch into TABBSE and TABBSE+2 its new position.

Example:

Suppose the symbol table starts at 0A000 and it is desired to move it to start at 09000. The following code will accomplish this:

```
*@2TABBSE->R.BC; TABBSE->R.HL; LD DE,09000; LD (TABBSE),DE; LDIR
```

The symbol table may now be re-imaged (along with the rest of Q if desired) between the addresses TABBSE and @2(TABBSE->R.BC)+R.BC-1.

12.0 DEBUGGING WITH QBUG

QBUG is the name given to the debugging facility provided within Q. It is entered while in Q by typing "/*" (can also be in a file; any open input or output files are closed). The prompt is a "*". (To return type "Q".)

From here, any instruction (or instruction list) that is typed will be executed immediately, with registers preserved from one line to the next (including any changes made). All of the commands (beginning with a "/") are also valid in the debugger.

Upon entering Q, breaks (opcode "FF") are set to go to the Q-debugger rather than the RIO prom debugger. To change back to the prom debugger, use the command

```
/BRKS->P
/BRKS->Q ...sends them to Q again.
```

On leaving Q, breaks are again sent to the PROM.

In addition to the regular Q instructions, there are some debug instructions which may only be executed from the debugger:

D address,n

Displays n lines of memory, 16 bytes per line, starting at "address". ",n" is optional and defaults to n=1. If "D" alone is typed, one line starting at the value of the stored program counter (R_PC) will be displayed. "address" may be either a hex number or an HL-expression, e.g. D ^ABC+012. Also note that R.HL is initialized to the starting address of the previous line dumped before the HL-expression is evaluated; thus, for example, a linked list (where the link address is the 1st word of the previous dump) may be followed by doing D @2HL. A special case of this command is when either n=0 or there is a comma with nothing following. One line is dumped and the cursor positioned under the 1st byte. The cursor may be advanced by typing a space or tab (to advance to the next word) or backed up using the backspace. Memory can be changed a nibble at a time by typing the desired hex digit when the cursor is in position. Hitting return or line feed completes the operation and displays the changed line. Line feed additionally causes the operation to be repeated on the next line.

B address,n

Sets a breakpoint with a repeat count of n at "address". "address" may be either a hex number or an HL-expression, as described above for "D". There may be a maximum of 8 different breakpoints at one time. A repeat count of n means the break will not occur until the nth time the breakpoint is hit. (After the break occurs, the repeat count is left at 1.) ",n" may be left off, with n defaulting to 1. In addition, just typing "B" will cause a listing of all the currently active breakpoints.

X address

Removes breakpoint at "address" where "address" takes the same form as above. "X" by itself causes all breakpoints to be removed.

N n

Execute next n instructions (after a breakpoint has been hit or a previous step) displaying registers that change. If n is not given, it defaults to 1.

S n

Same as "N" above, but skips over procedure calls. Note: CALL instructions at the PC are disassembled (including the procedure name) after breaks and steps. (If DEASM is present, all instructions are disassembled.)

G

Resume execution after a break--or any time there are valid values for the saved PC, SP, etc. (as shown with the "R" command)--with all registers restored. Note: if R_PC is at a set breakpoint, a "Next" will be executed first in order to restore the breakpoint; otherwise will just "go".

J address

Sets the stored program counter (R_PC) equal to "address", where "address" takes the same form as above with "D", then begins execution, initializing the registers with the values as shown with the "R" command. Note that regular jump or call instructions may also be used to start execution (without initializing the registers).

R

Displays the values of the registers that have been saved following a break or after an "N", "S", or "T" command

R,

Displays the values of the saved registers and puts the cursor underneath. The cursor may then be moved along to under any nibble in any register and the value there changed in the same manner as used to change memory (see the "D" command above). Hitting return completes the operation.

R'

Displays the values of the shadow registers (refer to ZILOG Z80 CPU manual for explanation of these).

H n

History: displays the last n addresses or values on the stack (including corresponding symbol names if any) following a break (or step, etc.). If n is not given, it defaults to 1.

T n

Starting after a break, traces program execution displaying the names of all subroutines entered indented to subroutine depth (as measured by the stack pointer), up to a depth of n (or infinitely deep, if n is not given). Continues until depth is less than original depth.

L address,hh

Locate a string in memory; hh is a pair of hex digits representing a byte. This may be extended for any number of bytes by adding, without intervening spaces, extra pairs of hex digits. The search starts at "address", which takes the same form as above with "D". (If "address" is left off, the search starts after the starting address of the most recent line dumped.)

U address,n

(valid only if DEASM is present; indicated by "Qx.D") Unassemble (dissassemble) n instructions starting at "address". "address" takes the same form as above with "D".

Miscellaneous:

1. During operation of the "S" or "T" commands, when 'skipping' a subroutine (in the "T" command, this happens when going below the maximum level to be traced),

QBUG checks to see if it can set a temporary break at the return address and 'go' rather than stepping all the way through. This will be done if after 4 instructions of the subroutine there have not been any pops or exchanges with the stack (which might indicate an altered return address). If the return address is otherwise changed, oops

2. During execution of a multiple step command, e.g. N 255 or S 255, or the "T" command, 'escape' can be pressed to cleanly abort the operation at that point.

3. The debugger uses disassembler to stepping through programs (used by the "N", "S", and "T".

4. If a user's program uses some routines in Q also used by QBUG (in par ticular those associated with the 'write' statement), a potential conflict exists. This would generally arise when stepping through that part of the program (including when using the "T" command).

5. If necessary, QBUG's stack pointer can be reset by doing "Q", then "/*".

6. Normally when an instruction list is typed following the "*" (for immediate execution), the code goes at the next code location. However, if a "/" is typed first, it will be put at an out-of-the-way location instead. This might be necessary if there was code at the next code location still to be executed, as in the following situation:

```
*B ^ABC
*CALL ABC  (this actually generates CALL ABC; JP Debug)
```

After the break at ~ABC, there still remains the code "JP Debug" to be executed when ABC eventually returns. Thus, to execute something else at this point, one should do:

```
*/TEST(2)  (or whatever)
```

7. The following procedures in QBUG may be called from a user's program (referenced via the global symbol Dbug):

Regs: EQU Dbug+6	writes out the current value of the registers (preserves registers)
Dump: EQU Dbug+3	displays memory starting at R.HL with R.A=number of 16-byte lines to display (if R.A=0 then does 1 line) (preserves registers other than HL and AF)

Also, the global symbol R_AF points to the registers stored in Q after a break (or after the "N", "S", or "T" commands). These are stored in the following order: R_AF, R_BC, R_DE, R_HL, R_IX, R_IY, R_SP, R_PC.

Also note: to add extensions to the debug commands, the place to patch in is at ~Dbug. This would be called immediately before the debug commands are parsed.

13.0 PATCHING

To do patching of code, a "patch" statement of the following form is provided:

```
patch address,instlist
```

This inserts at "address" a JP to the current code area; there the given in- struction list is inserted, followed by the 3 bytes of instructions displaced by the JP, followed by a JP back (to "address"+3). (A JP back to "address"+n, where n is a digit in the range 4-9, can be generated instead by replacing "address" by "address[n]".) "address" takes the form of either a hex number or an HL-expression.

```
Example: patch ^ABC+012, R.A->@HL; INC HL
```

An alternate form of the 'patch' statement is:

```
patchr address,instlist
```

This is the same except that the 3 (or n) bytes of instructions displaced by the JP to the patch area are not copied into the patch area.

Note that the 'patch' statement must be entered during assembly (as opposed to debug) mode.

14.0 SUBROUTINE LIBRARY AVAILABLE TO USER

These subroutines are contained in Q and are accessible to the user by making referrence to the subroutine name. Many other symbol names are listed in Q's symbol table but the following are the most useful.

Input and Input Checking:

Test:

Checks whether a sequence of bytes pointed to by R.DE matches a literal (in DEFT form at the return address). If all bytes match: Z flag =1; R.DE incremented past byte sequence; original R.DE->LASTDE
If no match: Z=0; R.DE restored
Calling seq: CALL Test; DEFT 'STRING'

Id:

Upper case letter followed by a sequence of upper or lower case letters or digits. Z=1 iff match; original R.DE->LASTDE; R.DE updated to next non-matching character.

Digit:

Tests whether R.DE points to an ASCII digit. R.BC, R.HL unchanged. If match: Z=1, @DE->R.A-; R.DE->LASTDE; R.DE+1->R.DE No match: Z=0, R.DE unchanged.

Hexd:

Tests whether R.DE points to one digit or one of the upper case letters A through F. Input and output same as for digit.

Num:

Converts a sequence of decimal digits pointed to by R.DE to their binary equivalent, storing that in R.HL, and updating R.DE. Input R.A contains assumed high order digit. Normal usage is:

```
CALL Digit
         JR NZ,NOTDIG
         CALL Num ... SET R.HL to value
         LD (NVAL),HL ... save it
      NOTDIG:

      or:  if P.Digit then begin P.Num; R.HL->NVAL end
```

HNum:

Converts a series of hex digits pointed to by R.DE to their binary equivalent, storing that in R.HL, and updating R.DE. On entering, R.DE is assumed to be pointing to first hex digit; if there are none, a value of 0 will be returned.

Sr:

Gets a string from at R.DE, advancing R.DE until a " is found (always takes at least one character). Original R.DE->LASTDE, Z=1.

Getcon:

When called, R.A should contain desired prompt character. This will be put out; then, one line of input will be accepted from the console and stored in an internal buffer. Returns with starting address of the input buffer stored in R.DE and LASTCR.

Get1:
Gets 1 character directly from keyboard and returns it in R.A (with bit 7 reset). Saves registers except AF.

Output and Output Assembly:

Outset:
Stores R.DE in NEXTDE and sets R.DE to the first byte of an internal buffer. Other registers unchanged.

WSbgn:
Pushes all registers used by Q onto stack, stores DE in LASTDE and sets R.DE to the first byte of an internal buffer (other registers unchanged). Meant to be used with WSend'.

Copy:
Copies a literal into an area pointed to by R.DE. R.DE updated past copied literal. Other registers (except flag) unchanged. Calling seq: CALL Copy; DEFT 'STRING'

Copyin:
Copies data into an output area pointed to by R.DE, updating R.DE. The data is addressed by the pointers LASTDE which contains the address of the first byte to be copied, and NEXTDE which addresses the byte after the last one to be copied.

Outhex:
Converts R.A. to the equivalent two characters hexadecimal ascii representation, storing the two characters at R.DE and R.DE+1. R.DE+2->R.DE. Other registers (except flag) unchanged.

HLout:
Puts out @DE R.H in hex followed by R.L in hex. R.DE+2->R.DE. Other registers unchanged.

Outp:
Will output the characters in the internal buffer up to R.DE (adding a carriage return) to list file if open, otherwise to the console. ("Up to R.DE" means not including R.DE.) NEXTDE->R.DE, Z flag=1.

Outmsg:
If a list file is open, will put out characters in the internal buffer up to R.DE (adding a CR) to both the file and the console, otherwise just to the console. (Note: calling Outmsg+4 skips adding the CR.)

Putcon:
Outputs characters in the internal buffer up to R.DE just to the con sole (without adding a CR).

WSend:
Calls Outmsg', then pops all the registers from stack. Meant to be used with WSbgn'.

Put1:
Puts out the character in R.A to directly to console. Saves registers.

Debugging:

Regs: Writes out the current value of the registers (preserving registers). Call via Dbug+6.

Dump: Displays memory starting at R.HL with R.A=number of 16-byte lines to display (if R.A=0 then does 1 line); preserves registers other than HL and AF. Call via Dbug+3.

Miscellaneous:

Lkup: Looks up symbol pointed to by R.HL and followed by a space in Q's symbol table. If not found, it is added in. Returns R.HL and R.IY pointing to the symbol's stored value (or chain link). Z=1 if found, Carry=1 if referenced but not defined. (R.DE preserved.)

Lkupnl: Same as Lkup but does not add symbol to table if not found.

Name: Enter with an address in R.HL (call $); returns R.HL=pointer to symbol (in symbol table) of greatest value <=$ (call ^LOC), R.BC=$-(LOC); preserves R.DE; Z=1 if found. Call via Dbug+9.

Err: Calls Q's error routine (R.DE should point to the error in the source code) and then restarts starting at the next line.

Errm: Same as Err except first prints out an error message which follows the call in DEFT form. (Also, it is not necessary for R.DE to be pointing to source.)

Eff1: Deletes instructions of the form 2000H (JR NZ,$+2) from the code area (and decrements by 2 the variable NCODE, which points to the current position in the code area).

RDaHL: Do LD A,(HL) from whichever memory bank is set in Q.

WRaHL: Do LD (HL),A from whichever memory bank is set in Q.

Outopn: Opens an output (list) file. Enter with R.DE pointing to 'FILENAME' (in text) followed by a delimiter. Call via Dbug+0F.

Quit: Closes all open input or output files. To just close the output (list) file, call Quit+3.

15.0 SOFTWARE NOTES

Notes on coding techniques using Q with Z80 architecture.

15.1 WRITING FOR READABILITY/MAINTAINABILITY:

The programmer should enforce a system of nesting when dealing with Q's block structures:

```
if - then
          begin
          ------
          ------
          ------
          end
else ------;
```

Following an "if", indent all statements that are conditionally executed (including begins and ends) i.e. DON'T use

```
if----then begin                ...INCORRECT
R.HL -> R.DE end
```

Line up "end" with preceeding statements that are executed. That is,

```
while ---- do
          begin
          -----
          -----
          end
```

Generally, lining up statements indicates they will be executed in the natural, sequential order. Also, the "lines" can be left to right for brevity as in

```
repeat INC DE; DEC L until @DE='1'
```

15.2 IN PLACE BYTE REGISTER TESTING FOR ZERO:

When R.A must be saved, then

```
if begin INC L; DEC L end zero then ...
```

is an effective means of testing an 8 bit register.

15.3 WORD TESTING FOR ZERO:

Q generates optimal code for cases like

```
if R.HL=O then ...
```

generating: LD A,H; OR L; JR NZ,- (NOTE: if R.HL<>0 also works) but since word expressions always leave the value in R.HL, forms like the above with R.DE, i.e.

```
repeat ...
until R.DE=O ...
```

unneccessarily copy R.DE into R.HL. Preferable would be the use of the direct from as in

```
until R.D|R.E zero ...
while R.B|R.C not zero do ...
```

which is reasonably self-documenting plus more efficient.

15.4 COUNTING TESTS FOR BYTE QUANTITIES:

The Z80 makes special use of R.B in the DJNZ instruction. Q will generate this instruction for the case

```
...-> R.B
repeat
...
until DEC B zero;
```

Omission of the trailing semicolon may cause the generation of two intructions (DEC B; JR NZ, S-?) at the end of the loop rather than the DJNZ instruction.

NOTE: The Z-flag is not set in the DJNZ form of the the loop.

15.5 COUNTING TESTS FOR WORD QUANTITIES:

If efficient code is needed for counting higher than 255, the following pattern could prove useful:

```
LD DE,O-number ...place negative of count in register
repeat
          ... code ...
until INC E zero and INC D zero;
```

this generates optimal code and is superior to the equivalent:

```
LD DE,O-number
repeat
          repeat
                    ... code
```

```
                        until INC E zero
                until INC D zero
```

15.6 WORD EQUALITY COMPARISONS:

Frequently it is desired to compare a word variable to a constant or some other register value without modifying the variable in R.HL. Since the ADD HL,DE or ADD HL,BC instuctions do not affect the Z flag, the following cases work:

```
        LD DE, ENDVAL
        if R.HL-R.DE+R.DE zero then ... HL equals ENDVAL
```

However Q now generates this code directly when

```
        if R.HL=R.DE then ...
or      if R.HL=R.BC then ...
```

is specified. Note that >, <, >=, <=, and <> cases are also supported.

15.7 INCREMENTING/DECREMENTING BYTES:

Users should be aware that

```
                VAR:  BYTE N
                ...
                VAR+1-> VAR ...
```

generates 7 bytes of code (LD A, (VAR);INC A; LD (VAR),A) while modifying R.A, of course. Often an alternative form is preferable, namely:

```
                ^VAR; INC (HL)
```

which generates 4 bytes of code, and is faster. R.HL is modified in this case, but R.A is not.

15.8 AVOIDING USE OF R.IX:

These registers provide a more convenient means of addressing elements of complicated structures than use of R.HL, but their use consumes memory space at a relatively high rate. Remembering that structures can have negative as well as positive adresses, some data can be kept just prior to the structure pointer (i.e. a reverse link, etc.). Consider:

```
        PREV:    WORD 0-0 ... pointer to previous item
        RECORD:  ... base of record
        NEXT:    WORD 0-0 ... pointer to next
        DATA:    DEFS SIZE
```

```
PREVD:   EQU RECORD-PREV ... displacements
NEXTD:   EQU RECORD-NEXT ...
```

with R.IX used as a pointer with the value RECORD, the "next" function is:

```
GONEXT:  LD L,(IX+NEXTD)
         LD H,(IX+NEXTD+1)
         PUSH HL
         POP IX .../ 9 bytes

GOPREV:  LD L,(IX+PREVD)
         LD H,(IX+PREVD+1)
         PUSH HL, POP IX .../9 BYTES
```

whereas with R.HL we have:

```
GONEXT:  @HL-> R.A;INC HL
         LD H,(HL)
         R.A-> R.L      /4 bytes     (This is identical to @2HL

GOPREV:  @(HL-1)-> R.A ;DEC HL
         LD H,(HL)
         R.A-> R.L      /5 BYTES
```

15.9 CAUTIONS:

Certain pleasing constructions expressible in Q generate excessive code. Some examples are:

```
EntryWidth:              EQU  6
R.DE - EntryWidth -> R.DE
```

which generates:

```
LD H,D; LD L,E        626B
LD A,6              BE06
CPL A               2F
ADD A,L             85
LD L,A              8F
JR C,$+3            3801
DEC H               25
INC HL              23
LD D,H; LD E,L      545D / 13 bytes
```

Preferable would be:

```
LD HL,0-EntryWidth        21FAFF
R.HL+R.DE -> R.DE         19345D / 6 bytes
```

If the result is needed in R.HL the relative improvement is greater, 11 bytes to 4 bytes.

Also,

```
W. EntryWidth -> R.BC
^Entry -> R.DE
^BUFFER -> R.HL
LDIR ... copy input
```

looks quite readable, but remember that Q has a policy of leaving the values of HL-Expressions in R.HL. Consequently, the first two lines generate 5 bytes each instead of the 3 generated by

```
LD BC,EntryWidth
LD DE,Entry
```