QBASIC

USER GUIDE

by John Walker

Marinchip Systems

Mill Valley, CA 94941

1. Introduction 1
1.1. Notation 1
1.2. Acknowledgements 2

2. Using QBASIC 3
2.1. Disc Executive instructions 3
2.2. Network Operating System instructions 5
2.2.1. Converting programs from Disc Executive 7
2.2.2. Using hard disc 7

3. Language syntax 9
3.1. Statements 9
3.1.1. Line numbers 9
3.1.2. Continuations 10
3.1.3. Multiple statements per line 10
3.1.4. Significance of spaces 10
3.2. Comments 10
3.2.1. REMARK Statement 10
3.2.2. Embedded comments 11
3.3. Constants 12
3.3.1. Integer constants 12
3.3.2. Real constants 12
3.3.3. String constants 13
3.4. Variable names 13
3.4.1. Subscripted variables 14
3.5. Expressions 15
3.5.1. Operators 15
3.5.1.1. Arithmetic operators 15
3.5.1.2. Relational operators 16
3.5.1.3. Logical operators 17
3.5.2. Order of evaluation 17

4. Assignment and control statements 19
4.1. LET (assignment) statement 19
4.2. GOTO statement 19
4.3. GOSUB statement 19
4.4. RETURN statement 20
4.5. IF statement 20
4.5.1. Single-line IF 20
4.5.2. Block IF statement 21
4.5.3. ENDIF statement 22
4.5.4. ELSEIF statement 22
4.5.5. ELSE statement 22
4.6. WHILE statement 23
4.7. WEND statement 24
4.8. FOR statement 24
4.9. NEXT statement 25
4.10. EXIT IF statement 25
4.11. ON statement 26
4.12. STOP statement 26
4.13. RANDOMIZE statement 27
4.14. DIM statement 27
4.15. CHAIN statement 28
4.16. COMMON statement 29

5. Predefined functions 31

5.1. Numeric valued functions 31
5.1.1. ABS(X) - Absolute value 31
5.1.2. ACOS(X) - Arccosine 31
5.1.3. ADRS(X) - Address of variable 32
5.1.4. ASC(A$) - ASCII code 32
5.1.5. ASIN(X) - Arcsine 32
5.1.6. ATN(X) or ATAN(X) - Arctangent 32
5.1.7. COS(X) - Cosine 32
5.1.8. COT(X) - Cotangent 32
5.1.9. CSC(X) - Cosecant 32
5.1.10. DATETIME(I%) - Date and time 32
5.1.11. EXP(X) - Exponential 33
5.1.12. FLOAT(I%) - Convert to Real 33
5.1.13. FRE - Total free space available 33
5.1.14. INT(X) - Integer part of Real 34
5.1.15. INT%(X) - Convert to Integer 34
5.1.16. LEN(A$) - Length of string 34
5.1.17. LOG(X) - Natural logarithm 34
5.1.18. MATCH(A$,B$,I%) - Search for pattern in string 34
5.1.19. MFRE - Largest memory block available 35
5.1.20. RND - Pseudorandom number 35
5.1.21. SADD(A$) - String address 35
5.1.22. SEC(X) - Secant 35
5.1.23. SGN(X) - Sign 36
5.1.24. SIN(X) - Sine 36
5.1.25. SQR(X) - Square root 36
5.1.26. TAN(X) - Tangent 36
5.1.27. VAL(A$) - Value 36
5.2. String valued functions 36
5.2.1. CHR$(I%) - Character from ASCII code 36
5.2.2. COMMAND$ - Command string 36
5.2.3. LEFT$(A$,I%) - Left part of string 37
5.2.4. MID$(A$,I%,J%) - Extract substring 37
5.2.5. OVERLAY$(A$,B$,I%) - Overlay string 38
5.2.6. RIGHT$(A$,I%) - Right part of string 38
5.2.7. STR$(X) - String representation of number 38
5.2.8. UCASE$(A$) - Upper case 38

6. User defined functions 39
6.1. Single line functions 39
6.2. Multiple line functions 40
6.3. Function calls 42
6.4. Call by reference 42

7. Input and Output Statements 44
7.1. Console and printer input/output 44
7.1.1. PRINT statement 44
7.1.2. PRINT USING statement - formatted output 45
7.1.2.1. ! - Single character string field 45
7.1.2.2. /.../ - Fixed length string field 45
7.1.2.3. & - Variable length string field 46
7.1.2.4. Numeric fields 46
7.1.2.5. Forcing field control characters 48
7.1.2.6. Matching of expressions and fields 48
7.1.3. CONSOLE statement 49
7.1.4. LPRINTER statement 49
7.1.5. Console/printer output functions 49

7.1.5.1. POS function 49
7.1.5.2. TAB function 50
7.1.6. INPUT statement 50
7.2. Direct console input functions 51
7.2.1. CONSTAT% function 51
7.2.2. CONCHAR% function 52
7.3. Program data input statements 52
7.3.1. DATA statement 52
7.3.2. READ statement 53
7.3.3. RESTORE statement 54
7.4. Array input/output 54

8. File input/output 56
8.1. OPEN statement 56
8.2. CREATE statement 58
8.3. FILE and GETFILE statements 58
8.4. READ statement 60
8.4.1. Sequential file variable READ 60
8.4.2. Sequential file line READ 60
8.4.3. Random file variable READ 61
8.4.4. Random file line READ 61
8.5. IF END statement 61
8.6. PRINT statement 62
8.6.1. Sequential file PRINT 62
8.6.2. Random file PRINT 63
8.7. PRINT USING statement 63
8.7.1. Sequential file PRINT USING 63
8.7.2. Random file PRINT USING 64
8.8. PUT statement 64
8.8.1. Sequential file PUT 64
8.8.2. Random file PUT 64
8.9. GET statement 65
8.9.1. Sequential file GET 65
8.9.2. Random file GET 65
8.10. CLOSE statement 65
8.11. DELETE statement 66
8.12. MOUNT and DISMOUNT statements 66
8.13. LOCK and UNLOCK statements 67
8.13.1. LOCK FILE statement 67
8.13.2. UNLOCK FILE statement 67
8.13.3. LOCK record statement 67
8.13.4. UNLOCK record statement 68
8.14. File related functions 68
8.14.1. RENAME function 68
8.14.2. SIZE function 68
8.14.3. MOUNT and DISMOUNT functions 69

9. Using QBASIC files 70
9.1. General file characteristics 70
9.2. File organisation 70
9.2.1. Stream files 71
9.2.2. Fixed files 71
9.3. Appending to a stream file 72
9.4. Device files 72
9.5. File and record locking 73
9.5.1. Using file lock 74
9.5.2. Using record lock 74

9.5.3. The UNLOCK program 75

10. Hardware and machine language interface 77
10.1. Memory inspect and change 77
10.1.1. PEEK function 77
10.1.2. POKE statement 77
10.2. Hardware input and output 78
10.2.1. INP function 78
10.2.2. OUT statement 78
10.3. Assembly language interface 78
10.3.1. CALL statement 79
10.3.2. Writing assembly language subroutines 79
10.3.3. Writing assembly language functions 81
10.3.4. Library entries available 82
10.3.5. Linking assembly language with QBASIC 84

11. Compiler directives 85
11.1. $INCLUDE — Copy source file 85
11.2. $DEBUG — Print line numbers in error messages 85
11.3. $DIAGNOSTIC — Compiler debugging feature 86
11.4. Ignored compiler directives 86

12. Separately compiled routines 87
12.1. Main programs, subprograms, and modules 87
12.2. SUBPROGRAM statement 87
12.3. ENTRY statement 88
12.4. EXTERNAL statement 88
12.5. Linking separate modules 89
12.6. EXTERNAL variables in the library 89

13. Reserved words (keywords) 91

14. Error messages 92
14.1. Compiler error messages 92
14.2. Second pass errors 93
14.3. Runtime error codes 93

15. Comparison of QBASIC with CBASIC 95
15.1. Restrictions present in QBASIC 95
15.2. Features treated differently in QBASIC and CBASIC 96
15.3. Restrictions removed in QBASIC 96
15.4. Extensions to CBASIC 97

16. Differences from earlier QBASIC releases 99
16.1. Changes in release 5.0 99
16.1.1. Transparent changes 99
16.1.2. Nontransparent changes 99
16.1.3. Extensions 100
16.2. Changes in release 2.0 100
16.2.1. Things to watch out for 100
16.2.2. Extensions 101

1. Introduction

QBASIC is a compiler for BASIC which runs on, and produces code for, the Marinchip 9900 computer system. The dialect of BASIC accepted by QBASIC is essentially the same as that used by the CBASIC (tm) compiler (version 2) developed by Compiler Systems of Sierra Madre, California. Unlike CBASIC, a pseudo-compiler which generates internal code which is interpreted at run time, QBASIC is a true compiler, which generates threaded code executable on the 9900.

Since QBASIC is a true compiler, programs compiled by it will generally run much faster than programs run under CBASIC. Since the compiled program is linked with the runtime library, QBASIC saves space by including only the support subroutines required by a program. For example, if a program does not use floating-point arithmetic, the floating-point package will not be included in the executable program. A QBASIC program which uses a minimal subset of support routines will occupy less than 4K bytes when executed.

The QBASIC language, being essentially compatible with CBASIC, is superbly suited to commercially-oriented programming. Features such as formatted output (PRINT USING), random access files, and the WHILE-WEND construct make for programs that are easy to write and maintain. The provision of long variable names (up to 31 characters) with full significance, and the fact that line numbers are required only on statements to which control may be explicitly transferred (by a GO TO or GOSUB, for example), removes the two major barriers which prevented development of large, complex programs in earlier versions of BASIC.

QBASIC contains numerous extensions not present in CBASIC. These extensions permit programs to be more efficient and expressive, and remove several severe limitations in CBASIC. Programs which are intended to run on both languages should, of course, avoid use of the extensions in QBASIC. A complete list of extensions and differences between CBASIC and QBASIC appears at the end of this manual.

1.1. Notation

The following no

Items enclosed in corner brackets, like <this>, refer to information supplied by the user. The text will explain what should be supplied.

Items enclosed in square brackets, like [this], refer to optional information which may be supplied if desired but is not required by the compiler. The text will explain the action taken when the optional item is omitted.

The ellipsis, "...", indicates that the preceding item may be repeated any number of times. For example, the sample text:

```
EAT <meat>,<vegetable>[,<cheese>...]
```

would describe the following statements:

```
EAT BEEF,POTATOES
EAT PORK,SPINACH,CHEDDAR
EAT CHICKEN,CARROTS,BRIE,JARLSBERG,GORGONZOLA
```

Examples given in the text are given in UPPER CASE type. Input to the compiler may be either upper case or lower case. The case of the input is insignificant except within String constants.

.2. Acknowledgements

The QBASIC compiler was designed and implemented by Mike Riddle of Evolution Computing, Phoenix, Arizona. Dan Drake of Marinchip Systems was responsible for the extensive testing required to insure compatibility between CBASIC and QBASIC, and several extensions to the system which added convenience and power to the language.

QBASIC is fully supported by Marinchip Systems. All questions regarding QBASIC should be directed to Marinchip Systems, as for any other Marinchip product.

2. Using QBASIC

This chapter describes how to compile and execute QBASIC programs. This chapter assumes that you have already written a properly-formed QBASIC program, and have placed its text in a file using one of the standard text editors, EDIT or WINDOW.

Beginning with level 2.0, QBASIC uses different libraries on Disc Executive and Network Operating System. This is necessary in order to make proper use of the file facilities in NOS without including large amounts of extra code in the library routines. Therefore, it is no longer possible to take a LINKed program from a Disc Executive disc, copy it to NOS with the CONVERT utility, and execute it. See "Network Operating System instructions" for more information on transferring programs.

2.1. Disc Executive instructions

The Disc Executive version of QBASIC is supplied on a diskette which replaces the normal system disc in drive 1 while work with QBASIC is being performed. One normally boots the system initially from the regular system disc, then removes that disc from drive 1 and installs the QBASIC release disc. The QBASIC disc contains the utilities EDIT, DIR, and CREATE/DELETE, as well as the QBASIC compiler and its runtime library, so this disc may be used for most program development functions as well as for actual QBASIC compilations.

Your QBASIC program will normally be edited into a file on disc 2. You must also create a file for the relocatable code output from the compilation, and for the final executable program. These files are normally placed on drive 2. The normal naming convention used is that the source program name ends in ".BAS", the relocatable in ".REL", and no type designation is used after the object file name. For example:

```
PRIMES.BAS          - The QBASIC source program
PRIMES.REL          - The relocatable code file
PRIMES              - The executable program
```

If the naming of files follows these conventions, and the program is of moderate size, the calls on the QBASIC compiler are very simple. These simple calls will described in the next few paragraphs. If files are not named according to these conventions, or the source, relocatable, and executable files are not on the same disc, or the program is large enough to require a special scratch file for the compiler, see the next section for the proper commands to use.

The compilation is performed by the command:

```
QBASIC <program>
```

where <program> is the name of the program, including the disc unit but without the suffix .BAS or .REL. For example,

```
QBASIC 2/PRIMES
```

To get a listing of the program as it is compiled, you can use:

```
QBASIC <program>,,<listing>
```

where <listing> is the file on which you wish the printed output, such as PRINT.DEV. Note the two commas. For example:

```
QBASIC 2/PRIMES,,PRINT.DEV
```

If errors are detected during the compilation, they will be indicated by messages identifying the nature of the error and the location where the compiler detected the error. Once an error has been detected, the rest of the program will be scanned by the compiler to detect other errors, but no output will be produced by the compiler. All errors must be corrected before a relocatable output will be produced.

Once the program has been successfully compiled, it must be linked with the QBASIC runtime library. This is normally accomplished by the command:

```
QLINKER <program>
```

This will take <program>.REL, pick up the necessary library routines, and write the executable output in file <program>. In the example above, we would have:

```
QLINKER 2/PRIMES
```

If the executable file has a different name from the relocatable, or it is on a different disc unit, see the end of this section.

The linker will produce a memory map for the program, write the object program to the output file, and return to the operating system. The object program may then be executed simply by typing its name, for example:

```
2/PRIMES
```

If the program files are not named <program>.BAS and <program>.REL, or they are on different discs, you must give the input and output file names explicitly:

```
QBASIC <reloc>=<source>
```

If there is no period in a name, the compiler always tacks on .BAS or .REL. Therefore, if you have a source file named "SOURCE" (with no .xxx after it), you must enter it as "SOURCE."; for instance:

```
QBASIC 2/OUTPUT=SOURCE.
```

which will compile from 1/SOURCE into 2/OUTPUT.REL.

The full form of the QBASIC compilation command is as follows:

```
QBASIC <reloc>[=<source>][,[<interpass>][,<listing]]
```

where <reloc> is the name of the file where the relocatable output of the compiler will be placed; <source> is the name of the file containing the QBASIC source program; <interpass>, if specified, is

where the intermediate code passed between the two passes of the compiler will be stored; and <listing>, if specified, is where the program listing will be produced.

If no =<source> is given, the compiler will assume that the source and relocatable files have the same name, distinguished by the suffixes .BAS and .REL.

If no <interpass> specification is given, this code will be written into the file TEMP1$ on the QBASIC release disc (or in the assumed directory, in case of NOS). <interpass> is usually specified on the QBASIC call only when compiling a very large program for which the TEMP1$ file is insufficient in size. In such cases, a large intermediate file on drive 2 is usually created and named on the QBASIC call. When compiling a very large program, you would use a command like:

```
QBASIC 2/BRAINSIM,2/TEMP
```

In linking the program, you may want the relocatable and executable files to have different names or to reside on different disc units. In this case you can use the form:

```
QLINKER <executable>=<reloc>
```

As in earlier releases of QBASIC, you may also link the program with the command:

```
LINK <object>=<reloc>,@QLINK
```

Or, in the Network Operating System,

```
LINK <object>=<reloc>,@1:QBASIC/QLINK
```

2.2. Network Operating System instructions

Using QBASIC under NOS/MT is essentially the same as under Disc Executive, but there a few changes which allow the use of the extra features of NOS/MT.

The NOS/MT version of QBASIC, like the Disc Executive version, is supplied on a diskette that is MOUNTed in drive 1 to act as a system disc while QBASIC work is being performed. (Instructions for copying QBASIC to a hard disc are at the end of this section.) To begin QBASIC work:

```
Type DISMOUNT 1:
Remove the normal system disc and insert the QBASIC disc
Type MOUNT 1:
```

The QBASIC disc contains the utilities EDIT, DIR, and CREATE/DELETE, as well as the QBASIC compiler and its runtime library, so this disc may be used for most program development functions as well as for actual QBASIC compilations.

Your QBASIC program will normally be edited into a file on disc 2. There must also be a file for the relocatable code output from the

compilation, and for the final executable program. You need not CREATE these in advance, since NOS/MT will do it automatically when you compile and link the program. The normal naming convention used is that the source program name ends in ".BAS", the relocatable in ".REL", and no type designation is used after the object file name. For example:

```
PRIMES.BAS          - The QBASIC source program
PRIMES.REL          - The relocatable code file
PRIMES              - The executable program
```

If the naming of files follows these conventions, and the program is of moderate size, the calls on the QBASIC compiler are very simple. These simple calls will described in the next few paragraphs. If files are not named according to these conventions, or the source, relocatable, and executable files are not in the same directory, or the program is large enough to require a special scratch file for the compiler, see the end of the preceding section for the proper commands to use.

Before performing a compilation, you must ASSUME a directory. QBASIC will use this directory to hold a temporary file used in the compilation; therefore, the directory should be a private one, not accessible to anyone else who might be using QBASIC at the same time. The obvious choice of a file to ASSUME is the one in which your relocatable and/or executable program will go.

With the normal conventions for naming the files, the QBASIC compilation is performed by the command:

```
QBASIC <program>
```

To get a listing, the command is:

```
QBASIC <program>,,<listing>
```

where <listing> is the file in which the listing will be produced, such as PRINT.DEV. Note the two commas.

To perform a normal QBASIC compilation, one might use commands such as:

```
ASSUME 2:PRIMEPROG
QBASIC PRIMES,,PRINT.DEV
```

If errors are detected during the compilation, they will be indicated by messages identifying the nature of the error and the location where the compiler detected the error. Once an error has been detected, the rest of the program will be scanned by the compiler to detect other errors, but no output will be produced by the compiler. All errors must be corrected before a relocatable output will be produced.

Once the program has been successfully compiled, it must be linked with the QBASIC runtime library. This is accomplished by the command:

```
QLINKER <program>
```

For example, to link the PRIMES program mentioned in the sample compiler call above, you would use:

```
QLINKER PRIMES
```

The linker will produce a memory map for the program, write the object program to the output file, and return to the operating system. The object program may then be executed simply by typing its name, for example:

```
PRIMES
```

If the executable file is to be in a different directory from the relocatable, you can link the program:

```
QLINKER <executable>=<reloc>
```

See the end of the preceding section for more information.

2.2.1. Converting programs from Disc Executive

If you have a program that has been developed under the Disc Executive, it is easy to carry it over to NOS. Simply use the CONVERT utility to copy program.BAS and program.REL (if you have both) to the NOS disc. Then go through the LINK procedure described above, to make the executable file. It is not necessary to recompile the program with QBASIC if you already have a REL file. Remember, though, that a program which uses file names may have to be modified to use the slightly different file naming conventions that apply in NOS. For programs that want to know which system they are running under, see the section "EXTERNAL variables in the library."

QBASIC data files can be carried over directly by the CONVERT utility, with no change.

2.2.2. Using hard disc

The entire QBASIC system can be copied to a hard disc, with all its advantages of large capacity, high speed, and lack of constantly mounting and dismounting discs. All that is required is to copy the proper files from the release disc to the hard disc. The release disc includes a script to automate the process. Both the script and the instructions given here assume that disc 1 is a hard disc and disc 2 is a floppy; if this is not the actual configuration, the instructions should be modified appropriately, and the script should be edited before use. The procedure is as follows:

1. Log in as the privileged user. (See the section "Privileged mode" in the NOS user guide.)
2. Put the QBASIC release disc in drive 2.
3. MOUNT 2:
4. If the system does not have hard disc on unit 1 and floppy disc on unit 2, edit the file QBASIC/HDISC.SCR on the release disc.
5. SCRIPT 2:QBASIC/HDISC.SCR
6. DISMOUNT 2:

The operating instructions for QBASIC are exactly the same as in a

FLOPPY DISC SYSTEM.

3. Language syntax

This chapter describes the components that make up a QBASIC program and gives the rules for combining these components to form complete QBASIC programs. Subsequent chapters will describe specific QBASIC statements.

3.1. Statements

A QBASIC program consists of one or more STATEMENTS. Each statement performs a specific function in the language, such as assigning a value to a variable, reading or writing data, transferring control to another statement, or making a decision.

QBASIC statements are written in free format: any number of statements may appear on one line, or one statement may continue for any number of lines (there are a few exceptions to these rules; they will be noted in the text).

3.1.1. Line numbers

Any statement (except the declarations COMMON, ENTRY, EXTERNAL, and SUBPROGRAM) may be preceded by a LINE NUMBER. Line numbers are used to identify statements to which control is transferred by a GOTO, GOSUB, ON, or IF statement. Although any statement may have a line number, only those statements to which control is explicitly transferred must have line numbers. In general, programs are easier to read and maintain if line numbers are used only when required, as it is easy to identify the flow of control in the program.

Line numbers consist of strings of digits and decimal points, with optional exponent specifications. A line number may be of any length, but only the first 31 characters are significant. The first character of a line number must be a decimal point or a digit; subsequent characters may be decimal points, digits, the letter "E", or plus or minus signs. Note that line numbers are treated as symbols by the compiler, not as numbers: hence the line numbers

1234 and 1234.00

are distinct.

Unlike other versions of BASIC, QBASIC line numbers need not be used in ascending numeric order. Line numbers are simply statement labels, and may be used in any order desired by the programmer, as long as no two statements have the same label.

The following are valid QBASIC line numbers:

```
1
0100
226762.33
1.23E-3
6.02E+23
3.141596235
.36788
```

```
217.50.0320
12.........22......11
...E++-.72-?EE.-
```

3.1.2. Continuations

Statements may be continued from line to line through the use of the backslash character, "\". When a backslash is encountered, the rest of the line containing it will be ignored, and the statement will be continued onto the next line. A backslash may be used wherever a space may be used, but may not be used in the middle of keywords, variable names, or constants.

3.1.3. Multiple statements per line

Multiple statements may be written on one line by separating them by colons, ":". Any number of statements may be written on one line in this manner. See the description of the IF statement later in this manual for important uses for this feature.

QBASIC allows REMARKs to be added at the end of any statement (except a DATA statement) without the need for a colon before the REMARK. See the description of the REMARK statement below for more information.

In CBASIC, the statements DATA, DEF, DIM, and END must appear on lines by themselves, and IF must be the first statement on a line. QBASIC removes these restrictions; these statements may be combined with each other and with other statements as long as they are separated by colons.

3.1.4. Significance of spaces

In QBASIC, spaces are significant. This means that whenever two words appear adjacent to one another, one or more spaces must appear to separate the words. Also, no spaces may appear within words or numbers. Wherever one space is permitted, any number of spaces may appear. QBASIC's handling of spaces is identical to CBASIC's.

3.2. Comments

QBASIC offers two ways of including comments in programs: the REMARK statement, which is compatible with CBASIC and many other BASIC implementations, and the "embedded comment", which was borrowed from Pascal and is often far more convenient to use.

Neither form of comment occupies space in the executable program generated by the compiler, nor does the inclusion of comments affect the execution speed of a program in any way. Hence, the user should feel free, and is strongly encouraged, to include comments describing the function and operation of QBASIC programs.

3.2.1. REMARK Statement

```
[<line number>] REMARK <text>
[<line number>] REM <text>
```

A REMARK statement, which may be abbreviated REM, is used to include comments in a program. The rest of the line following the word REMARK or REM will be totally ignored by the compiler. If a <line number> is present, it may be used as the object of a GO TO or other control transfer statement. Note that a space must follow the word REM or REMARK. Note especially that since the compiler ignores the <text> in a REMARK, a colon may not be used to include a statement on a line following a REMARK. For example, the following statement:

```
100 REMARK  Set A to 1 : A=1
```

is totally a REMARK. The statement A=1 is taken as part of the REMARK text and is not executed.

The only exception to REMARK text being totally ignored is that a backslash, "\", may be used to continue the REMARK onto another line. Hence, the following is permitted:

```
120 REMARK  This program sorts an input array \
            and stores the sorted output in   \
            a disc file.
```

Be careful not to use a backslash in a REMARK by accident, as it will cause the next line to be ignored, regardless of its content. This feature of continued REMARKs is compatible with CBASIC, although the CBASIC manual does not mention that REMARKs may be continued.

An unlabeled REMARK statement may follow any other statement without an intervening colon. Such a REMARK turns the rest of the line after the word REMARK into a comment. For example:

```
ZORCH=YIBBLE+FRIZZBAT    REMARK  Compute sum
```

This feature is compatible with CBASIC. QBASIC programmers may find that "embedded comments", described below, are more convenient.

3.2.2. Embedded comments

QBASIC treats all characters occurring between a left curly bracket, "{", and a right curly bracket, "}", as a comment. The entire comment will be treated by the compiler as equivalent to a single space, and hence may not be embedded within a name or constant. Embedded comments are not recognised within String constants, so curly brackets may be used in String constants without difficulty.

Embedded comments do not "nest". An embedded comment is terminated by the first right curly bracket encountered, regardless of whether any more left curly brackets were encountered first. Embedded comments may continue from line to line, and hence are useful for including large block comments. The following are examples of embedded comments:

```
A=B*SIN(C)    { Compute next element }

Q=FN.COMP(PREV.DATA { Old data }, TRANS.ID { Op code })
```

```
PRINT "Result is ";FINAL.RESULT    ( Print the result
                                     from the computation and
                                     label it for the user )
```

3. Constants

Constants are elements of the language that stand for values. There are three types of constants in QBASIC: INTEGER constants, which stand for positive or negative whole numbers, REAL constants, which represent decimal fractions, and STRING constants, which represent strings of characters.

3.3.1. Integer constants

Integer constants are whole numbers in the inclusive range from -32767 to +32767. Integer constants are written without a decimal point, as any number with a decimal point will be taken as Real even if the decimal fraction is zero and the number is within the Integer range. Integer values are stored by QBASIC as sixteen bit two's complement binary numbers.

Integer constants may also be written in hexadecimal by preceding them with a leading zero. The following are hexadecimal constants:

```
0100 0FFC 01B0C
```

Note that a leading zero must not be used on a decimal constant, as it would cause the constant to be interpreted as hexadecimal. This convention for hexadecimal numbers differs from CBASIC, which uses a trailing "H" to denote such numbers. CBASIC's trailing "B" for binary numbers is not supported in QBASIC.

3.3.2. Real constants

Real constants are of the form:

```
[<sign>]<whole>.[<frac>][E[<esign>]<exp>]
```

Where <sign> is the optional sign of the number (+ or -), <whole> is the digits for the whole part of the number, <frac> is the digits for the fractional part of the number, <esign> is the optional exponent sign, and <exp> is the power of ten to which the number should be raised.

Note that a Real number must contain a decimal point. The only exception is that numbers with no decimal point are treated as Real if they are too large for Integer representation (absolute value greater than 32767). Real numbers are stored by QBASIC as double precision numbers with 14 or more digits of accuracy. The range of exponents permissible is between 1.0E-64 to 1.0E64. The following are properly formed Real constants:

```
0.0 1.0 -18.2133 1.238 .1 0.

0.000121 8738731.223E4  -1.225E-54  999999
```

3.3.3. String constants

A String constant represents zero or more characters. A String constant is written by enclosing the characters in quotation marks, ("). A simple String constant might be:

"Cosmic muffin"

A quotation mark can be included in a string by writing two adjacent quotes. For example, the text:

"Halt", I said, "I'll shoot".

would be written as a string constant as follows:

""Halt"", I said, ""I'll shoot"."

The string with zero characters is called the NULL STRING, and is written as:

""

Since all characters other than the quote mark are taken as part of the string text, string constants may not be continued onto multiple lines, nor may they contain embedded comments.

3.4. Variable names

VARIABLE NAMES are user-defined names which may be assigned values. As for constants, there are three types of variables: Integers, Reals, and Strings. The type of a variable name is indicated by its last character.

Variable names consist of a letter followed by letters, numbers, and periods. A variable name may be of any length, but only the first 31 characters of the name are significant. The last character of a variable name identifies its type: if the name ends in a percent sign, "%", the variable represents an Integer; if the name ends in a dollar sign, "$", a String; and if it ends in neither a percent nor a dollar sign, a Real.

QBASIC's reserved words (statement names, function names, and so on) may not be used as variable names. At the end of this manual a complete list of QBASIC reserved words is given. Because of the convention used to identify user-defined function names, no variable name may begin with the letters "FN".

Examples of Real variables are:

J I INDEX NEW.VALUE
MAXIMUM.MEASURED.SO.FAR J9920 Q21.INDEX3

Examples of Integer variables are:

LOOP.COUNTER% J% T1902.B7%

Examples of String variables are:

A$ CUSTOMER.ADDRESS.LINE1$ Q7..5..$

Note that the variables:

I I% I$

are all distinct: they represent a Real, an Integer, and a String, and may all occur in the same program and have no connection with each other.

All variables have values associated with them at all times. If an Integer or Real variable is used before it is assigned a value, it will have a value of zero. A String variable will have a value of the null string (""). Integer and Real variables take on values within the limits described above for Integer and Real constants. String variables may be assigned any string desired. There is no limit on the length of the value assigned to a string variable; the only limitation is the memory available on the computer.

3.4.1. Subscripted variables

Variables may represent vectors or arrays of values by being subscripted. Subscripted variables may represent Integers, Reals, or Strings, like any other variable, but must always be used with a subscript list which selects the specific value being used. Unlike non-subscripted variables, subscripted variables must be declared via the DIM statement before being used (see the description of the DIM statement later in this manual for details).

The following are examples of references to subscripted variables:

METER.READING(3,12) TEMP(Q%,DTIME.R+12)
IVAL%(IDX%) CANNED.MESSAGE$(CONTEXT%,MLINE%)

Subscripted variables may have any number of subscripts, but the number of subscripts used must agree with the number declared in the DIM statement. Subscript values must be numeric: an error will occur if a string is used as a subscript. If a Real value is used, it will be rounded to an Integer. For greatest efficiency, all subscripts should be Integers. Subscripts range from zero to the highest value declared in the DIM statement; a subscript outside the declared range will cause an error.

The handling of each item in a subscripted variable is exactly as described above for non-subscripted variables.

Note that a subscripted and non-subscripted variable are distinct, even if they have the same name and type. For example, the following statement:

```
V=V(2,3)
```

is completely valid, and sets the non-subscripted variable V to the value of the designated element of the subscripted variable V. This rule, combined with the distinctness by types mentioned above, can result in statements as silly as:

```
A$=A$(A(A%),A%(A))
```

which contains six completely different variables. Users are warned that extensive use of this "feature" can lead to programs that are very difficult to comprehend and expensive to maintain.

3.5. Expressions

Expressions are combinations of constants, variables, and function references, with OPERATORS. Expressions, like constants and variables, take on values of type Integer, Real, and String. The rules for determining the type of an expression will be explained below.

The simplest expression consists of a single constant or variable name. In that case, the expression's value is simply the value of the constant or the current contents of the variable. More complex expressions are formed by combining values with operators, which are discussed in the following section.

3.5.1. Operators

Operators are used, along with the normal rules of algebra, to modify and combine constant and variable values. Operators fall into several subgroups.

3.5.1.1. Arithmetic operators

```
+  -  *  /  MOD  ^
```

The arithmetic operators have their normal meanings of addition, subtraction, multiplication, division, and exponentiation. If the two values combined by the operator are both Integer, the result will be Integer. If the values are both Real, the result will be Real. If the values differ in type, the Integer value will be converted to Real, then the operation will be done in Real arithmetic yielding a Real result.

Note that the result of division of two Integers is the whole part of the result, with any remainder discarded. To get a Real result, one or both of the Integers must first be converted to a Real before performing the division.

The result of the operator "MOD" is the remainder when the left operand is divided by the right operand. If either operand is Real, the remainder will be calculated as a Real; for instance, 4 MOD 1.25 will give 0.25. The sign of the result of MOD is the sign of the first operand: -5 MOD 3 and -5 MOD -3 are both -2. This rule, which was not applied correctly in earlier versions of QBASIC, assures the identity:

```
I% = J% * (I%/J%) + (I% MOD J%)
```

Exponentiation (performed by the "^" operator) is performed in two different ways depending on whether the operands are Integer or Real (if they differ, the Integer is converted to Real). Integer to

nteger power is performed by successive multiplication. The base may be positive or negative. If the exponent is negative, the result will e zero. Real to Real power is performed by taking the logarithm of he base, multiplying it by the exponent value, and then taking the exponential of the result. Since the logarithm of a negative number is undefined, negative Real numbers should not be used as the base ith the "^" operator. The value of 0^0 is 1; 0^X, where X is nonzero, is zero.

tring variables may not be used with any of the arithmetic operators xcept "+" and "*". For string operands, "+" has the special meaning of CONCATENATION. The result of the "+" operator on two strings is he string consisting of the left string with the right string ppended to it. For example, in the following program fragment:

```
A$="Big bad "
B$="bear."
C$=A$+B$
```

he variable C$ will contain "Big bad bear."

The operator "*" can be used between a String expression and an nteger expression to duplicate the string. Thus, the statements

```
A$=" "*9
I% = 7
B$=(A$+"*")*(I%+1)
```

will set A$ to a string of 9 blanks and B$ to a string of eighty haracters with an asterisk in every tenth position and blanks elsewhere. If a Real expression is used as the second operand, it will be rounded to an Integer.

he operators "+" and "-" may be used as UNARY operators. A unary operator has no operand on the left, and one operand on the right. he "+" operator has no effect on the right operand, and the "-" perator reverses the sign of the right operand. For example, if the variable B has a value of 12, the statement:

```
A=-B
```

will set A to -12.

3.5.1.2. Relational operators

=	>	>=	<	<=	<>
EQ	GT	GE	LT	LE	NE

he relational operators compare their two operands and return a value epending on whether the relationship being tested holds for the perands supplied. Either the algebraic relational operators given in he first row, or the mnemonic forms given in the second row, may be ed. Relational operators may be used between Integer, Real, and tring operands.

an Integer and Real are compared, the Integer will be converted to Real before the comparison is done.

If one operand is a String, both must be Strings. String comparison is performed from left to right, based on the ASCII collating sequence. The end of a string is treated as if it were a character that collates before any actual character. That is, ABC is less than ABCx, where x is any character whatever.

The result of a relational operator is always an Integer. If the relation holds, the result is -1, which signifies TRUE. If the relation does not hold, the result is 0, which signifies FALSE. Examples of relational expressions and their result are given in the following table:

Expression	Result
1=1	-1
1=1.0	-1
1=1.01	0
1>12	0
1<12	-1
-1<=0	-1
"Me"<"My"	-1
"AA"="Aa"	0
"Za">"Z"	-1

3.5.1.3. Logical operators

NOT AND OR XOR

The logical operators perform bit-by-bit logical operations on their operands. The operands are treated as 16 bit Integers. Real operands will be converted to Integer before the operation is performed (note that there is no test for overflow in this conversion, so if the Real operand is outside the range -32768 to 32767, undesirable results will be obtained). The operator "NOT" is a unary operator which inverts all the bits in its operand to the right. The operators "AND", "OR", and "XOR" perform the indicated operations on their two operands.

Note that since the values for TRUE (-1) and FALSE (0) generated by the relational operators are all bits set and clear respectively, the logical operators may be used to combine the results of relational tests into logical expressions. For example, the following expression tests whether a single character string, A$, is numeric:

```
A$ >= "0" AND A$ <= "9"
```

Note, however, that NOT I% is DIFFERENT from I%=0. In fact, if the value of I% is neither 0 nor -1, then both I% and NOT I% will have TRUE (non-zero) values. There is no problem as long as logical operators are applied only to 0, -1, and the results of comparisons.

In addition, the logical operators may be used to perform bitwise operations on integers. Such operations are especially useful when dealing with hardware-defined items, since interface to them often involves testing, setting, and clearing individual bits.

3.5.2. Order of evaluation

When QBASIC encounters an expression containing more than one operator, it applies a set of rules so that the expression will have the value expected from the normal rules of algebra. For example, the expression:

```
A+B*C
```

is taken to mean that B and C are multiplied, then the result is added to A. The order of evaluation may be modified by grouping parts of the expression in parentheses. The expression:

```
(A+B)*C
```

causes A and B to be added, then the result multiplied by C. The order in which QBASIC will evaluate operators in an expression is given in the following table.

```
 1)   ( ),
 2)   ^
 3)   Unary +, Unary -
 4)   *  /
 5)   MOD
 6)   +  -
 7)   =   <>  <   >   <=   >=
      EQ  NE  LT  GT  LE   GE
 8)  NOT
 9)  AND
10)  OR  XOR
```

This order of evaluation is the same as that used by CBASIC except that Unary + and Unary - are performed before multiplication and division by QBASIC, but in CBASIC they have no priority over addition and subtraction. QBASIC's interpretation is in keeping with the rules of algebra and virtually all other programming languages.

When QBASIC processes an expression, all operators will be evaluated in the order given in this table. Hence, all multiplications and divisions will be done before any additions. The fact that parenthesised subexpressions have the highest priority insures that they will be evaluated before anything outside the parentheses. Parentheses may be nested to any level desired.

Familiarity with the rules of operator hierarchy often makes it possible to write very complex expressions without parentheses. Although this is possible, it is not recommended, since the next person who examines the program may not know the rules as well as the author, and may misinterpret the intent of the expression. Often including parentheses which are redundant can make an expression easier to follow. Inclusion of redundant parentheses does not affect the output of the compiler in any way, so there is no speed or memory penalty for using them. As an example, compare the ease in understanding the following two equivalent expressions:

```
A+B<=B*2 OR B+C<=19 AND NOT A+4=8
((A+B)<=(B*2)) OR (((B+C)<=19) AND (NOT (A+4)=8))
```

4. Assignment and control statements

This chapter describes the statements which make up the "core" of the QBASIC language. These statements perform the basic functions of setting variables, making decisions, and determining which statements will be executed and in what order. Later chapters will discuss input, output, and other statements.

Any of the statements described in this chapter (except COMMON) may have a line number, so line numbers will not be mentioned in these descriptions.

4.1. LET (assignment) statement

```
[LET] <variable>=<expression>
```

The value of the <expression> is computed and stored into the <variable>. If the value of the <expression> is a String, the <variable> must be a String variable. If the <expression> is Integer and the <variable> is Real, the value will be converted to Real. If the <expression> is Real and the <variable> is Integer, the result will be converted to an Integer by truncation: the fractional part, if any, will be dropped. This is one of only two cases (the other being the INT% function) in which Reals are converted to Integer by truncation rather than rounding.

The keyword LET is optional in the LET statement. The <variable> may be subscripted, if desired. The following are examples of LET statements:

```
LET A%=1
A=B/D+4
A$="Pea soup"
PLURAL.WORD$=SINGULAR.WORD$+"s"
XARRAY(2,C%+1)=VTAB(J%)
```

4.2. GOTO statement

```
GO TO <line number>
GOTO <line number>
```

The next statement executed will be the statement with the specified <line number>. Examples are:

```
GO TO 123
GOTO 0.3736
```

4.3. GOSUB statement

```
GO SUB <line number>
GOSUB <line number>
```

The location of the statement following the GO SUB is saved, then control is transferred to the statement with the named <line number>. Control may be returned following the GO SUB by executing a RETURN statement (described in the next section). Each GOSUB consumes memory

to save the return point. This memory is released when the corresponding RETURN is done, so programs should be careful to always RETURN from GO SUB calls.

4.4. RETURN statement

RETURN

The RETURN statement causes control to be passed to the statement following the most recent subroutine call. The subroutine call may be a GOSUB statement, an ON -- GOSUB statement, or a multiple line function call (described later in this manual).

4.5. IF statement

The IF statement allows a program to perform different actions based on the value of an expression. If the value is non-zero, the program will perform some specified action; if it is zero, there may or may not be alternate actions to perform.

QBASIC supports two forms of IF statement. The first is the Single-line IF, which is entirely contained on one line, possibly extended by continuation (\) characters. The second is the Block IF, in which any number of lines may appear between IF and ENDIF statements.

4.5.1. Single-line IF

IF <expression> THEN <stmt list/line number>
[ELSE <stmt list/line number>]

The <expression> is evaluated, then its value is tested against zero. If the <expression> is nonzero, the action described in the THEN clause will be taken. If the <expression> is zero, and the ELSE clause is present, the action described in the ELSE clause will be taken; if no ELSE clause is present, the IF statement will do nothing and control will pass on to the next statement. The <expression> in an IF statement must evaluate to an Integer or Real. Normally the expression in an IF statement is the result of one of the relational operators, but any Integer or Real expression may be used. If the value is Real, it will be converted to Integer by rounding.

The <stmt list/line number> which follows the THEN and/or ELSE in an IF statement may be either a line number or one or more QBASIC statements (with multiple statements separated by colons, as always). If a line number is specified, execution of the clause will transfer control to the desisnated line number.

If a statement list is used as the THEN or ELSE clause, any number of statements may make up the clause. The statement may be continued onto as many lines as required by using the backslash (\) character. Unlike CBASIC, QBASIC allows an IF statement to appear in the ELSE clause of another IF, or in the THEN clause of an IF that has no ELSE. (These awkward restrictions are absent from the Block IF.) Also, either the THEN or the ELSE clause or both may be line numbers. CBASIC disallows an ELSE clause if the THEN clause is a line number.

The following examples illustrate IF statements in QBASIC.

```
IF EPSILON<0.0005 THEN 1800
IF EPSILON<0.0005 THEN GO TO 1800
```

The above two statements are equivalent.

```
IF CH$="s" THEN 1400 ELSE PRINT "Wrong!"
IF J<K+8 THEN J=K+8 ELSE 1200.30

IF NOT.DONE.YET THEN \
   J=J+1 :\
   IF J>10 THEN \
     PRINT "Overflow!" :\
     GO TO 200

IF TIME>QUITTING.TIME THEN \
   PRINT "Time to go home!" : \
   GOSUB 1800  :\
   TIME=0 :\
   GO TO 100 \
ELSE \
   TIME=TIME+1 :\
   GOSUB 3800
```

Note that no colon is required after the THEN nor before or after the ELSE.

4.5.2. Block IF statement

```
IF <expression>
```

The Block IF allows multiple statements to be controlled by an IF without the use of ":\" as in the previous section. Unlike the Single-line IF, the Block IF can be nested without restriction.

When a Block IF statement is executed, the <expression> is evaluated according to the same rules used for the Single-line IF. If the value is non-zero, the code following the IF statement will be executed, up to the next matching ELSEIF or ELSE statement (if there is one); then everything will be skipped until the matching ENDIF statement. If the value is zero, the code following the IF will be skipped until there is a matching ELSEIF, ELSE, or ENDIF.

The following examples illustrate simple IF blocks.

```
IF INPUT.ERROR
   PRINT "Bad input"
   GOSUB 9100
   GOSUB 3947
ENDIF

IF NOT.DONE.YET
   J=J+1
   IF J>10
      GOSUB 1000
      J=1
```

```
    ENDIF
    GOSUB 2000
ENDIF
```

The first example could be replaced with a Single-line IF, using some ":\" sequences. The second, which has the GOSUB 2000 after the end of an IF condition, could not.

The Block IF can not appear inside a Single-line IF.

4.5.3. ENDIF statement

```
ENDIF
```

The ENDIF statement marks the end of a Block IF. See the description of the Block IF above for examples. ENDIF may not appear within a Single-line IF statement.

4.5.4. ELSEIF statement

```
ELSEIF <expression>
```

The ELSEIF statement provides an optional alternative to the code which is controlled by a Block IF statement.

If the <expression> in the preceding Block IF or ELSEIF has a value of zero, then the <expression> in the ELSEIF is evaluated. If the value is non-zero, the code in the following statements will be executed, up to the next matching ELSEIF or ELSE statement. If it is zero, the code will be skipped up to the next matching ELSEIF, ELSE, or ENDIF statement.

If the <expression> in the Block IF was non-zero, then the ELSEIF and the statements following it will be ignored.

There may be any number of ELSEIF statements in a single IF block.

The following is an example of the ELSEIF statement:

```
IF NOT.DONE.YET
   J=J+1
ELSEIF ERROR.STATUS
   PRINT "Error status";ERROR.STATUS
ELSEIF INTERRUPTED
   PRINT "What did you want to do?"
ENDIF
```

4.5.5. ELSE statement

```
ELSE
```

The ELSE statement, appearing in a Block IF, provides an alternative to be executed if neither the matching IF statement nor any of its ELSEIF statements causes anything to be executed. The following are examples of the ELSE statement:

```
IF INPUT.ERROR
```

```
      PRINT "Bad input"
      GOSUB 9100
      GOSUB 3947
   ELSE
      PRINT "OK"
   ENDIF

   IF NOT.DONE.YET
      J=J+1
   ELSEIF ERROR.STATUS
      PRINT "Error status";ERROR.STATUS
   ELSEIF INTERRUPTED
      PRINT "What did you want to do?"
   ELSE
      PRINT "Done"
   ENDIF
```

If the word ELSE appears inside a Single-line IF, it is always interpreted as part of that Single-line IF, never as part of a Block IF. Together with the fact that a Block IF can not appear inside a Single-line IF, this prevents any ambiguity.

4.6. WHILE statement

```
WHILE <expression>
```

The WHILE statement causes all statements between the WHILE statement and the next matching WEND statement to be executed as long as the value of the <expression> is nonzero. If the value of the <expression> is initially zero, the statements between the WHILE and WEND will not be executed. Statements within the WHILE - WEND loop may change variables in the <expression>, as it is reevaluated every time the WHILE statement is executed. WHILE - WEND loops may be nested.

The <expression> on the WHILE statement must evaluate to an Integer or Real value. If the value is Real, it will be rounded to an Integer before being tested against zero.

The following are examples of WHILE - WEND loops.

```
   WHILE V%<10
    PRINT "Starting pass ";V%
    GOSUB 1.5400
    V%=V%+1
   WEND

   WHILE -1
    INPUT I
    IF I>1000 THEN 1420
    PRINT I
   WEND
```

Note that the second example is an "endless loop" which is escaped only by the IF statement within it. Since the expression on the WHILE statement is a nonzero constant, the WHILE loop will never itself

terminate. There are no restrictions on jumping into or out of WHILE loops; the effect is always what would be expected.

4.7. WEND statement

```
WEND
```

The WEND statement serves to mark the end of a WHILE - WEND loop. See the description of WHILE above for more information about WHILE - WEND loops and examples. Note that the WEND statement is in effect a declaration to the compiler rather than an executable statement. Consequently, it must not appear as the object of a conditional statement such as an IF. For example, the following statement is INCORRECT:

```
IF J%>5 THEN WEND
```

A consequence of this fact about the WEND statement is that there must be one and only one WEND for each WHILE in the program.

4.8. FOR statement

```
FOR <index>=<expression> TO <expression>
    [STEP <expression>]
```

The FOR statement causes the statements between it and the corresponding NEXT statement to be executed a number of times determined by the values of the <expression>s used on the FOR statement. An <index> variable is set to a different value for each execution of the loop.

When a FOR statement is initially executed, the <index> variable is set to the value of the first <expression>. All statements in the loop are then executed. The <index> variable may not be subscripted, nor may it be of type String. Either a Real or Integer <index> variable may be used, but the FOR statement will be much more efficient if the <index> variable and all <expression>s are Integer.

When the end of the loop is reached, the TO and, if present, the STEP expressions are evaluated. If the STEP expression is present, its value is added to the <index> variable. If no STEP expression appears, the <index> variable will be incremented by one. The value of the <index> variable will then be compared with the value of the TO expression. If the STEP expression is positive or no STEP expression appears, then execution of the loop will stop if the <index> variable is greater than the TO expression. If the STEP expression is negative, execution of the loop will stop if the value of the <index> variable is less than the TO expression.

The TO and STEP expressions are reevaluated at the end of each execution of the loop. Hence they may be changed from within the loop by changing variables which are part of the TO and STEP expressions. The <index> variable may also be changed during the loop.

Unlike CBASIC, QBASIC does not require that the <index> variable and all the <expression>s have the same type. However, the program will be more efficient if they do. If a STEP of one is desired, it is more

efficient to omit the STEP clause than to specify a STEP of 1. If the STEP clause is present, the compiler must generate more complicated code than if the default of 1 is used.

The following are examples of FOR - NEXT loops.

```
FOR J%=2 TO 10
  PRINT "Yes, we have";J%;"bananas."
NEXT J%

FOR VOLTAGE=10.50 TO 0.0 STEP -0.05
  NEUTRON.COUNT=FN.LASERZAP(VOLTAGE)
  PRINT VOLTAGE,NEUTRON.COUNT
NEXT VOLTAGE
```

4.9. NEXT statement

```
NEXT [<variable>,...]
```

The NEXT statement identifies the end of the closest previous unmatched FOR statement. If a <variable> is specified, it must match the <index> variable on the corresponding FOR statement. More than one variable may appear on a NEXT statement, which allows one NEXT statement to terminate multiple nested FOR loops. The action of such a statement is identical to that of consecutive NEXT statements for each variable.

As with the WHILE and WEND statements, each FOR must be balanced by exactly one NEXT statement (with no variable list) or one item in a list of variables on a NEXT statement.

The following illustrate various forms of the NEXT statement:

```
FOR I%=1 TO 10
 FOR J%=1 TO 10
  FOR K%=1 TO 10
   A(I%,J%,K%)=1
NEXT K%,J%,I%

FOR I%=1 TO 10
 PRINT I%
NEXT
```

4.10. EXIT IF statement

```
EXIT IF <expression>
```

The EXIT IF statement provides a quick way of escaping from a loop without using a GO TO and a line number. When the EXIT IF statement is executed, the <expression> is evaluated in the same way as for WHILE or IF statements. If the value is zero, the next statement is executed. If the value is non-zero, the program exits from the current (innermost) loop or block of statements, as described below.

If the current loop is WHILE, the exit is to the statement immediately after the WEND statement.

If the current loop is FOR, the exit is to the statement immediately after the next NEXT. If the NEXT statement is of the form:

```
NEXT <variable 1>,<variable2>[,<variable 3>,...]
```

then the EXIT IF will execute NEXT <variable 2>.

If the current block is an IF block, the exit is to the statement immediately after ENDIF. EXIT IF is not valid inside a Single-line IF.

If the current block is a DEF block (multi-line function definition), the exit is the same as a RETURN statement.

The following is an example of the EXIT IF statement:

```
WHILE 1
   X=X+1
   Y=Y+Z
EXIT IF X*Y>LIMIT
   GOSUB 9900
WEND
```

4.11. ON statement

```
ON <expression> GO TO <line number>,...
ON <expression> GO SUB <line number>,...
```

The ON statement permits an expression to select one line number from a list. A GO TO or GO SUB is then done to the selected line number.

The <expression> is evaluated. If its value is Real, it is converted to Integer by rounding. The <expression> may not have a String value. If the value of the <expression> is 1, the first line number is selected. If 2, the second line number is chosen, and so on. If the <expression> evaluates to less than 1 or greater than the number of line numbers in the list, an error will occur. Once the line number has been selected, the ON statement acts exactly like a GO TO or GO SUB to the selected line number. The keywords GO TO and GO SUB may be written GOTO and GOSUB, respectively, if desired.

The following are examples of ON statements:

```
ON OP.CODE% GO TO 100,200,300,400
ON V GOSUB 1.100,1.200,1.300
ON SELECTION.CODE%+1 GO SUB \
    1200, 1300, 1400, 1500, 1600
```

4.12. STOP statement

```
STOP
```

Execution of a STOP statement terminates a QBASIC program. All open files are closed, writing out any information still in memory, and control is returned to the operating system. Do not confuse the STOP statement, which is an executable statement which terminates the program, with the END statement, which declares the end of the program

text. While a program must only have one END statement, which must be the last statement of the program, any number of STOP statements may appear in a program, and these may be the object of IF statements to conditionally terminate the program. Executing the END statement ("falling off" the end of the program) is equivalent to a STOP statement.

The following are examples of the STOP statement:

```
STOP
IF OPERATION%>15 THEN PRINT "Error!" : STOP
```

4.13. RANDOMIZE statement

```
RANDOMIZE [<expression>]
```

The RANDOMIZE statement initialises the "seed" of the pseudorandom number generation which is accessed by the RND function. If no <expression> is specified, or the value of the <expression> is zero, the generator will be seeded from a value computed from the random contents of unused memory following the program. This will normally result in a program which behaves differently each time it is called, which is generally what is desired for game programs. If a nonzero <expression> is supplied, it is used as the "seed", which may be convenient when performing statistical studies where it is important to be able to reproduce the exact sequence of pseudorandom numbers at a later time. The <expression> given to the RANDOMIZE statement is rounded to Integer before use if Real. In any case, the low two bits of the Integer are ignored. Hence, two values must have different integral values and different quotients when divided by four to result in different pseudorandom sequences.

QBASIC's RANDOMIZE statement differs from that of CBASIC in that CBASIC does not allow the <expression> to be specified, and always seeds the generator based on the typing time of the last INPUT statement. Note that in CBASIC the RANDOMIZE statement may not be used unless at least one INPUT has been done. There is no such requirement in QBASIC. Since the seed may be specified, if a hardware random number generator is available, it can be used to seed the pseudorandom generator in QBASIC.

4.14. DIM statement

```
DIM <variable>(<expression>,...),...
```

The DIM statement declares and allocates space for subscripted variables. Subscripted variables must be created by executing a DIM statement before they can be used in other statements. The <variable> name is the name of the variable to be created, and the <expression>s specify the maximum value each subscript can have. The minimum subscript value is always zero. For example, the following statement:

```
DIM I%(10), A(2,2), S$(100)
```

declares an Integer subscripted variable, I%, which may be referenced as I%(0) through I%(10), a Real subscripted variable with 9 elements A(0,0) through A(2,2), and a String subscripted variable, S$, with 101

elements S$(0) through S$(100).

Note that the DIM statement is an EXECUTABLE statement; it is not a declaration. Hence, the DIM statement must be executed before the variable it declares is first used in the program. If a DIM statement is performed on a variable which had been previously created with a DIM statement, the storage previously allocated will be released, and new storage will be assigned. This may be used to alter subscript limits, or to declare the variable with a different number of subscripts. Note, however, that all data in the variable before it is redimensioned will be lost, and the new variable will be filled with zeroes if Real or Integer, and null strings if String.

It is not necessary to set the contents of a String subscripted variable to the null string ("") before reinitialising it with a DIM statement; the library subroutines will automatically recover all the string space which it was using. This is in contrast to CBASIC, which fails to recover the space.

Note that each element in a String subscripted variable may have any desired length. Storage is assigned when a value is given to the element, so no space is used unless an element is actually assigned a value.

There is no limit on the number of subscripts which a subscripted variable may have, nor on the number of elements in such a variable other than that imposed by the amount of memory available in the machine running the program.

The <expression>s used as subscript bounds must be either Integer or Real. If Real, they are rounded to Integer. The value of each <expression> must be non-negative. There is no other restriction on the expression; a reference to the array may be used in the list if it makes sense.

The following illustrate the DIM statement:

```
DIM METER.READING(10,2)
DIM A$(I%+4), B$(J%)
DIM A(A(A(0)))
```

4.15. CHAIN statement

```
CHAIN <expression>
```

The CHAIN statement allows a QBASIC program to pass control to another program. That program may be a QBASIC program, a program in any other language, or one of the Marinchip utility programs.

The <expression> is evaluated, and must be of type String or an error will occur. The value of the expression should be the command to call the program to be CHAINed to, including any parameters which may be desired on the call line. In other words, the expression should be a string identical to what would be typed on the console to call the program from operating system command mode.

Before transferring to the new program, all files are closed, and the

QBASIC program is terminated exactly as if a STOP statement were executed. Unlike CBASIC, the contents of the print buffer are not lost when a CHAIN is done.

If the program being CHAINed to is a QBASIC program, variables may be passed on to it by using the COMMON statement, described below. Variable values may not be passed in COMMON when CHAINing to non-QBASIC programs, but they can often be passed as parameters on the command calling the program.

The contents of DATA statements are not preserved when CHAINing to another QBASIC program; this is different from CBASIC.

The following illustrate uses of the CHAIN statement:

```
CHAIN "FILEUPD"
CHAIN "ACCTWORK:PROGS/PR005 OFILE=IFILE"
CHAIN "ASM CWORK.REL=CWORK"
CHAIN "EDIT "+SOURCE.FILE.NAME$
```

Note that in every case, execution of a CHAIN statement is exactly identical to executing a STOP statement, then typing the command from the CHAIN statement on the console. However, a CHAIN statement must be used if COMMON variables are to be passed on to a QBASIC program being chained to.

4.16. COMMON statement

```
COMMON <variable>,...
```

The COMMON statement allows one or more variables to be passed from one QBASIC program to another when the CHAIN statement is used to pass control between QBASIC programs.

If a program has one or more COMMON statements, they must be grouped at the start of the program, immediately after any ENTRY or EXTERNAL statements.

All programs which CHAIN from one to another must have compatible COMMON statements, or none at all. COMMON statements are compatible if they contain the same number of variables, variable types agree on the statements, and variables occur in the same order. Variable names need not be the same. The following two COMMON statements are compatible:

```
COMMON I%, J, S$, T%
COMMON INDEX%, VALUE, CODENAME$, LOCATION%
```

The following pairs of COMMON statements are INCOMPATIBLE:

```
COMMON I%, J%
COMMON I%        ( Different no. of variables )

COMMON I%, R
COMMON I, R%     ( Types don't match )
```

Subscripted variables may be passed in COMMON. They must be

identified in the COMMON statement by enclosing a number in parentheses following the variable name. For example:

```
COMMON I%, A(1)
```

declares a simple variable I% and a Real subscripted variable called A to be in common. In QBASIC, the number in parentheses has no meaning, but for compatibility with CBASIC, it should be the number of subscripts which will be used with the variable. The initial program must declare the subscripted variable in COMMON, then create it with a DIM statement. Once created, the variable will be passed on in COMMON to all subsequent programs. If a program which is the object of a CHAIN performs a DIM on the variable in COMMON, the values passed in COMMON will be discarded and the variable will be reallocated with the new size. That new variable will then be passed on in COMMON to any subsequent programs.

Variables in COMMON are actually passed between programs by writing them out into the file TEMP2$ before exiting the CHAINing program, then reading them back at the start of the program CHAINed to. Hence, TEMP2$ must be present when QBASIC program with COMMON variables executes a CHAIN statement. Under the Network Operating System, this file will be automatically created in the user's working directory. Under the Disc Executive, however, the file must be created before the program is run, with sufficient size to hold all variables in COMMON. Since there is normally a TEMP2$ file on the standard Disc Executive system disc, if QBASIC programs are run with the system disc in place, COMMON will work without any special effort by the user.

5. Predefined functions

This chapter describes QBASIC's predefined functions. A function takes zero or more ARGUMENTS, takes some action based on their values, and returns a RESULT. Functions are used within expressions, and may be combined with other values using operators.

Functions will be described below based on the type of value they return as their result. Note that a function may take, for example, a String as an argument and return an Integer as a result. If a function returns a String result, it is called a "string function", regardless of the type of its argument(s).

Functions are called by referencing the function name, with the arguments following in parentheses. Arguments are separated by commas. Arguments must agree in type with those expected by the function, except that Real and Integer values will be interconverted when required.

In the descriptions below, argument types will be indicated by the sample arguments given. Real arguments will be represented by X, Y, Z, etc., Integer arguments by I%, J%, K%, etc., and String arguments by A$, B$, C$, etc. Actual arguments used in a function call may, of course, be arbitrarily complex expressions as long as they have the required type. If an argument is indicated as being Real, an Integer may be used, and will be converted to Real. If an argument is indicated as being Integer, and a Real is used, it will be rounded to an Integer. Supplying a String argument where a numeric argument (Real or Integer) is expected, or vice versa, will cause an error.

The following sections describe all of QBASIC's general purpose functions. There are several other functions in QBASIC which are used for specific purposes in the Input and Output mechanisms and for hardware interface. Those functions are described in the appropriate sections; they are called according to the same rules described here.

5.1. Numeric valued functions

The functions described in this section return numeric values. If the function name ends in "%", the value returned will always be Integer. Other functions may be Real or Integer; where the type is not specified in the descriptions, it is Real.

NOTE: EXP, LOG, and the trigonometric functions are calculated in single precision, to an accuracy of only about six decimal digits. This is also true of the exponentiation operator (^) when used with Real operands.

5.1.1. ABS(X) - Absolute value

ABS returns the absolute value of its argument. If X is greater than or equal to zero, X is returned. If X is negative, -X is returned.

5.1.2. ACOS(X) - Arccosine

ACOS returns the Arccosine, in radians, of its argument X. The value

returned is Real. This function is unique to QBASIC; CBASIC does not support it.

5.1.3. ADRS(X) - Address of variable

ADRS returns the actual memory address where the named variable is stored. ADRS must be called with a variable, not an expression. ADRS, unlike all other functions, may take an Integer, Real, or String argument. For an Integer or a Real it returns the address of the first byte of the number. For a String, it returns the address of a word which contains either (1) the address of the "string buffer" or (2) a zero if the string is null. A string buffer consists of an initial word containing the length of the string, and additional words containing the string text. Regardless of the argument type, the value returned is an Integer.

For more information about the ADRS function, see the chapter on hardware and machine language interface later in this manual. This function is unique to QBASIC; CBASIC does not support it.

5.1.4. ASC(A$) - ASCII code

ASC returns the ASCII code for the first character in the argument string. If A$ is the null string, an error will occur. The value returned is an Integer.

5.1.5. ASIN(X) - Arcsine

ASIN returns the Arcsine, in radians, of its argument X. The value returned is Real. This function is unique to QBASIC; CBASIC does not support it.

5.1.6. ATN(X) or ATAN(X) - Arctangent

ATN (or ATAN) returns the arctangent, in radians, of its argument X. The value returned is Real.

5.1.7. COS(X) - Cosine

COS returns the Cosine of the argument X, which must represent an angle in radians. The value returned is Real.

5.1.8. COT(X) - Cotangent

COT returns the Cotangent of its argument X. X is assumed to represent an angle in radians. The value returned is Real. This function is unique to QBASIC; CBASIC does not support it.

5.1.9. CSC(X) - Cosecant

CSC returns the Cosecant of its argument X. X is assumed to represent an angle in radians. The value returned is Real. This function is unique to QBASIC; CBASIC does not support it.

5.1.10. DATETIME(I%) - Date and time

DATETIME allows a program to read the date and time, then return

components of it to the program as Integers. If running under the Disc Executive, DATETIME will work only if the CLK-24 clock/calendar board is installed in the system. Under the Network Operating System, DATETIME will always work as long as the user has set the time in the system (or has a CLK-24 board). To insure that the time does not change while being returned by calls on DATETIME, the time is first read and stored by calling DATETIME with an argument of zero:

```
STAT%=DATETIME(0)
```

The result, stored here in STAT%, will be -1 if a valid time was read and stored, and 0 if the time was not available (no clock board under Disc Executive or time not set under NOS). Once this call is made, the time will stay the same until another call on DATETIME with an argument of zero is made.

Once the time has been read, DATETIME may be called with nonzero arguments to return the numeric components of the date and time. These arguments return the following information:

1	Year (for example 1987)
2	Month (1 to 12)
3	Day (1 to 31)
4	Hour (0 to 23)
5	Minute (0 to 59)
6	Second (0 to 59)
7	Day of week (0=Sunday, 1=Monday, ... 6=Saturday)

Any nonzero argument other than these will return zero and be otherwise ignored.

5.1.11. EXP(X) - Exponential

EXP returns the value of "e" (the base of the natural logarithms, approximately 2.718281828) raised to the power X. The value returned is Real.

5.1.12. FLOAT(I%) - Convert to Real

FLOAT converts its argument to a Real. The argument is expected to be an Integer. If the argument is a Real, FLOAT first rounds it to an Integer, then re-converts it to Real. Therefore FLOAT(REAL) will round REAL to the nearest Integer, provided the absolute value is no greater than 32767.

5.1.13. FRE - Total free space available

FRE returns a Real number equal to the total number of bytes of free space left to the program. As a program runs, this number will vary as arrays and strings are allocated and released and files are opened and closed. A program can use the result of FRE to take action before being terminated due to running out of memory. Note that FRE tells the total free space available; due to fragmentation of storage the largest available block may be less than this size. See MFRE below for information on how to obtain that value.

FRE takes no argument. No argument list should be supplied.

5.1.14. INT(X) - Integer part of Real

INT returns the whole part of its Real argument; that is, it truncates any decimal places in the argument. The result returned is Real.

5.1.15. INT%(X) - Convert to Integer

INT% converts its Real argument to Integer. Note the difference between INT and INT%: INT% actually returns an Integer, and will generate invalid results if the argument is outside the range -32768 to +32767. INT simply truncates the fractional part from a Real, and may be used on numbers of any magnitude.

5.1.16. LEN(A$) - Length of string

LEN returns the length of its string argument. If the argument is the null string, zero is returned. The value returned is an Integer. Unlike the corresponding function in CBASIC, LEN returns the actual length of its argument, including leading blanks.

5.1.17. LOG(X) - Natural logarithm

LOG returns the natural logarithm of its argument. The value returned is a Real. Note that the common log (log to the base 10) can be obtained by dividing the value returned by LOG by LOG(10.0).

5.1.18. MATCH(A$,B$,I%) - Search for pattern in string

MATCH searches string B$, starting at character position I%, looking for the first occurrence of the pattern A$. The value returned is an Integer. If the pattern A$ is found in string B$, the value returned will be the character position that the occurrence of A$ starts in B$. If the pattern is not found, zero will be returned by MATCH.

I% specifies the character position in B$ where the search should start. If I% is negative or zero, an error will occur. If I% is greater than the length of B$, MATCH will always return zero. To start at the beginning of B$, I% should be 1.

The pattern in A$ is made up of text characters and/or "wild card" pattern matching characters. Text characters must match exactly. "Wild card" characters match classes of characters, as described below:

- # Matches any digit (0 - 9).
- ! Matches any letter (upper or lower case).
- ? Matches any character.

The pattern matching characters may be "forced" to be tested as text characters by preceding them with a backslash (\). For example, to search for the string "Gadzooks!" in variable INTXT$, the following call might be used:

```
FPOS%=MATCH("Gadzooks\!",INTXT$,1)
```

Comparison of letters by MATCH is case sensitive, that is, a lower case letter will match only a lower case letter. MATCH may be made to perform a case-insensitive match by converting both its arguments to the same case before calling it, as in:

```
F%=MATCH(UCASE$(A$),UCASE$(B$),1)
```

5.1.19. MFRE - Largest memory block available

MFRE returns a Real equal to the number of bytes in the largest block left in free space. Like FRE, described above, it may be used by a program which wishes to take some special action to avoid being terminated due to running out of memory. MFRE, as opposed to FRE, is useful when a program is about to DIM a large array or create a very long string and wishes to make sure that a large enough block exists to hold the new variable. Since QBASIC appends control storage to strings and arrays, the user must allow extra space (say 15%) for this storage when testing the size of the variable against the value returned by MFRE.

MFRE takes no argument. No argument list should be supplied.

5.1.20. RND - Pseudorandom number

RND returns a uniformly distributed pseudorandom Real number between 0 and 1. Each number returned depends on the last number generated. The generator is initially "seeded" by the RANDOMIZE statement. See the description of the RANDOMIZE statement for more information on initialising the generator for the RND function.

The RND function takes no argument. No argument list should be given following it.

5.1.21. SADD(A$) - String address

SADD returns an Integer value which is the actual memory address of the buffer containing the supplied String argument. If the argument is the null string, the contents of the address given will be a zero, indicating zero length.

Note the differences between SADD and ADRS: SADD can take any string expression as an argument, while ADRS takes only a variable, which can be of any type. SADD gives the address of the string buffer, while ADRS gives the address of a variable, which in turn - if a String variable - contains either the address of a string buffer or zero.

SADD is provided mainly because it exists in CBASIC. Though it can be used in conjunction with the LEN and PEEK functions in dealing with string data at the machine language level, there are better ways of doing such things, as the chapter "Hardware and machine language interface" shows. SADD must NEVER be used with POKE to modify a String.

5.1.22. SEC(X) - Secant

SEC returns the Secant of its argument X, which is taken as an angle in radians. The value returned is Real. This function is unique to

QBASIC; CBASIC does not support it.

5.1.23. SGN(X) - Sign

SGN returns an Integer value representing the sign of its argument. The argument to SGN may be either Integer or Real. If the argument is 0, SGN returns 0. If the argument is negative, SGN returns -1. If the argument is positive, SGN returns +1.

5.1.24. SIN(X) - Sine

SIN returns the Sine of its argument, which is assumed to be expressed in radians. The value returned is Real.

5.1.25. SQR(X) - Square root

SQR returns the square root of its argument. If the argument is negative, its absolute value will be taken and its square root will be returned. SQR always returns a Real value.

5.1.26. TAN(X) - Tangent

TAN returns the Tangent of the argument, which is assumed to be an angle expressed in radians. The value returned is Real.

5.1.27. VAL(A$) - Value

VAL scans its string argument as a number, and returns a Real value equal to the value of the number found. Characters are scanned and converted until either the end of the string is encountered or an invalid character is found. Leading spaces are ignored in the string. If the string does not begin with a valid number, zero is returned. The rules for numbers scanned by VAL are identical to those for Real numbers read by the INPUT statement. VAL does not recognize hexadecimal input.

Note that the value returned by VAL is always Real, even if the string argument specifies a value the compiler would treat as an Integer (for example "14").

5.2. String valued functions

The following functions return String values.

5.2.1. CHR$(I%) - Character from ASCII code

CHR$ returns a one character string. The character making up the string is the character with the ASCII code of the argument I%. If a Real argument is supplied, it will be rounded to Integer.

CHR$ is often used to place ASCII control characters in strings. For example, CHR$(7) is a one character string consisting of the ASCII BEL code. CHR$ is also frequently used in hardware interface code where it is desired to convert a character received as an Integer into a string for use within a program.

5.2.2. COMMAND$ - Command string

COMMAND$, which takes no arguments, returns the command line used to call the currently executing program. The command line returned depends upon which operating system is being used. Under the Network Operating System, the entire command line is returned. For example, if a program were called:

```
PRIMES 1,1000 FASTMODE
```

The following program:

```
PRINT "<";COMMAND$;">"
```

would print:

```
<PRIMES 1,1000 FASTMODE>
```

Under the Disc Executive, the command string returned starts with the first character following the program name. If the same program were run under the Disc Executive, the output would be:

```
< 1,1000 FASTMODE>
```

Because a space will always precede the command string under the Disc Executive, programs may be easily written which will run under both systems by simply discarding all text in the COMMAND$ string before the first space, then examining the balance of the string.

Unlike CBASIC, if a CHAIN is done to another program, that program will "see" its own call line (from the CHAIN statement) rather than the line used to call the original program.

5.2.3. LEFT$(A$,I%) - Left part of string

LEFT$ returns the leftmost I% characters of string A$. If A$ is the null string, or I% is zero, the null string will be returned. If I% is greater than the length of A$, the entire string A$ will be returned (but it WILL NOT be padded by blanks to length I%, beware!). If I% is negative, an error will occur.

5.2.4. MID$(A$,I%,J%) - Extract substring

MID$ extracts the substring starting at the I%'th character of A$, J% characters long, and returns it. If I% is greater than the length of A$, the null string will be returned. If there are fewer than J% characters between the I%'th character and the end of A$, the rest of A$ starting at the I%'th character will be returned. The first character of A$ is number 1. If J% is zero, the null string will be returned. If either I% or J% is negative an error will occur.

Note that MID$ can be used to truncate characters from the start of a string by specifying a J% value much larger than the string length. For example:

```
B$=MID$(A$,2,2000)
```

sets B$ to all characters in A$ after the first one, unless A$ is an extraordinarily long string.

5.2.5. OVERLAY$(A$,B$,I%) - Overlay string

OVERLAY$ returns a string which is equal to B$ except that characters starting at position I% are replaced (overlaid) with characters from A$. (Compare the order of arguments in the MATCH function) In most cases this is just a much faster way of getting the effect of

```
LEFT$(B$,I%-1)+A$+RIGHT$(B$,LEN(B$)-(LEN(A$)+I%-1))
```

For example, in the program fragment:

```
OLD$="ABCDEFG"
NEW$=OVERLAY$("!!!",OLD$,2)
```

NEW$ is set to "A!!!EFG".

The result always includes all of string A$ and has a length of at least I%-1+LEN(A$). If the length of string B$ is less than that, then the result is longer than B$; otherwise it is the same length.

If the length of B$ is less than I%-1, OVERLAY$ will add blanks and then append A$. Therefore, the following expression will produce a string of twelve blanks:

```
OVERLAY$("","",13)
```

If I% is negative or zero, a value of one is used instead.

5.2.6. RIGHT$(A$,I%) - Right part of string

RIGHT$ returns the rightmost I% characters of string A$. If A$ is the null string or I% is zero, the null string is returned. If I% is greater than the length of A$, all of A$ will be returned (but WILL NOT be padded with blanks on the left).

5.2.7. STR$(X) - String representation of number

STR$ returns a string containing the edited representation of the value of the Real argument X. The format of the result returned by STR$ is the same as that used by a PRINT statement used to print the same value, with one exception: STR$ does not edit a positive number with a leading blank as PRINT does (this is compatible with CBASIC). If the argument is an Integer, it will be converted to Real. STR$ is normally used by programs which wish to perform their own formatting of numeric information.

5.2.8. UCASE$(A$) - Upper case.

UCASE$ returns a string equal to its argument but with all lower case letters converted to upper case. All characters other than lower case letters are unchanged. UCASE$ is especially useful when performing a string comparison where case is not to be significant. Converting both strings with UCASE$ before the comparison will achieve the desired goal.

6. User defined functions

QBASIC allows the user to define functions which may be called within expressions in the program. QBASIC functions are very powerful, and permit complex sequences of code to be called repeatedly without replicating them throughout the program.

A user defined function must be DECLARED. A function declaration specifies the FUNCTION NAME by which the function will be called, names DUMMY ARGUMENTS to stand for the arguments which the function will accept, and then specifies the calculations to be performed when the function is INVOKED. A function is invoked by being used in an expression.

All functions in QBASIC return a value. The value returned by a user defined function may be Integer, Real, or String, with the function type indicated by the last character of the name, following the QBASIC naming conventions.

All user defined function names must begin with the characters "FN". Following the "FN" may be any sequence of letters, numbers, and periods desired by the user. Only the first 31 characters including the "FN" are significant. The following are examples of valid user defined function names:

```
FNA
FN4301B
FN.PACK.USER.NAME$
FNORD$
FN.DEVICE.INDEX%
FNJ%
FN...THIS.CAN...GET..SILLY....%
```

Note that Real function names end in no special character, String function names end in a dollar sign, and Integer function names end in a percent sign.

There are two types of user defined functions: single line functions and multiple line functions. The declarations of these functions differ, so they will be discussed separately below. There is no difference in the way the two kinds of functions are called.

6.1. Single line functions

Single line functions are used when the result of the function can be written as an expression in terms of the function arguments. A single line function declaration is of the form:

```
DEF <function name>[(<arg>,...)]=<expression>
```

where <function name> is the name by which the function will be called, <arg> are the dummy variable names, and <expression> is the expression in terms of the dummy argument names which gives the value of the function. The type of the dummy arguments is indicated by the last character according to the normal rules. The names used for dummy arguments have significance only within the function. They may

duplicate variable names used elsewhere in the program and no harm will be done. A function may have any number of dummy arguments, or it may have none at all. If it has no dummy arguments, the declaration should contain no parenthesised argument list, nor should a call on the function specify arguments.

The type of the arguments used to call the function must agree with the type of the dummy arguments with which it was declared. A common error is to provide an Integer argument (such as 1) where the function expects a Real (such as 1.0). This will cause the program to fail with an error code of 0109.

The following is an example of a single line function declaration and its use within a program.

```
DEF FN.HEIGHT(AZIMUTH,RANGE)=RANGE*SIN(AZIMUTH)
PRINT "Plane now at ";FN.HEIGHT(TRANR,RFRDG);" meters."
```

Single line functions may also have string results. The following string function returns the portion of its string argument A$ to the left of the first occurrence of argument B$.

```
DEF FN.LPART$(A$,B$)=LEFT$(A$,MATCH(B$,A$,1))
```

There are no restrictions on the content of the <expression> used in a single line function. QBASIC permits recursive calls on single line functions, even though CBASIC does not.

2. Multiple line functions

A multiple line function allows construction of more complex functions which may use virtually all QBASIC statements and facilities in computing their result. Multiple line functions may also be used as a subprogram facility far more powerful than that offered by GOSUB.

A multiple line function declaration consists of a function header line, the function body (the statements that make up the function), and a terminating FEND statement. The function header line is of the form:

```
DEF <function name>[(<arg>,...)]
```

where <function name> is the name by which the function will be called and <arg> are the names of the dummy arguments. The rules for function names, types, and argument types are identical to those explained above for single line functions. A multiple line function is identified by the fact that no =<expression> for its value appears on the DEF line.

Following the DEF line for the function are any number of statements which make up the function body. Any valid QBASIC statement with the exception of DEF, COMMON, END, ENTRY, EXTERNAL, and SUBPROGRAM may appear with a function body. Great care should be exercised when using the DIM statement within a function body. First of all, it is impossible to DIM a dummy argument name, except when using call by reference, as described in the next section. Second, a DIM performed within a function declares an array global to the entire program, not

local to the function as are dummy variables used within the function. As a result, functions which use the DIM statement may not be used recursively (at least without great care in programming), and they must be careful not to use a name used in the rest of the program, as the DIM statement would erase a previously existing variable.

As alluded to above, multiple line functions may be used recursively without restriction (that is, they may call themselves). Each invocation of the function will have its own dummy variables, so in QBASIC recursive calls work as intuition would suggest. Dummy variables may be used freely as local variables; that is, their values may be changed at will.

The value for a multiple line function is returned by assigning it to the function name, as declared on the DEF line. If more than one assignment to the function name occurs, the last value assigned will be returned from the function. If no value is assigned to the function name by the time the function returns to the caller, zero will be returned if the function is of type Integer or Real, and the null string will be returned if the function is of type String.

The assignment to the function name is a special kind of statement, and should not be taken to imply that the function name is a variable within the function body. Using the function name within an expression will NOT retrieve the last value assigned to it, but will cause a recursive call to the function!

A multiple line function returns to the line which called it by executing a RETURN statement or by "falling off" the end of the function and executing the FEND statement at the end of the function. In CBASIC executing the FEND causes an error; in QBASIC it performs a normal return. An EXIT IF statement which is on the outermost level of the function definition (i.e., not in a WHILE, FOR, or IF block within the function) can also cause a return.

The end of a multiple line function is marked by a FEND statement, which is simply written:

```
FEND
```

Since a multiple line function may contain virtually any sequence of QBASIC statements, it is hard to illustrate all the things that can be done using them. The following example is a fairly typical use of a multiple line function. The function declared below takes two String arguments, LINE$ and WORD$, and returns an Integer equal to the number of occurrences of WORD$ within LINE$.

```
DEF FN.WORD.COUNT%(LINE$,WORD$)
    I%=0    ( Occurrences found )
    K%=1    ( Offset into string for search )
    WHILE 1
        J%=MATCH(UCASE$(WORD$),UCASE$(LINE$),K%)
    EXIT IF J%=0
        I%=I%+1
        K%=J%+LEN(WORD$)
    WEND
FN.WORD.COUNT%=I%
```

```
FEND
```

6.3. Function calls

A user defined function, whether single or multiple line, may be called within any expression. The call on a function must use the same number and type of arguments as there were dummy arguments in the function declaration. Arguments used in function calls may be arbitrary expressions (of the required type); the only exception is a dummy variable that uses call by reference, as described in the next section.

The most common error in calling functions is to provide an Integer argument where a Real was expected, or vice versa. This will cause error termination with a status of 0109. Given the definition line:

```
DEF FNX(NUMBER)
```

The following calls are INCORRECT:

```
X = FNX(1)
X = FNX(I%)
I% = FNX(LEN(A$))
```

The following are CORRECT:

```
X = FNX(1.0)
X = FNX(FLOAT(I%))
I% = FNX(FLOAT(LEN(A$)))
```

Sometimes a multiple line function is used purely as a subroutine; that is, it is used to invoke a section of code rather than to compute a value. Since all functions must return some value, such a function is normally called in an assignment statement referencing a dummy variable, such as:

```
DUMMY%=FN.UPDATE.MASTER.FILE%(MFILE$,TRANSFILE$)
```

6.4. Call by reference

Function arguments, as described in the previous section, use the convention of CALL BY VALUE: when a function is invoked, the arguments are evaluated and stored in an area that is private to the particular function call. There is no way to pass a whole array as an argument (though single elements of an array can be passed), and a change in the value of a dummy argument has no effect outside the function. Both of these features can be changed by using CALL BY REFERENCE, which is similar to the handling of arguments in Fortran.

To define a dummy variable as using call by reference, place an asterisk before it in the DEF statement. If the dummy variable is a whole array, it must be followed by empty parentheses. For example, the following defines a function of a Real value, an Integer called by reference, and a String array:

```
DEF FNA(X,*I%,*A$())
```

The following function reads a specified number of lines from the console. The input is returned in a string array, the number of lines containing a question mark is returned in a variable supplied in the function call, and the function value is the number of non-blank lines read.

```
DEF FNREADLINES%(NLINES%,*LINES$(),*QUERIES%)
   DIM LINES$(NLINES%)
   NONBLANKS% = 0
   QUERIES% = 0
   FOR I%=1 TO NLINES%
      INPUT LINE LINES$(I%)
      NONBLANKS% = NONBLANKS%+SGN(LEN(LINES$(I%)))
      QUERIES% = QUERIES%+SGN(MATCH("\?",LINES$(I%),1))
   NEXT I%
   FNREADLINES% = NONBLANKS%
FEND
```

In any function call the argument list must match the dummy variable list in the DEF statement as to the types (Real, Integer, and String). In addition, for any dummy variable which uses call by reference, the corresponding argument MUST be a variable (not an expression) preceded by an asterisk. The following would be a valid call on the function defined in the example above:

```
LINESREAD% = FNREADLINES%(WANTED%+2,*SARRAY$(),*QUERIES%)
```

The first argument can be an expression because NLINES% is not called by reference. The main program does not need a DIM statement for SARRAY$ because the function takes care of it. The third argument in this example happens to have the same name as the dummy variable.

When a dummy variable is a whole array, the argument supplied in the call must also be a whole array. The following calls on FNREADLINES% are NOT VALID:

```
I% = FNREADLINES%(J%,*S$(1),*K%)
I% = FNREADLINES%(J%,A$,*K%)
```

7. Input and Output Statements

This chapter describes the basic input and output facilities of QBASIC. Statements described in this chapter allow data to be read from the user's terminal or from data tables stored within the program, and to be displayed on the terminal or sent to a printer. The next chapter will describe QBASIC's file input and output facilities. Understanding the operation of the statements in this chapter is necessary to understand the file oriented statements, since the latter are simply extensions of the statements described here.

7.1. Console and printer input/output

This group of statements permits data to be read from the user's terminal, and to be displayed either on the terminal or on the standard printer (PRINT.DEV).

7.1.1. PRINT statement

PRINT <expression> [<sep> <expression>]... [<sep>]

The PRINT statement prints the values of the <expression>s on either the terminal or the printer. Where the information is printed depends on whether a CONSOLE or LPRINTER statement was executed last (see below). The <expression>s to be printed may be Integer, Real, or String.

The formatting of data printed by the PRINT statement is controlled by the separator character, <sep>, used between multiple expressions in the statement. If a comma is used, the expression following the comma will be printed in the next column of 20 characters. Thus, output may be placed in columns by using commas between expressions. If a semicolon is used as the separator, one space will be output after numbers (Integer or Real) and no space after Strings. Note that when a number is printed, either a space or a minus sign always precedes the number.

Normally, after printing all items in the statement, the output line will be ended, and the terminal or printer will advance to the next line. If the last item in the statement is a separator, either comma or semicolon, the terminal carriage will be left extended or the end of line will not be sent to the printer. This may be used to build up output lines with multiple PRINT statements, or to print "prompts" for user input on the terminal.

If no <expression> appears on the PRINT statement (that is, all it says is "PRINT"), a blank line will be printed (or if information has already been printed on this line, it will simply advance to the next line).

Unlike CBASIC, QBASIC does not automatically check for output which is longer than the output device width and break it into two or more lines. QBASIC will simply send output to the device, and the action taken if the output is longer than the device's line length is dependent on the characteristics of the output device.

An Integer expression may be printed as five hexadecimal digits by following it by a sharp sign (#).

The following are examples of PRINT statements:

```
PRINT
PRINT A,B,C
PRINT "And the answer is";THE.ANSWER
PRINT "Quasiviable metaparameter index=";QMPI%
PRINT "Enter value for Point";I%;
PRINT V1(J%),
PRINT "Hardware status =";STAT%#
```

7.1.2. PRINT USING statement - formatted output

```
PRINT USING <format string> ; <expression>,...
```

The PRINT USING statement permits output to be written in which the user has complete control over the presentation of the information written. The PRINT USING statement takes a String expression called the FORMAT STRING, which describes how the output is to appear, and edits an output line by inserting the data <expression>(s) into the output line edited as described.

The format string may contain both "literal data" which is simply copied to the output unchanged and "data fields" which are replaced by edited representations of the <expression>s to be output. Any character not described below as a data field character will simply be copied to the output unchanged.

7.1.2.1. ! - Single character string field

The "!" character in a format string causes the first character of the next expression to be edited in the output. For example:

```
F$="This radio receives !.!."
PRINT USING F$ ; "Frequency", "Modulation"
```

would print:

```
This radio receives F.M.
```

7.1.2.2. /.../ - Fixed length string field

Two slashes in a format string delimit a fixed length string field. Any characters between the slashes are ignored and simply serve to define the width of the field.

The string expression will be edited into the field left-justified and space filled. If the string being edited is longer than the field supplied, characters will be truncated. The following example illustrates fixed length string fields.

```
F$="The time is /......../"
PRINT USING F$ ; "10 P.M."
PRINT USING F$ ; "too late"
PRINT USING F$ ; "fourscore and seven years ago"
```

This program would print:

```
The time is 10 P.M.
The time is too late
The time is fourscore
```

7.1.2.3. & - Variable length string field

An ampersand (&) specifies a variable length string field. The entire contents of a string expression will be edited into the output line. The following is an example of use of variable length strings in output.

```
F$="I don't have enough & to do it."
PRINT USING F$; "time"
PRINT USING F$; "money"
PRINT USING F$; "good reasons"
```

This program would print:

```
I don't have enough time to do it.
I don't have enough money to do it.
I don't have enough good reasons to do it.
```

7.1.2.4. Numeric fields

Numeric fields are made up of sharp signs (#) and other characters which select editing options for the number printed in the field.

The simplest numeric field consists of one or more sharp signs, optionally with a decimal point included in the field. Numbers printed within the field will be rounded to the number of decimal places indicated by the field and printed with the decimal aligned as specified by the field. If the number printed is negative, a minus sign will be printed in the field to the left of the most significant digit.

The following is an example of simple numeric fields:

```
F$="####  ####.##  ####.######"
A=1
PRINT USING F$; A,A,A
A=2.71828183
PRINT USING F$; A,A,A
A=-A*10
PRINT USING F$; A,A,A
```

This program will print:

```
   1     1.00     1.000000
   3     2.72     2.718282
 -27   -27.18   -27.182818
```

If one or more commas appear within a numeric field, the number will be printed with commas separating each group of three digits before the decimal point. The placement and number of commas in the field has no effect on the way the number is printed; it will always be

printed the conventional way. Note that each comma included in the field reserves one place in the number, so if the field is to correspond with the number being printed, the commas should be included where they will appear in the number.

The following is an example of numeric fields with commas:

```
F$="########  ##,###,###.##  #,########"
A=1.234567
FOR I%=1 TO 7
 PRINT USING F$; A,A,A
 A=A*10
NEXT I%
```

This program will print:

```
       1          1.23          1
      12         12.35         12
     123        123.46        123
    1235       1,234.57      1,235
   12346      12,345.67     12,346
  123457     123,456.70    123,457
 1234567   1,234,567.00  1,234,567
```

A numeric field may be asterisk filled (check protection) by beginning it with two asterisks (**). A field will be printed with a floating dollar sign if it begins with two dollar signs. The asterisk fill and floating dollar sign options may be combined with sharp signs, commas, and a decimal point in a numeric field, but may not be used together. If the number entirely fills the field, no asterisk or dollar sign will be printed. If the number is negative, the dollar sign will be printed before the - sign; this is different from CBASIC and may be changed in a future release. The following is an example of asterisk fill and floating dollar sign:

```
F$="$$###,###.##    **********"
A=12.36
PRINT USING F$; A,A
A=7353.32
PRINT USING F$; A,A
A=-A
PRINT USING F$; A,A
```

This program will print:

```
     $12.36     ********12
  $7,353.32     ******7353
 $-7,353.32     *****-7353
```

If a minus sign is appended to the end of a numeric field, the number will be printed with a trailing minus sign if negative. If the number is positive, a blank will be printed following the number. The following is an example of trailing minus signs:

```
F$="####.##  ###.##-"
A=123.456
PRINT USING F$; A,A
```

```
A=-A
PRINT USING F$; A,A
```

This program will print:

```
 123.46  123.46
-123.46  123.46-
```

If a number to be printed within a numeric field is sufficiently large that it cannot be printed within the designated field without truncating digits before the decimal point, it will be replaced by a field of all asterisks (*). For example:

```
PRINT USING "###"; 12345.6
```

would print:

```
***
```

7.1.2.5. Forcing field control characters

Any of the field control characters mentioned in the above sections may be printed as literal text data by preceding it with a backslash (\). The backslash causes the next character to be printed as a literal character regardless of what it is. Note that to print a backslash, it must be forced, so two backslashes must be used.

For example:

```
F$="Gosh\!  Jar \### has a ##"" & in it."
PRINT USING F$; 18, 14, "bullfrog"
```

would print:

```
Gosh!  Jar #18 has a 14" bullfrog in it.
```

Note that the quote character had to be forced into the string constant, but since it is not a field character, did not have to be preceded by a backslash. The exclamation point and sharp sign did have to be forced, as otherwise they would be treated as field characters.

7.1.2.6. Matching of expressions and fields

Processing of the format string is controlled by the type and number of <expression>s to be edited into it. If there are more <expression>s than data fields in the format string, the format string will be reused from the start. Hence, many items may be printed with the same format without repeating it for each item. For example:

```
F$="Point ##=####.## "
PRINT USING F$; 1,1.23,2,-3.01,3,22.87
```

will print:

```
Point  1=   1.23 Point  2=  -3.01 Point  3=  22.87
```

Each <expression> is evaluated, then the format string is searched from the current position for a data field of the corresponding type. For example, if the <expression> has a numeric value (Integer or String), the format string will be scanned for the next numeric field. If a string field is encountered first, it will simply be output as literal characters. The same holds true if the expression has String value: numeric fields will be output as literals as the scanner searches for a string data field. If no data field of the right type is found in the format string, an error will occur. This somewhat bizarre feature has been included for compatibility with CBASIC. When all the <expression>s have been edited, any remaining characters in the format string will be printed literally.

7.1.3. CONSOLE statement

```
CONSOLE
```

The CONSOLE statement directs subsequent output from the PRINT statement to the user's terminal. This mode is in effect initially when a program is called, so the CONSOLE statement is necessary only to restore output to the terminal after having used the LPRINTER statement (see below).

7.1.4. LPRINTER statement

```
LPRINTER [WIDTH <expression>]
```

The LPRINTER statement directs all subsequent output from the PRINT statement to the standard printer device, PRINT.DEV. Output may be restored to the user's terminal at a later time by the CONSOLE statement (see above). If PRINT.DEV can not be assigned (for instance, if another user is using it under NOS/MT), the program is terminated in error.

The WIDTH clause on the LPRINTER statement is used by CBASIC to specify the line width of the printer. As discussed above in the description of the PRINT statement, QBASIC does not wrap around output based on the line width, so the WIDTH specification, if present, is ignored.

7.1.5. Console/printer output functions

The following two functions allow the column pointer used by the PRINT statement to be examined and changed.

7.1.5.1. POS function

```
POS
```

The POS function returns the current value of the column pointer used by PRINT. It can be used anywhere in a program, including a PRINT output list. The leftmost column on the page is column 1, and the number returned is the column number where the next item placed in the buffer by PRINT will start. For example, in the following program fragment:

```
PRINT "One small step for a man, ";
```

```
A=POS
PRINT "one giant leap for mankind."
B=POS
```

The variable A would be set to 27 and B would be set to 1.

7.1.5.2. TAB function

```
TAB(<expression>)
```

The TAB function may appear only within a PRINT statement. It sets the column pointer used by PRINT to the value of the <expression> given as the parameter. The <expression> must be either Integer or Real. If Real, it is rounded to an Integer before being used. If the column pointer is already at a column higher than that specified by <expression>, the line will be printed, and the TAB will be done on a new line. The following are examples of the TAB function:

```
PRINT "Qty";TAB(10);"Name";TAB(40);"Price"
PRINT TAB(INDENT.COUNT%);TEXT$
PRINT TAB(45+SIN(J)*40);"*"
```

The last example illustrates how plotting may be done using the TAB function.

7.1.6. INPUT statement

```
INPUT [<prompt> ;] <variable>,...
INPUT [<prompt> ;] LINE <variable>
```

The INPUT statement reads data entered by the user on the terminal and stores it into variables named on the INPUT statement. If no <prompt> is specified, the user will be prompted for input with a question mark. If <prompt> is specified, it must be a String CONSTANT. Note that an expression cannot be used for a <prompt>, nor may the <prompt> be Integer or Real (this can be achieved by preceding the INPUT with a PRINT statement with a trailing semicolon, then using the null string as the <prompt> in the INPUT statement). After the <prompt> QBASIC will print one blank; even if the <prompt> is null, the user will see a prompt of a single blank. The extra blank, which is included for GBASIC compatibility, can be suppressed; see the section "External variables in the library".

In the first form of the INPUT statement, values entered on the terminal will be scanned and assigned to variables in the order named in the INPUT statement. Items entered on the terminal must be separated by commas. If the value entered is an Integer and the variable is a Real, it will be converted to Real before being stored in the variable. If the variable is an Integer, the value entered on the terminal MUST NOT contain a decimal point or an exponent. Leading plus signs may not be typed on numbers used with INPUT. String variables read with INPUT may simply be words separated by commas. Strings may be enclosed in quotes, as string constants in QBASIC programs are written, and this is the only way for a normal INPUT statement to read a string with leading blanks and/or embedded commas (but see the INPUT LINE statement below for an alternate approach).

The number of values entered on the terminal must be the same as the number of variables on the INPUT statement, and values typed must be separated by commas. If input disagrees with the variable list or is badly formed, an error message will appear and the information must be reentered correctly.

The following are examples for the first form of the INPUT statement:

```
INPUT A
INPUT A,I%,V(T%+1)
INPUT "Enter part number:"; PART.NO
INPUT "Last name, first name, age?";LNAME$, FNAME$, AGE%
```

If the LINE specification appears on the INPUT statement, only one variable may be specified. This variable must be a String variable. The entire line of input typed on the terminal will be stored in the String variable. No format conversion will be done; if quotes are typed on the console, they will be placed in the String variable. The LINE form of the INPUT statement is used when entering information which will be examined by the program and for which QBASIC's normal input scanning is undesirable or inadequate, and also is frequently used when reading String data which might contain characters to which the normal INPUT scanner is sensitive, such as commas. If the user simply responds to an INPUT LINE statement with a carriage return, the variable will be assigned the null string.

The following are examples of the LINE form of the INPUT statement:

```
INPUT LINE A$
INPUT "Enter City, State, Zip code: "; LINE ADD3$
```

7.2. Direct console input functions

Two functions in QBASIC permit direct input from the user's terminal. These functions allow a QBASIC program to read characters from the terminal bypassing both QBASIC's and the operating system's normal input scanning. These functions are similar to those provided by CBASIC; differences will be noted.

7.2.1. CONSTAT% function

The CONSTAT% function waits until a character is typed, remembers its value, then returns an Integer value of -1.

In CBASIC, CONSTAT% always returns immediately, returning 0 if no character is available and -1 if a character has been typed. QBASIC's implementation is equivalent for all programs which use CONSTAT% in a wait loop before processing the character. Note that in a multi-user system a program which is waiting on CONSTAT% will not tie up the whole computer.

In a single user system it may be desirable to make CONSTAT% function as it does in CBASIC. For example, a program making a long computation might check CONSTAT% occasionally to see if the user wants to interrupt. The section "EXTERNAL variables in the library" tells how to do this.

7.2.2. CONCHAR% function

The CONCHAR% function returns one character from the user's terminal. The value returned by CONCHAR% is an Integer equal to the ASCII code for the character. The character is echoed to the console when typed, unless it is a control character (ASCII code less than 020).

Unlike CBASIC, QBASIC actually waits for a character to be typed when CONCHAR% is called, so it is not necessary to test CONSTAT% before calling CONCHAR%. If CONSTAT% has been called, CONCHAR% returns the value that CONSTAT% saved.

When a character is entered using CONCHAR% or CONSTAT%, the standard system actions on control characters, such as Control-X and Control-C, are suppressed.

The following program fragment, coded to work in both QBASIC and CBASIC, illustrates direct console input.

```
        PRINT "Enter data: ";
        PW$=""
    110 WHILE NOT CONSTAT%
        WEND
        C%=CONCHAR%
        IF C%=13 THEN 200   REMARK  CARRIAGE RETURN
        IF C%=127 THEN \
         PRINT CHR$(8);" ";CHR$(8); \
        ELSE \
         PW$=PW$+CHR$(C%)
        GO TO 110
    200
```

This program uses the direct console input feature to read data from the user, bypassing the normal system action on control characters, and treating the ASCII DEL character as a backspace.

When a character is input through CONSTAT%, it is echoed on the terminal unless it is a control character (ASCII value less than 32 decimal). This is compatible with some releases of CBASIC; compare the console input code in the Osborne commercial packages as originally published. If you need compatibility with other CBASIC releases which do not echo, or you just don't want echoing, see the section "EXTERNAL variables in the library."

7.3. Program data input statements

The READ, DATA, and RESTORE statements allow QBASIC programs to read data stored within the program itself. This feature is useful for initialising program variables, or to provide tables commonly searched by a program.

7.3.1. DATA statement

```
DATA <constant>,...
```

The DATA statement is used to declare the data which is read by the READ statement. The <constant>s used in the DATA statement may be

Integer, Real, or String. String constants need not be enclosed in quotes as long as they contain none of the following: blank, comma, quote, colon, backslash, or vertical bar (|).

Constants declared on the DATA statement are not checked by the compiler, but rather are stored in symbolic form and processed at execution time when the READ statement is done. Consequently, syntax errors in the DATA statement will not be detected until execution time. This is compatible with CBASIC.

If a DATA item is a String and the corresponding variable on the READ statement is an Integer or Real, the results are indeterminate. If the DATA item is numeric and the variable in the READ statement is a String, the number will be read as a string of digits, in the same way as any String not surrounded by quote marks. A value containing a decimal point or an exponent can not be read into an Integer variable; any numeric value can be read into a Real variable.

In QBASIC, unlike CBASIC, the DATA statement need not be the only statement on a line. A colon which is not inside quotes is recognised as a statement terminator, just as in other statements. The CBASIC manual explicitly states that DATA statements cannot be continued from line to line. Nevertheless, the CBASIC compiler supports continuation of DATA statements, and so does QBASIC.

As many DATA statements may be used in a program as are required to declare all the data needed. Data will be read from the statements in the order they appear in the source program. The DATA statements need not appear first in the program, nor need they appear in a block; the compiler will group the DATA together. DATA statements are not executable: if control passes through a DATA statement nothing will happen.

The contents of DATA statements are not preserved through a CHAIN operation. SUBPROGRAMs may not contain DATA statements.

The following are examples of the DATA statement:

```
DATA 1,3,12.373,-23.4,18.03E-8
DATA Mercury,Venus,Earth,Mars,Jupiter
DATA Saturn,Uranus,Pluto,Neptune
DATA "Input phase", "Sort phase", "Output, edit phase"
```

7.3.2. READ statement

```
READ <variable>,...
```

The READ statement reads information from DATA statements into variables. Information is read into variables in the order it appears in the DATA statement(s), and the pointer is moved as the data are read. There need be no correlation between the groupings of constants in DATA statements and variables in READ statements. An attempt to READ past the end of the last DATA statement will result in an error termination. The following program fragment illustrates how DATA and READ might be used to initialise tables in a program.

```
DIM COLOUR$(9), DIGIT%(9)
```

```
FOR I%=0 TO 9
 READ COLOUR$(I%),DIGIT%(I%)
NEXT I%
DATA BLACK,0,BROWN,1,RED,2,ORANGE
DATA 3,YELLOW,4,GREEN,5,BLUE,6
DATA VIOLET,7,GRAY,8,\
     WHITE,9
```

7.3.3. RESTORE statement

```
RESTORE
```

The RESTORE statement resets the pointer into the DATA in the program so that the next READ statement will start back at the first constant in the first DATA statement in the program. This allows a program to read the DATA over and over again. For example:

```
RESTORE
IF I%>=9 THEN RESTORE
```

7.4. Array input/output

Programs that deal with arrays frequently have statements that look like

```
PRINT X(1);X(2);X(3);X(4);X(5);X(6);X(7);X(8);X(9);X(10);X(11)
```

To make the program shorter and more readable, QBASIC allows a FOR - NEXT loop to be included in any I/O list. Specifically, an element of an I/O list can be of the form:

```
FOR <index>=<expression> TO <expression> [STEP <expression>]
    <sep> <i/o list> <sep> NEXT
```

The item <i/o list> can contain another FOR - NEXT list. Unlike the normal FOR loop, the FOR - NEXT in an I/O list does not allow constructions such as "NEXT I%" or "NEXT X,Y".

In a PRINT statement the spacing of output is not affected by the choice of comma or semicolon as <sep> after the FOR and the NEXT. Therefore, the following two statements are equivalent:

```
PRINT X, FOR I%=1 TO 3, Y(I%), NEXT, Z
PRINT X, FOR I%=1 TO 3; Y(I%), NEXT; Z
```

The following statement is not equivalent, as it will put just one space after each Y(I%):

```
PRINT X, FOR I%=1 TO 3, Y(I%); NEXT, Z
```

The following example, more realistic for file input than for console input, illustrates input to a two-dimensional array using nested loops:

```
INPUT DIM1%, \
   FOR I% = 1 TO DIM1%, X(I%),DIM2%, \
      FOR J% = 1 TO DIM2%, Y(I%,J%),Z(I%,J%), NEXT, \
```

NEXT

3. File input/output

This chapter describes the QBASIC statements which provide the ability to transfer information to and from files. The files used by QBASIC are maintained by the operating system under which QBASIC runs, and hence the naming conventions used for files must agree with the system under which QBASIC is used. Most examples given in this manual will use Disc Executive file names, as they are simpler. When using QBASIC under the Network Operating System, refer to the Network Operating System user guide for an explanation of file names under that system.

QBASIC files may be used without regard to the actual physical properties of the devices used to store them. QBASIC programs need not be aware of the record length or addressing of the devices used to hold files. Of course, applications must be designed not to use more storage space than is available on the system intended to run them.

QBASIC file access statements exist to open and/or create files (OPEN, CREATE, FILE, and GETFILE), to transfer data to and from files (READ, PRINT, GET, PUT), and to close and optionally delete files (CLOSE, DELETE). For use under the Network Operating System there are statements to change discs (MOUNT and DISMOUNT) and to protect files against simultaneous updates by different users (LOCK, UNLOCK). In addition, two functions, RENAME and SIZE, allow file names to be changed, and file size to be determined by a program. All of these statements and functions are supported on both the Disc Executive and the Network Operating System. Even though the Disc Executive cannot normally dynamically create or rename files, QBASIC is able to provide these functions through special interfaces to the system.

3.1. OPEN statement

```
OPEN <expression> [RECL <expression>[,LOCK]]
     AS <expression> [BUFF <expression>]
                     [RECS <expression>],...
```

The OPEN statement makes a file which already exists available for access by a QBASIC program. The first <expression> must be a String expression, and is the name of the file to be OPENed. The value of this string expression is the file name, in the format normally used by the operating system. All qualifications and defaults permitted by the operating system are allowed here.

The operating system is asked to open the named file. If the system indicates that no such file is present, a test is made whether an IF END statement has been performed on the file number following the AS expression. If so, control will be transferred to the line number designated on the IF END statement, otherwise an error will occur. See the description of the IF END statement below for more details.

Following the keyword AS is an <expression> referred to as the FILE NUMBER. This expression must have a Real or Integer value, and if Real, it is rounded to Integer before use. File numbers must be in the inclusive range from 1 to 20. Each file activated by an OPEN or CREATE statement must have a unique file number; if another file is already open with the same number, an error will occur. The file

number specified on the OPEN statement will be used in subsequent READ, PRINT, GET, PUT, CLOSE, and DELETE statements to identify the file.

The number of files that may actually be open at one time depends on the configuration of the operating system, not on QBASIC's internal tables. The limit is usually ten.

If the optional RECL clause is included in the statement, the file will be written with a fixed record length specified by the value of the <expression> following the RECL keyword. The record length expression must be Integer or Real, with Real values being rounded to Integer before use. If the RECL clause is specified, the file may be accessed either sequentially or randomly. If the RECL clause is omitted, the file may be accessed only sequentially. When reading and writing text files intended to be compatible with all other Marinchip software, the RECL clause must not be used. Files written with RECL specified may be used only within QBASIC programs.

If the option ",LOCK" is appended to the RECL specification, the file will be eligible for record locking under the Network Operating System, as described in the next chapter. This option does not by itself cause any kind of locking, but causes the records to occupy one more byte of disc space than the number given in RECL. A file written with LOCK must always be opened with the LOCK option, even if the program does not lock or unlock records. When used in the Disc Executive, the LOCK option reserves space for the lock byte, so that the resulting file can later be transferred to NOS by the CONVERT utility and used with record locking.

The optional BUFF clause makes it possible to control the amount of buffer space used by a file. Usually it is omitted, resulting in a buffer of 256 bytes. If it is included, the <expression> must be Integer or Real, with Real values rounded to Integer before use. The value of the <expression> is multiplied by 128 to give the number of bytes in the buffer. The buffer size is completely independent of the record size given in the RECL clause, if any. A large value in the BUFF statement can improve the performance of many programs by reducing the number of disc accesses. A value of 1, or any odd value, may seriously degrade performance on double density floppy discs.

The RECS specification is ignored by QBASIC and should be omitted when writing new programs. It is included in the syntax for compatibility with CBASIC programs.

The buffers used to access files are allocated when a file is OPENed and released when a file is CLOSEd. Hence, to conserve memory space files should be CLOSEd whenever they are no longer needed by a program.

The following examples illustrate the OPEN statement:

```
OPEN "INPUT.TXT" AS 1

OPEN "2/"+CUST.DATA$ AS 2

OPEN ACCT.DATABASE$ RECL 145 AS 3, "TEMP1$" AS 4
```

```
OPEN "2/WIDGIN.ZOT" AS 3 BUFF 10 RECS 128

OPEN MULTI.USER$ RECL 293,LOCK AS 11
```

8.2. CREATE statement

```
CREATE <expression> [RECL <expression>[,LOCK]]
       AS <expression> [BUFF <expression>]
                       [RECS <expression>],...
```

The CREATE statement is identical to the OPEN statement described above except that the file name need not exist before the the CREATE statement is performed. If a file already exists with the name given by the first <expression> it will be deleted. In any case, a new file with that name will be CREATEd. The meanings of the RECL, LOCK, AS, BUFF, and RECS specifications is identical to that in the OPEN statement.

IMPORTANT NOTE FOR DISC EXECUTIVE USERS!!! Since the Disc Executive preallocates files and does not permit dynamic expansion, the CREATE statement MUST specify the size of the file being CREATEd. This is done by appending a file size specification to the file name in the CREATE statement. The file size is given in terms of 128 byte blocks, exactly as used by the CREATE command in the Disc Executive itself. (If the file is created on a double density disc, the size will be rounded to the next even number so that physical sectors will not be split between files.) For example, to create a file named "NEWDATA" on drive 2, and reserve 130 blocks of storage, each 128 bytes in length, the following CREATE statement would be used:

```
CREATE "2/NEWDATA,130" AS 1
```

The file size may be provided by the program by using the string concatenation and editing facilities of QBASIC. For example, to create a file whose name is in the variable NEWFILE$ and whose length is specified by the Integer variable NFLEN%, the following statement might be used:

```
CREATE NEWFILE$+","+STR$(NFLEN%) AS 5
```

The file size needs to be specified only when creating files under the Disc Executive. Under the Network Operating System, the file size is not required, and the CREATE statement is exactly compatible with QBASIC. If a file size is specified under the Network Operating System, it will be ignored; hence programs may be constructed which will work under either system without modification.

The following are examples of the CREATE statement:

```
CREATE "MYFILE.DAT,20" AS 3
CREATE "FILE1,100" AS 2, "2/FILE2,25" AS 3
CREATE "PAYROLL/CHECKS/MAR80" RECL 150 AS 12
```

Note that the last example can be used only under the Network Operating System, and hence the file length is not specified.

8.3. FILE and GETFILE statements

```
FILE <variable>[(<expression>)],...
GETFILE <expression> [RECL <expression>[,LOCK]]
     AS <expression>,...
```

The FILE (or GETFILE) statement opens an existing file, if present. If no file with the specified name exists, a new file is created by that name, then opened. Note the difference between the FILE statement and the CREATE statement. If a FILE statement is used and the named file already exists, it is opened and data in it may be read. If a CREATE statement is used, all data in the file are lost, as the CREATE statement deletes any previously existing file with the name given.

The FILE statement is compatible with CBASIC, while the GETFILE statement is an extension unique to QBASIC. We will discuss the two forms separately.

The FILE statement supplies the file name in a String <variable>. If a parenthesised <expression> follows the file name, it specifies the record length and marks the file as fixed record length, equivalent to the RECL clause on the OPEN or CREATE statements. You will note that there isn't any specification of file number in this statement. The FILE statement overcomes this omission by assigning each file the next UNUSED file number. If a FILE statement is performed at the start of a program, the files will be assigned numbers 1, 2, 3, and so on. If some files are CLOSEd, then another FILE statement is done, the files it opens will be assigned the currently unused numbers in ascending order. One suspects that this statement is more the result of history than conscious design. Nevertheless, CBASIC has it, and so does QBASIC. In CBASIC, the <variable> may not be subscripted, nor may it be an expression (not even a constant!!!). QBASIC does not have these silly restrictions. Of course, the file name expression must be a String, and the record length expression must be Integer or Real, with Real values being rounded to Integer before use.

QBASIC offers the GETFILE statement, which takes on the form of the CREATE statement. In this form, all the comments in the description of the CREATE statement apply, except that if the file already exists it will be opened rather than deleted and re-created. In new programs, this is obviously the form to use, since it eliminates the "file number roulette" of the original CBASIC FILE statement.

When using the FILE or GETFILE statement under the Disc Executive, the file length must be specified as part of the file name as for the CREATE statement. The file length is needed only if the file must be created, so it may be omitted on files known to already exist.

The following are examples of the CBASIC form of the FILE statement:

```
FILE INRECS$
FILE FOXBAT$,BACKFIRE$(1202)
```

The following illustrate the FILE statement with extensions permitted by QBASIC, but not CBASIC:

```
FILE "NIDNUTS.INP,120", "VEEBLE,184"(J%+9)
FILE IFN$+",398"
```

The following examples demonstrate the QBASIC GETFILE statement:

```
GETFILE "2/BOGGLE,120" AS 1, "YOYO,10" RECL 36 AS 2
GETFILE MBFN$ AS MFN%
```

8.4. READ statement

The READ statement used with files has four forms depending on whether the data from the file are being read sequentially or randomly, and whether individual variables are being read, or entire records from the file are being read into String variables. Each form of the READ statement will be discussed separately below.

8.4.1. Sequential file variable READ

```
READ # <expression> ; <variable>,...
```

This form of the READ statement reads one or more variables from a sequential file. The <expression>, which must be Integer or Real (Real is rounded to Integer), specifies the file number to be read. This file number must correspond with the file number used on the OPEN, CREATE, FILE, or GETFILE statement used to activate the file. Variables read from the file may be Integer, Real, or String. The type of the <variable> should agree with the type of the data being read from the file.

As in GBASIC, a sequential READ from a file which was opened without the RECL specification reads data items one by one into <variable>s named on READ statements without regard to record boundaries. Hence, one READ statement may read part of a record or many records, depending on the number of <variable>s named and the number of data items per record in the file being read. This is different from releases of QBASIC before 2.0, which did not ignore record boundaries in stream input.

Note that this form of the READ statement can be used both with files intended for sequential access (no RECL specification) and with files for which random access is permitted (RECL specification present).

The following are examples of the sequential variable READ statement:

```
READ # 1; PART.NAME$, ON.HAND, ON.ORDER, PRICE

READ # IFILE%; KEY$, RECTEXT$
```

8.4.2. Sequential file line READ

```
READ # <expression> ; LINE <variable>
```

This form of the READ statement is equivalent to the LINE form of the INPUT statement. It reads the next record from the file number designated by the <expression> into the String <variable> named after the keyword LINE. The <variable> must be of type String.

The data placed in the <variable> will be the characters making up the record in the file being read. The end of record will be delimited by the end of record character in the file (this character will not be

placed in the variable; only data characters are transferred).

The following are examples of the sequential line READ statement:

```
READ #4; LINE PROG.TEXT$

READ # IN.FILE% ; LINE ITEXT$(I%)
```

8.4.3. Random file variable READ

```
READ # <expression>, <expression> ; <variable>,...
```

The random file variable READ statement may be used only on files activated with the RECL specification. The action of the random file variable READ is identical to that of the sequential file variable READ, except that the second <expression> after the file number selects which record is to be read from the file.

The first <expression> specifies the file number, and must correspond to a file number previously activated with the OPEN, CREATE, FILE, or GETFILE statement, specifying a record length. The second <expression> selects the record number to be read. This expression must have an Integer or Real value (Real is rounded to Integer), and must be in the inclusive range from 1 to 32767. The first record in the file is record 1.

The <variable>s named will be read from the selected record. An attempt to read beyond the end of data in the record will cause an error termination.

8.4.4. Random file line READ

```
READ # <expression>, <expression> ; LINE <variable>
```

The random file line READ statement may be used only on files activated with a RECL specification. It acts exactly like the sequential file line READ described above except that the second <expression> selects which record from the file is to be read into the named String <variable>.

The following are examples of the random file line READ statement:

```
READ # 7, 1 ; LINE FILE.TITLE$

READ # CUST.FILE%, CUST.INDEX% ; LINE CUST.NAME$
```

8.5. IF END statement

```
IF END # <expression> THEN <line number>
```

The IF END statement allows the user to specify the action to be taken when the end of a file is reached. The <expression> specifies the file number that the IF END condition is to apply to. When a READ statement attempts to read sequentially past the end of that file number, control will be transferred to the <line number> specified in the IF END statement. If no IF END condition has been specified for a file and an attempt is made to READ past the end, an error will occur.

If an IF END statement is performed on a file number before an OPEN statement is performed on that number, then if the OPEN fails to find the file in the directory control will pass to the line number named on the IF END statement.

An IF END statement may also be used on an output file. If an IF END condition is in effect, it will be triggered by a PRINT, PRINT USING, or PUT statement which attempts to write off the end of a contiguous file, or which exhausts the free space on the volume the file resides on.

Any number of IF END statements may be executed on the same file number. Only the most recent IF END statement will be effective. The IF END condition is dropped when the file is closed.

8.6. PRINT statement

The file PRINT statement is used to write information from variables into files. There are two forms of the PRINT statement, used respectively for writing files sequentially and randomly. Each form of the PRINT statement for files will be discussed separately below:

8.6.1. Sequential file PRINT

```
PRINT # <expression> ; <expression>,...
```

The sequential file PRINT statement writes the values of variables into the next record of a file. Note that this form of the PRINT statement may be used on files opened with or without the RECL clause.

The first <expression> specifies the file number, and must be Integer or Real (Real is rounded to Integer), and must correspond to a file number currently open. The <expression>s following the semicolon are the values to be written into the file. Any number of values may be given, as long as their total length when printed does not exceed the record length of a file opened with the RECL clause.

Values printed will be separated by commas. Strings will be enclosed in quotes. The format in which the values are printed is compatible with that expected by the READ statement, so a file may be written with PRINT and then read back with READ.

QBASIC contains two extensions in the PRINT statement not provided by GBASIC. If an <expression> with an Integer value is followed by a sharp sign (#), it will be printed as a hexadecimal value. If an <expression> with a String value is followed by an exclamation point (!), the string will be printed without quotes. Use of the latter feature may result in a file which cannot be read with a READ statement but may be useful as input to other Marinchip software or to a QBASIC program using the LINE input option. These features can often remove the need for a PRINT USING statement.

The following are examples of the sequential file PRINT statement:

```
PRINT #1; A,B,J%,V$,T%
```

```
PRINT # QFILE% ; DAT$,M

PRINT # QTEXT% ; LINE$!
```

8.6.2. Random file PRINT

```
PRINT # <expression>, <expression> ; <expression>,...
```

This form of the PRINT statement may be used only with files activated with the RECL clause. The first <expression> specifies the file number being written. The second <expression> specifies the record number being written. Record numbers must be in the inclusive range from 1 to 32767. If either the file number or record length <expression> is Real, it will be rounded to Integer. If either is a String, an error will occur.

The <expression>(s) following the semicolon will be written to the file as described above for the sequential file PRINT statement. Since the random file PRINT may only be done to files used with the RECL specification, it is important to make sure that the RECL specification is long enough to accommodate the longest record written to the file with a PRINT statement.

The following are examples of the random file PRINT statement:

```
PRINT # 1,1 ; DATE, FILE.TITLE$
PRINT # INVFILE%, RECNO% ; PARTNAME$, QTY
```

8.7. PRINT USING statement

The PRINT USING statement may be used with files to write formatted data to a file. While such data may not normally be read back with a READ statement, it is useful when writing files intended to be read by other languages or utilities, or when preparing a file to be printed offline or saved for later printing. The PRINT USING statement for files uses format specification strings identical to those used for the normal PRINT USING statement for the console and printer, so format strings will not be described here. Refer to the discussion of PRINT USING above for complete information on format strings.

PRINT USING may be used with both sequential and random files. The two forms of the statement will be discussed separately.

8.7.1. Sequential file PRINT USING

```
PRINT USING <expression> ; # <expression> ;
            <expression>,...
```

The sequential form of the PRINT USING statement takes the first <expression> as the format string, the second <expression> as the file number, and the <expression>s following the second semicolon as the values to be edited and output according to the format specifications.

The first <expression>, the format string, must be of type String. The rest of the statement adheres to the same rules described above for the Sequential file PRINT statement.

The following are examples of the Sequential file PRINT USING statement:

```
PRINT USING "Amount owed $$#####.##" ; # 1; AMT.OWED
PRINT USING FMT1$; # RPT.FILE% ; C.NAME$, C.ADDR$, \
            C.ZIP, C.BAL
```

8.7.2. Random file PRINT USING

```
PRINT USING <expression> ; # <expression>, <expression> ;
            <expression>,...
```

The random file PRINT USING may be used only on files activated with a RECL specification. The action of the random file PRINT USING is identical to that of the sequential form described above except that the second <expression> following the sharp sign (#) selects which record (from 1 to 32767) the output will be written into.

The following are examples of the random file PRINT USING statement:

```
PRINT USING "## &"; #1, J%; ITM.NO%, NAME.TAB$(ITM.NO%)
PRINT USING FORMAT1$ ; #ZFN%, Q%; A, B, C
```

8.8. PUT statement

The PUT statement writes data to a file in internal format, without editing numbers into decimal form or putting quote marks around strings. The resulting record can NOT be read by any standard Marinchip software except the GET statement in QBASIC, described below.

Because decimal conversion involves a good deal of computation, the use of PUT can increase the speed of a program that does much writing of numeric values. PUT can also reduce the record size in a random access file, since the size of a numeric field is always predictable and usually smaller than that produced by PRINT. Specifically, an Integer values uses 2 bytes, a Real uses 8, and a String uses 2+LEN(string). There is no comma between fields, no end of record character, and no end-file record.

8.8.1. Sequential file PUT

```
PUT # <expression> ; <expression>,...
```

The sequential file PUT statement behaves like the sequential file PRINT statement in all respects except that the <expression>s are written in internal format and not edited into a standard ASCII representation.

The following is an example of the sequential PUT statement:

```
PUT #1; A,B,J%,V$,T%
```

8.8.2. Random file PUT

PUT # <expression> , <expression> ; <expression>,...

The random file PUT statement behaves like the random file PRINT statement in all respects except that the <expression>s are written in internal format and not edited into a standard ASCII representation.

The following is an example of the random PUT statement:

```
PUT # INVFILE%, RECNO% ; PARTNAME$, QTY
```

8.9. GET statement

The GET statement is used to read records that have been written by the PUT statement, described in the preceding section. Data fields are read directly, without any format conversion or type checking. Therefore, the GET statement should have exactly the same I/O list as the PUT statement that wrote the record.

8.9.1. Sequential file GET

GET # <expression> ; <expression>,...

The sequential file GET statement behaves like the sequential file READ statement in all respects except that the <expression>s are assumed to have been written in internal format by a PUT statement.

The following is an example of the sequential GET statement:

```
GET #1; A,B,J%,V$,T%
```

8.9.2. Random file GET

GET # <expression> , <expression> ; <expression>,...

The random file GET statement behaves like the random file READ statement in all respects except that the <expression>s are assumed to have been written in internal format by a PUT statement.

The following is an example of the random GET statement:

```
GET # INVFILE%, RECNO% ; PARTNAME$, QTY
```

8.10. CLOSE statement

CLOSE <expression>,...

The CLOSE statement deactivates an open file. Each <expression> on the CLOSE statement must be an Integer or Real (which is rounded to an Integer before being used) and refers to the file number of a currently open file. An error will occur if the file number is not open.

A CLOSE statement will release all file buffers and return the file to an idle state. For sequential files being written, an end of file

record will be placed at the end. More precisely: an end of file will be written if the last operation on the file was a PRINT and the file has not been accessed randomly since it was opened.

The following are examples of CLOSE statements:

```
CLOSE 1,2,3
CLOSE INPUT.FILENO%, OUTPUT.FILENO%
```

QBASIC will automatically close all files when a STOP or CHAIN statement is executed, or the program is terminated by typing Control C. Files WILL NOT be closed if the program is terminated by a runtime error.

8.11. DELETE statement

```
DELETE <expression>,...
```

The DELETE statement deletes open files from the file directory. Each <expression> on the DELETE statement refers to a file number of a currently active file. In addition to closing the file as described for the CLOSE statement above, the DELETE statement will cause the operating system to release all space assigned to the file. Its name will be dropped from the file directory.

Under the Network Operating System an error will occur if the user who makes the DELETE request is not permitted to delete the named file.

The following are examples of DELETE statements:

```
DELETE 1,2,3
DELETE TEMPFILE.NO%
```

8.12. MOUNT and DISMOUNT statements

```
MOUNT <expression>
DISMOUNT <expression>
```

Under the Network Operating System disc volumes may not be changed without the use of the MOUNT and DISMOUNT commands. The QBASIC statements MOUNT and DISMOUNT execute those commands and should be used wherever the user of the system is instructed to change volumes. In the MOUNT statement the <expression> must be a String which gives the unit number, a colon, and (optionally) the name of the volume to be mounted. In the DISMOUNT statement the <expression> must name either the unit or the volume name, followed by a colon.

If the operating system returns an abnormal status for MOUNT or DISMOUNT operation, it will cause an error termination of the program. Reasons for an abnormal status include the specification of a non-existent disc unit or the mounting of the wrong disc by the user. The MOUNT and DISMOUNT predefined functions, described at the end of this chapter, can be used to avoid error termination.

The Disc Executive ignores these statements; therefore, programs may be written to be compatible with both systems.

The following are examples:

```
MOUNT "1:"
MOUNT "2:PRODUCTION"
DISMOUNT UNIT$
DISMOUNT "PRODUCTION:"
```

8.13. LOCK and UNLOCK statements

Under the Network Operating System it is possible to lock a file or a single record to assure that only one user is updating it at a time. Use of this feature is discussed in detail in the next chapter.

Under the Disc Executive system the LOCK and UNLOCK statements have no effect.

8.13.1. LOCK FILE statement

```
LOCK FILE <expression>
```

The LOCK FILE statement uses facilities of the Network Operating System to assure that only one user is accessing the file at a time. The <expression> must have a Real or Integer value; if it is Real, it is rounded to an Integer. The result must be the file number of a file which is currently open.

If the file is not locked, or this program already has it locked, QBASIC will lock it and return control to the program. If another user has the file locked, the program will be suspended until the file is unlocked, and will then lock it and continue.

The following are examples:

```
LOCK FILE 1
LOCK FILE INVENTORY%
```

8.13.2. UNLOCK FILE statement

```
UNLOCK FILE <expression>
```

The UNLOCK FILE statement unlocks a file which has been locked by the LOCK FILE statement. The <expression> must be an Integer value or a Real value (which will be rounded to an Integer) and must be the number of a file which is currently open and locked.

The following is an example of the UNLOCK FILE:

```
UNLOCK FILE RECEIVABLES%
```

8.13.3. LOCK record statement

```
LOCK #<expression>,<expression>
```

The LOCK record statement assures exclusive access to a record in a file which has been opened or created with the LOCK option. Both <expression>s must be Integers or Reals (which will be rounded to Integers). The first <expression> must be the number of a currently

open file. The second is the number of the record to be locked. Note that this operation can only be used on a file with fixed record length.

If the record is already locked, the program will be suspended until the record is available.

The following are examples of the LOCK record statement:

```
LOCK # FILE1%, 93
LOCK #4,RECNO%+2
```

3.13.4. UNLOCK record statement

```
UNLOCK #<expression>,<expression>
```

The UNLOCK record statement releases the exclusive use of a record which has been locked by the LOCK statement. Both <expression>s must be numeric and are interpreted as in the LOCK record statement.

The following is an example of the UNLOCK record statement:

```
UNLOCK # PAYABLES%,RECNO%
```

3.14. File related functions

QBASIC contains four predefined functions related to file handling. These functions are described in the following sections.

3.14.1. RENAME function

```
RENAME(NEW$,OLD$)
```

The RENAME function takes two String arguments and returns an Integer value. The String expression OLD$ represents the name of an existing file. The RENAME function causes that file's name to be changed to the name given by the String expression NEW$. If the RENAME operation is successful, -1 is returned by RENAME. If the RENAME cannot be done, 0 is returned. Hence, an IF statement may be used on the result of RENAME to test whether it was performed successfully. The following is an example of the use of RENAME:

```
A%=RENAME("NAMEFILE.DAT","TEMP3.WRK")
```

3.14.2. SIZE function

```
SIZE(NAME$)
```

The SIZE function takes a String argument containing a file name and returns an Integer value equal to the size of the named file in "K" (blocks of 1024 bytes). If the named file does not exist, SIZE will return zero. SIZE returns the amount of space allocated to the file, rounded up to the next 1024 byte boundary. Under the Disc Executive, this will be the size allocated to the file when it was created. Under the Network Operating System, the number returned will represent the highest address written in the file.

Under QBASIC the file name given to SIZE must be an explicit file name. The CBASIC feature of permitting names designating groups of files is not implemented.

The SIZE function is especially useful when creating files under the Disc Executive. For example, the following program fragment will CREATE a file named "UPDATE.DAT" with the same size as the existing file "CURRENT.DAT".

```
S$=STR$(SIZE("CURRENT.DAT")*8)
CREATE "UPDATE.DAT"+","+S$ AS 1
```

Note that in the above example we multiplied the value returned by the SIZE function by 8 because SIZE returns blocks of 1024 bytes (to be compatible with CBASIC) while CREATE requires a file size in units of 128 bytes.

8.14.3. MOUNT and DISMOUNT functions

```
MOUNT(<expression>)
DISMOUNT(<expression>)
```

The MOUNT and DISMOUNT functions have the same effect as the MOUNT and DISMOUNT statements described above, but return an Integer value indicating whether the operation was performed successfully. If the operation is successful (normal status returned by the operating system), the function will return a value of -1 (true). If it fails for any reason, it will return a value of 0 (false) without causing program termination.

The following program fragment is an example of the use of the MOUNT function:

```
WHILE MOUNT("2:GOODDISC")=0
   INPUT "Mount the right disc and press RETURN"; LINE A$
WEND
```

9. Using QBASIC files

The previous chapter has described the file-oriented facilities in QBASIC. Effective use of these facilities requires more than a simple enumeration of all the statements and functions available. This chapter explains how QBASIC files are implemented, how the statements access them, and how they should be used when designing applications and writing programs in QBASIC.

9.1. General file characteristics

Regardless of what modes are selected and which statements are used to read and write files, certain characteristics of QBASIC files remain unchanged.

All QBASIC files used with the READ and PRINT statements are written in character format, using the ASCII code used throughout all Marinchip software. String data is written as stored in the program, and numeric information is edited to the character representation and written as an ASCII number. String data is written surrounded by quote marks ("). On input, QBASIC scans numbers into internal format, and processes quoted strings according to the normal rules. Non-quoted strings may also be read from files, but they may not contain commas, which are used as delimiters for such strings.

Because QBASIC uses quote marks to delimit strings in a file, a string containing an embedded quote mark receives special treatment from the system. In writing such a string, QBASIC will replace each embedded quote with two successive quotes; on input, pairs of quotes will be reduced to one quote. This operation is invisible to the QBASIC program; however, the string will occupy more space on disc than it did in memory, which may cause a fixed-length record to overflow.

Files used with the GET and PUT statements are written in an internal format which cannot be read by any other standard Marinchip software. The advantage of such files is that they are compact and can be read and written more quickly than ASCII files. The program must determine where the end of file is, since QBASIC does not write an automatic end-file record as it does when a file is written sequentially with PRINT. If GET and PUT statements are mixed with READ and PRINT in accessing the same file, it is the responsibility of the program to know which records are in which format.

All QBASIC files are made up of RECORDS. These records may be fixed length or variable length, depending on the file type, and may be accessed either sequentially (in order of appearance in the file) or randomly (in any order whatsoever) depending on the file type and the QBASIC statements used to access the file.

Each RECORD consists of one or more FIELDS. Each field represents one data value. The number of fields in a record is limited only by the length of the record.

9.2. File organisation

The organisation of a file refers to how the records that make up the

file are physically stored on the medium on which the file resides. QBASIC offers two kinds of organisation, STREAM (or variable), and FIXED.

9.2.1. Stream files

A QBASIC stream file is a file activated without the RECL specification in the OPEN, CREATE, FILE, or GETFILE statement. Such a file is written as a free-format stream of characters, hence the name. Records are variable length, with records being separated by carriage return characters. The end of file is indicated by a record with an EOT character (ASCII 4) in column 1. For compatibility with CP/M files, a record with an ASCII SUB character (hex 01A) in column 1 will also be treated as an end of file.

This method of file storage is exactly compatible with that used by all other Marinchip utilities and languages, so QBASIC programs may interchange programs with these other software components.

Stream organisation is the most efficient in terms of disc space; a stream file will always use less disc storage than a fixed file of the same number of records. However, since stream file records may all be of different lengths, there is no way to directly locate a record without reading all records prior to it; that is, there is no way to randomly access a stream file. This is the reason why stream files may not be used with the random access forms of the READ, PRINT, GET, or PUT statements.

When using QBASIC to read and write files intended to be used with other languages, output files are normally written with the PRINT USING statement, and input files are normally read with the LINE option on the READ statement. Since other languages use different conventions for storing data in lines, it is normally necessary to treat the file data as strings which are composed and scanned within the QBASIC program itself.

9.2.2. Fixed files

All records in a fixed file will have the same length. The term "fixed file" refers to the fixed length of the records within the file. Fixed files are normally used where random access is required. The format of fixed files is unique to QBASIC; it is not in general possible to interchange fixed files with other Marinchip languages or utilities.

Fixed files are declared by using the RECL clause in the CREATE, OPEN, FILE, or GETFILE statement. Once a fixed file has been initially created, it must always be used subsequently with the same record length (RECL specification). Failure to do so will lead to strange and undesirable results. The same applies to the LOCK option, described later in this chapter.

Fields within records in fixed files are written exactly as described for stream files above. However, for fixed files each record will be padded to be equal to the RECL specified number of characters. The data used to pad the record will be whatever random characters happen to be in a buffer at the time. The user must insure that the longest

record written to the file will be shorter than the RECL length by at least one character (to allow for the carriage return character at the end), or else the program will be terminated in error. If the RECL specification is much larger than the longest record in the file, much disc space will be wasted because each record will consist mainly of garbage to pad the record to the specified length.

Fixed files may be used with either sequential or random access. If a file is to be used entirely for sequential access, it should be made a stream file, since making it a fixed file only wastes disc space. It is quite common to intermix sequential and random accesses on the same fixed file. For example, a file may be initially written sequentially, then updated and searched randomly.

Care should be taken when using the IF END statement with fixed files. If the file is read and written sequentially and never accessed randomly, IF END will work as expected. It is unwise to use IF END when a file is being updated, extended, or accessed randomly, as results may be very difficult to understand. It is much better to maintain a variable in the program which keeps track of the highest record number in the file, which is then used to test for accesses past the end of the file.

If you must have an end-file record at the end of a file which is accessed randomly, the following statement will write one:

```
PRINT FILENO%,LASTREC%+1; CHR$(4)!
```

In this example LASTREC% is the number of the last record that contains good data. READing LASTREC%+1 will trigger IF END. For compatibility with CP/M files, a record with an ASCII SUB character (hex 01A) in column 1 will also be treated as an end of file.

9.3. Appending to a stream file

It is possible to read part or all of any file, using the standard sequential READ statement, then begin writing to it with sequential PRINT. The following program fragment illustrates this:

```
    OPEN A$ AS 5
    IF END #5 THEN 99
    WHILE 1
       READ #5; LINE DUMMY$
    WEND
99  PRINT #5; "One more record"
    CLOSE 5
```

QBASIC allows the writing of a stream file to begin only at the start of the file (before any READ) or after end of file is detected. QBASIC allows writing to begin at any point. Of course, any records after the point at which writing begins will be lost. An attempt to read a stream file after writing has begun is an error.

9.4. Device files

Marinchip's operating systems allow devices (e.g., terminals and printers) to be assigned as files. The standard file input/output

facilities in QBASIC allow for the fact that devices are generally used differently from discs.

Operations on a disc file use a buffer which is private to that file. To get the best execution speed, actual reading and writing of the disc take place only when necessary; if the buffer is big enough, it will be possible to read or write several records with only a single disc access.

On a device file visibility is more important than execution speed. A line printed on the console should not be held in a buffer until the buffer fills up, but should go on the screen immediately. The same applies to the last line of a report on the printer. Therefore, every operation on a device file will cause a single, immediate operation on the device, at the start of a READ or the end of a PRINT.

Because nothing is held in a buffer between operations, all device files use a common buffer, and the BUFF specification in the OPEN statement is ignored.

At present devices must be treated as stream files, with variable record length. The concept of random access or fixed record length in a device file is not defined.

9.5. File and record locking

Under the Network Operating System several programs or several copies of the same program may be running at the same time. If two of these programs try to update the same file at the same time, unpleasant things can happen.

For example, consider an on-line inventory maintenance system. Salesman A gets a request for 40 widgets from a customer; he asks the inventory program for the current count of widgets, finds that there are 50 in stock, and starts negotiating a price. While this is going on, salesman B sells 30 of the 50 widgets and has the inventory program reduce the stock of unsold widgets to 20. Now A finishes negotiating the sale of 40 widgets and has the program reduce the count from 50 to 10. B's transaction has disappeared, the widgets are oversubscribed, and someone is going to be upset.

Obviously the inventory program should not remember the count of 50 widgets for ten minutes while A is negotiating; it should reread the count from the file immediately before it removes the 40 widgets from stock. Then it will find that there are no longer 50 widgets, and the file will not get updated wrongly. Of course, A's customer will be disgruntled when the deal suddenly evaporates! Worse yet, the problem of file integrity still isn't solved. There is a moment between reading the record and writing it back - maybe a very long moment, depending on the vagaries of computer time sharing - in which someone else might update the record without A's program detecting it.

What A needs is a way of locking everyone else out of the file for at least a short time while he updates the file. The LOCK and UNLOCK statements in QBASIC enable the author of the inventory program to provide such protection.

It must be stressed that LOCK and UNLOCK provide a way of writing well behaved programs that do not interfere with each other, but they do NOT protect against renegade programs which misuse or completely ignore LOCK and UNLOCK.

9.5.1. Using file lock

The simplest way of protecting a file against conflicting updates is to lock the file, as in the following program fragment:

```
LOCK FILE 2
READ #2,RECNO%; DATAFIELD, ANOTHER$
PRINT #2,RECNO%; DATAFIELD+1, ANOTHER$
UNLOCK FILE 2
```

File locking is a simple and efficient technique. It can be performed on any open file and does not require extra disc space or disc accesses. Its only drawback is that you may not want to lock up a whole file for a long time; in that case, you may need to use record locking, described in the next section.

File locking is handled on a first come, first served basis. If a program tries to lock a file that another program has already locked, it is suspended until the file is unlocked. If many programs are trying to lock the same file, each one will eventually get its turn.

A file is unlocked under the following conditions: (1) the program UNLOCKs the file; (2) the program CLOSEs or DELETEs the file; (3) the program takes a normal exit, which automatically closes all files; (4) the operating system is reloaded. If the program takes an error termination, files may not be reliably unlocked. This is a good reason for not keeping a file locked unnecessarily. (See the section on the UNLOCK program.)

9.5.2. Using record lock

Sometimes locking a whole file is too drastic. In the example of salesmen A and B the inventory record for widgets needs to be locked, but there is no need to lock the whole inventory file. If the widget record is locked without locking the whole file, people who are trying to sell gadgets and blivets can go about their business while A is negotiating.

Record locking is allowed only in files which have been CREATEd and OPENed with the LOCK option. Such files have extra space in each record, beyond the length given in the RECL specification, to hold a lock. The following program fragment illustrates record locking:

```
OPEN "INVENTORY" RECL 112,LOCK AS 1
(Ask the user for a part name and look it up)
LOCK #1, RECNO%
READ #1, RECNO%; PART.NAME$, COUNT%
PRINT "How many ";PART.NAME$;
INPUT "s did you sell?"; SOLD%
IF SOLD%>COUNT% THEN \
   UNLOCK #1,RECNO%  :\
   GO TO 9999
```

```
PRINT  #1, RECNO%; PART.NAME$, COUNT%-SOLD%
UNLOCK #1, RECNO%
```

If the user spends a long time thinking about his answer to the INPUT statement, other people trying to access that record will be held up, but the rest of the file will be available.

Note that in case of error (selling more widgets than are available) this program unlocks the record without bothering to write any new data.

Record locking presents far more hazards than file locking in terms of recovery from error conditions. For instance, a locked record is NOT automatically unlocked when the file is closed or the program exits. Program bugs, system crashes, and the chance that the user may kill a program with Control-C all contribute ways of leaving a record locked indefinitely, causing programs to hang when they try to access the record later. Therefore, the designer of a data base that will use record locking must work out procedures for recovering from all types of crashes and must be sure that the users of the system will follow those procedures.

When a new record is added to a file, there are two conditions that require special attention. First, whatever procedure is used to allocate a record number for a new record must obviously must be executed under a lock, so that two users will not simultaneously allocate the same space. Second, the new record space may contain random garbage in the record lock field. In particular, suppose that the file contains records 1 through 23, and for some reason you decide to write record 41 next, leaving the intervening space empty for the moment. When you write record 41, the system will allocate disc space for records 24 through 40, and the contents of that space may or may not look like records that are already locked.

To avoid these problems, we strongly recommend this procedure:

1. Lock the file.
2. Execute the procedure to allocate a new record number.
3. Lock the new record.
4. Unlock the file.
5. Write the new record.
6. Unlock the new record.

Steps 3 and 4 use a special case in the record locking system: while the file is locked, any attempt to lock a record will immediately succeed, even though the record may already be marked as being locked. For this reason, file locking and record locking should be used on the same file ONLY when executing procedures such as this.

9.5.3. The UNLOCK program

The UNLOCK program, provided on the NOS/MT QBASIC release disc, is designed for manual recovery of error conditions in which a file or record is left locked. It can be run immediately after an error by the same user who was running when the error occurred, and it can be used by a privileged user to clear up error conditions left by any user in any file.

UNLOCK will act on commands to unlock a file or a specified set of records in a file. It will also lock or unlock all the records in a file. For more information, execute UNLOCK, which displays its own operating instructions.

UNLOCK is primarily a tool for the use of a data base manager doing a manual cleanup. For a simple system using file locking it may be the only tool needed. It will not take the place of careful planning of recovery procedures, which is necessary in any data base system whatever. For more complex systems, or for any system that uses record locking, it will almost certainly be necessary to write recovery programs which are to be invoked, as automatically as possible, after any crash.

10. Hardware and machine language interface

This chapter describes the features in QBASIC which permit QBASIC programs to directly interface with hardware devices, and to call subroutines written in assembly language.

Since QBASIC is a true compiler, assembly language subroutines may be written with much greater ease than with CBASIC, and the interface is much more powerful, permitting arguments to be passed to them. Assembly language routines are called by name, not by address, so programs are much easier to maintain and update.

10.1. Memory inspect and change

The PEEK function and POKE statement allow direct access to system memory, and data transfers to memory-mapped peripherals.

The PEEK and POKE functions both reference actual machine addresses. These addresses are in the range from 0 to 65535, the address range of a user space on the machine. The expression for the address is expected to be an Integer; if it is a Real, it will be rounded. Since an address is a 16 bit unsigned number and an Integer is a 16 bit signed number, it is not possible to directly specify an address greater than 32767 using an Integer without some trickery. To represent an address greater than 32767, it is necessary to use the negative Integer which corresponds to the desired unsigned address. For example, address 65534, which in hexadecimal is FFFE, would be addressed as -2, since -2 is the Integer whose bitwise representation is FFFE.

A much more straightforward way of specifying addresses for PEEK and POKE is to use QBASIC's hexadecimal constant feature. Using this feature, an address is simply written in hexadecimal with a leading zero. Hence, the address FFFE can be written as 0FFFE rather than -2. This is generally much easier to understand when reading the program at a later date.

10.1.1. PEEK function

```
PEEK(<expression>)
```

The PEEK function returns an Integer value equal to the 16 bit word at the address given by the <expression>. The address <expression> must be Integer or Real; if Real, it will be rounded to Integer before use. If the address given to PEEK is odd, it will be rounded down to the preceding even address to form the address for the word returned by PEEK.

QBASIC's PEEK function differs from CBASIC's in being word-oriented, as befits the 16 bit computer on which QBASIC runs.

10.1.2. POKE statement

```
POKE <expression>,<expression>
```

The first <expression> specifies a word address within computer

emory, and the second <expression> gives the 16 bit value to be tored there. Both <expression>s must be Integer or Real, with Real values beins rounded to Integer before use. If the address expression> is odd, it will be rounded down to the precedins even ddress to form the word address where the information is stored.

Note that POKE always stores 16 bits. To replace only a single byte, he word should be read with PEEK, the data should be masked into the ord usins the losical operators, then the word stored back with POKE.

BASIC's POKE statement differs from CBASIC's in that it is word riented rather than byte oriented.

0.2. Hardware input and output

The INP function and OUT statement allow direct byte transfers to and from input/output ports on the machine. These mechanisms work only or S-100 ports; CRU devices such as the Quad SIO board require an assembly language interface.

0.2.1. INP function

INP(<expression>)

he INP function returns an Integer value which is the result of reading the S-100 I/O port with the address given by the <expression>. he <expression> must have an Integer or Real value, with Real beins ounded to Integer before use. I/O ports have numbers between 0 and 255, so only numbers in that range will produce valid results. The value returned by INP will be between 0 and 255 if the port read is an bit device, and will be a full Integer value if the port is a 16 bit evice.

0.2.2. OUT statement

OUT <expression>,<expression>

he OUT statement sends the value of the second <expression> as output to the S-100 I/O port whose number is given by the first <expression>. Both <expressions> must be of type Integer or Real, with Real values eins rounded to Integer before use. Since I/O ports have numbers etween 0 and 255, only values in that range will result in correct action by the OUT statement. The entire 16 bits of value of the econd <expression> will be sent to the output port. If the port is n 8 bit device, only the low order 8 bits will be used. If the port is a 16 bit device, the entire word will be used.

0.3. Assembly language interface

BASIC programs may call assembly language subroutines. These ubroutines are separately compiled subprograms which are LINKed with he QBASIC program before execution. The subprogram linkage permits rsuments to be passed to these subprograms.

BASIC's assembly language linkage facilities are totally different from those of CBASIC, which suffer from the fact that CBASIC is really n interpreter. In QBASIC subroutines are called by name, not by

address, and memory for them is assigned by the Linker, not by a "mechanism" such as the CBASIC SAVEMEM statement.

10.3.1. CALL statement

```
CALL <name>[(<args>,...)]
```

The CALL statement is used to invoke an assembly language subroutine. The <name> used in the CALL statement corresponds to the externalised name of the entry point to the assembly language subroutine and must not duplicate a variable name or any name used in an ENTRY or EXTERNAL statement. Since this name is used as a Linker external name, only the first six characters are significant.

The name may be followed by an optional list of argument expressions. Arguments may be Integer, Real, or String, and any expression desired may be used as an argument to an assembly language subroutine.

The CALL statement does not provide an explicit way for an assembly language subroutine to return a value. This restriction can be overcome by use of the function ADRS. The function ADRS returns the address of its argument, so the address of a variable can be passed, rather than its value, permitting a cooperating assembly language subroutine to store a result into the variable. For example, we might have an assembly language routine called CPYFIL which copies one file into another. This routine might be called with:

```
CALL CPYFIL(INFILE$,OUTFILE$,ADRS(STATUS%))
IF STATUS%<>0 THEN PRINT "Failure!" : ...
```

The address of the variable STATUS% was passed, allowing the assembly language routine to return a status code in it.

10.3.2. Writing assembly language subroutines

When a CALL statement is executed, each argument expression is evaluated and their values are placed on the runtime stack. Register R10 is the stack pointer, and always points to the next available word on the stack. Arguments will be pushed on the stack with the last argument on top (highest address).

An Integer uses 2 bytes of stack space; a Real takes 8; a String takes 2 bytes. Reals are stored in the standard IBM 370 long format used throughout the system, and Integers are stored as 16 bit two's complement numbers. The 2 bytes used for a String argument consist of the address of a string buffer which actually contains the value of the String argument. The first word of a string buffer holds the length of the string in bytes, and the actual text of the string starts in the second word and continues for as many words as are required. (Certain rules must be strictly observed in creating or modifying strings. Read this entire chapter before you even consider such operations.)

The assembly language subroutine is called by performing a BLWP to the subroutine name used in the CALL statement. Hence, the external label should be a BLWP vector, not the first executable instruction of the subroutine. Since the assembly language subroutine is called with a

BLWP, it has its own private register set, in which it may freely use R0-R12 with no danger of register conflicts with QBASIC. Since the assembly language routine must find its arguments on the runtime stack, it should copy the caller's R10 from 20(R13) into its own R10. It must then pop the arguments from the stack, decrementing R10 as it goes (note that QBASIC's stack usage is compatible with the PSHR and POPR pseudo instructions in the Assembler). Before returning to the caller, it must then store the final R10 with all arguments popped from the stack back into 20(R13). The actual return to the calling program is effected by performing a RTWP instruction.

This is actually a lot easier to do that the above explanation seems to imply. Examining the following example should make things much more clear. In this example, we wish to write an assembly language subroutine which will perform a circular shift on an Integer argument with the shift count being supplied as a second argument. We will call this subroutine from QBASIC with a call like:

```
CALL SRC(ADRS(VALUE%),COUNT%)
```

where VALUE% is the variable we wish to shift and COUNT% is the number of bits we wish to shift VALUE% right circularly. We would code this subroutine as follows:

```
        idt     "SRC"
        dstk    R10
.
src*    data    regs,src1           BLWP vector for entry
.
src1    mov     20(r13),r10         load caller's stack pointer
        popr    r0                  pop shift count into R0
        popr    r1                  pop argument address into R1
        mov     *r1,r2              load argument value
        src     r2                  shift R2 by count in R0
        mov     r2,*r1              store back in variable
        mov     r10,20(r13)         update QBASIC stack pointer
        rtwp    .                   return to caller
.
regs    bss     32                  register workspace
.
        end     .
```

Examination of the above subroutine should illustrate how an assembly language subroutine is called, how it accesses its arguments, how it returns values through ADRS arguments, and how it updates the stack and returns to the program which called it.

The fact that the CALL subroutine has its own workspace prevents conflicts with register use by QBASIC, but is sometimes inconvenient. For instance, the QBASIC library routines, which operate in the main workspace, are not directly available. The QBASIC library provides two routines which make it easier to link to the rest of the library.

To get into the main workspace, simply BLWP SYSWS$. Upon return you have direct access to the QBASIC library, the stack (R10), and R0-R5. Other registers must NOT be used.

To return to the private workspace from which SYSWS$ was called, simply BLWP USRWS$. From there, a RTWP will return to the calling program.

For convenience, a CALL routine which has transferred to the main workspace via SYSWS$ can return directly to the calling program by executing a RTWP. That is, these two sequences are equivalent:

(1)

```
        blwp    sysws$          enter system workspace
        blwp    usrws$          back to local workspace
        rtwp    .               back to caller
```

(2)

```
        blwp    sysws$          enter system workspace
        rtwp    .               directly back to caller
```

10.3.3. Writing assembly language functions

QBASIC provides a reasonably simple interface for writing user defined functions in assembly language. These functions are called in exactly the same way as functions written in QBASIC and return a value of type Integer, Real, or String.

The entry to an assembly-language function looks like this:

```
name*   mov     r11,r1
        bl      fentr$
        <string parameter definitions>
        <terminator>
        <start of code>
```

The exit is simply:

```
        b       ret$
```

The function executes in the main workspace and should use only R0-R5, R11, and the stack -- don't forget the directive "DSTK R10". When the routine starts executing at <start of code>, the arguments to the routine (if any) are in ascending locations starting at the address to which R7 points. As always, Integers and Strings take 2 bytes; Reals take 8.

The function result is at -6(R7) if it is a String or an Integer, or -12(R7) if a Real. At <start of code> the value is pre-set to 0 (or the null string). Normally your function will store a function result, but it does not have to. If the result is a String, do not simply stuff a string buffer address in -6(R7); use the library routine AS$, described in the next section.

The <string-parameter definitions> are omitted if the function has no string-valued arguments. If it has, then for every String argument to the function there must be one DATA word giving the location of the argument within the block that R7 points to. See the example below.

The <terminator> has the high bit (08000) set. The next bit is set

(0C000) if and only if the function result is a String. The rest of the bits give the total length of the argument block; this allows ENTR$ to catch a grossly wrong call.

The following example illustrates function linkage. It is the same as the routine which illustrated the CALL subroutine, except that it returns its result as a function result, not through an ADRS argument.

```
        idt     "RSHIFT"
        dstk    r10
.
src*    mov     r11,r1          copy the return address
        bl      fentr$          initialise
        .                       no string arguments
        data    08004           terminator: 4 argument bytes
        .                        (2 Integers)
        mov     2(r7),r0        load shift count
        mov     *r7,r1          load argument value
        src     r1              shift R1 by count in R0
        mov     r1,-6(R7)       store function result
        b       ret$            return
.
        end     .
```

The program that calls this function must use an EXTERNAL statement, defined in a later chapter, to link to it. It might contain the following two statements:

```
EXTERNAL SRC=FNSHIFT%
A% = FNSHIFT%(B%,3)
```

10.3.4. Library entries

An assembly-language routine operating in the main workspace can call many QBASIC library routines, some of which are described here. A CALL routine must call SYSWS$ before calling library routines, in order to get into the right workspace; a function is automatically in the right workspace.

To allocate and release free memory space for any purpose other than strings, use ALB$ and RLB$:

```
        li      r0,<length>
        bl      alb$
        <block location in r1>

        li      r1,<location of block to release>
        bl      rlb$
```

The rest of this section describes string-handling routines. Any assembly language program that creates, deletes, assigns, or modifies Strings MUST use these routines. Any attempt to duplicate their functions may destroy all Strings in the program. In particular, a program must never modify the contents of an existing String; instead, it should copy the String with STCOP$, then modify the copy.

All the String handling routines preserve registers R2-R5.

AS$ is used to assign a new value to a String variable. It is called with the new value (a pointer to a string buffer) on top of the stack and the address of the variable in R0. AS$ takes care of releasing the value previously assigned to the String variable. If you define a String variable in an assembly-language routine, be sure to initialize it with "DATA 0".

SR$ places the value of a String variable on the stack. It is called with the variable address in R0. Do not stack a String with a simple PSHR <variable address>.

SC$ puts a String constant on the stack. The call is

```
        bl      sc$
        data    08001
        data    <text length>
        text    "<text string>"
        even
```

The constant may not be modified in any way.

STNEW$ creates a string of a specified length, containing random data. It is called with a count in R0, and returns with a string pointer in R1. The character count that was given in R0 has been copied into the count field of the string (at *R1); you need only to copy the desired text into the text space starting at 2(R1). Having prepared the string, do something with it: leave it on the stack for somebody else to pick up, or assign it to a String variable with AS$ (see below). If you want to save its address for use in a later call on your routine, do NOT simply store the address in a local variable; do a formal string assignment using AS$.

STCOP$ creates a copy of a given piece of text. It is called with a count in R0 and text address in R1. It returns with R1 pointing to a string into which the given text has already been copied. You may modify this string in any way you like, except increasing the length. Then do something with it, as after STNEW$.

STNUL$ places a null string on top of the stack. NOTE that a String variable normally holds a null string as 2 bytes of binary 0, but a null string on the stack must not be a simple 0; it must be a pointer to a string with length 0.

The following example illustrates function linkage and string handling. It takes a string and an integer as arguments and returns a string which is a copy of the string argument except that the first 2 bytes are replaced with the 16 bits taken from the integer.

```
mki*    mov     r11,r1
        bl      fentr$          call the entry routine
        data    0               first argument is a string
        data    0C004           string value,
        .                        4 bytes of argument
        mov     *r7,r1          get address of string
        mov     *r1+,r0         length to r0,
        .                        text address to r1
        ci      r0,2            is the string long enough ?
```

```
        jlt     mki9.           no.  Don't try to store
        bl      stcop$          make a copy of the string
        pshr    r1              save address on stack
        mov     2(r7),2(r1)     store integer in bytes 1-2
        mov     r7,r0           find arg block base
        ai      r0,-6           point to result cell
        bl      as$             assign the result
mki9    b       ret$            that's all
```

10.3.5. Linking assembly language with QBASIC

When a QBASIC program which calls assembly language subroutines is LINKed, the assembly language relocatable files must be named on the command to LINK or QLINKER. For example, if we are linking a QBASIC program called IOTEST which calls the subroutine SRC defined above (and assuming that the SRC subroutine has been assembled into a file named SRC.REL), we would use one of the following:

```
QLINKER 2/IOTEST,2/SRC
LINK 2/IOTEST=2/IOTEST.REL,2/SRC.REL,@QLINK
```

If there are more than two assembly language routines, the LINK command should be used.

11. Compiler directives

Compiler directives are special statements which control the QBASIC compiler. These statements take action at compile time rather than execution time.

Compiler directives all begin with a percent sign (%). The percent sign may start in any column of the input line (in CBASIC, it must start in column 1). There must not, however, be a line number on a compiler directive line. Any information on a line after a compiler directive will be ignored, so compiler directives should be put on lines by themselves.

11.1. %INCLUDE - Copy source file

```
%INCLUDE <filename>
```

The %INCLUDE directive causes the named <filename> to be included into the compilation at the line where the %INCLUDE directive appeared. The <filename> may be any valid file name for the operating system under which QBASIC is being run. Note that the file name does not appear in quotes. For example, a QBASIC program might include a library of file accessing subroutines with the statement:

```
%INCLUDE 2/FILEUTIL.BAS
```

Note that unlike CBASIC, QBASIC makes no assumptions about the form of the file name used on an INCLUDE statement. If the file name ends in ".BAS", it must be specified on the INCLUDE directive.

QBASIC does not permit an INCLUDEd file to %INCLUDE another file (nested includes). This is a restriction not present in CBASIC, which allows includes to nest up to six deep. QBASIC does not, however, impose any restrictions on the total number of includes which may be done in one program.

The %INCLUDE directive is very useful when maintaining large systems of programs. Subroutines, functions, and variable definitions used throughout a system of programs may be placed in INCLUDE files, so that when they are changed, they may be incorporated in all programs which use them simply by recompiling. This eliminates all the extra work, bookkeeping, and probability of error that occurs when multiple copies of one piece of code exist.

INCLUDE is also extremely useful as a means of including assignment statements which specify system standards and the external environment in which the program is run. This makes reconfiguring programs much more straightforward.

The INCLUDE directive may be used to include ENTRY, EXTERNAL, and COMMON statements at the beginning of a program.

11.2. %DEBUG - Print line numbers in error messages

The %DEBUG directive causes the compiler to insert additional code in the program which permits the inclusion in runtime error messages of

he source program line on which the error occurred. The %DEBUG directive should be placed before the first executable statement of the program. Use of the %DEBUG directive adds to the size of the ompiled program, but has little effect on execution speed. Because t is far easier to find program errors when the line number is known, %DEBUG should be used in all programs except those known to be xtremely reliable or where memory size is critical.

If several separately compiled routines are linked together into one rogram, and some but not all contain %DEBUG directives and an error ccurs in a routine where %DEBUG was omitted, the line number printed will be the last line executed in a routine compiled with %DEBUG.

f a program contains %INCLUDEs, lines within the included text will e numbered from 1, and the number of the %INCLUDE, counting from the top of the program, will be appended to the error message.

1.3. %DIAGNOSTIC - Compiler debugging feature

he %DIAGNOSTIC statement is designed to aid Marinchip in debugging he compiler. It causes the compiler to identify the intermediate code generated by the first pass of the compiler with the statement that generated the code. This statement has no other use, and roduces no information of use in finding programming errors. It is entioned here only for completeness.

1.4. Ignored compiler directives

The following compiler directives are used by CBASIC, but are ignored y QBASIC. Their appearance in a program will have no effect on the ompilation.

%LIST	- Turns on CBASIC listing
%NOLIST	- Turns off CBASIC listing
%PAGE	- Sets CBASIC page length
%EJECT	- Ejects CBASIC listing page
%CHAIN	- Sets CBASIC storage sizes

The CHAIN directive is unnecessary since a QBASIC chain completely eplaces the old program with the new, so there is no need to specify emory sizes in advance as is required in an interpreted language like CBASIC.

12. Separately compiled routines

A QBASIC program can be compiled in several pieces, which are then linked together by the system LINK program, just as is done in languages such as Fortran. This should not be confused with the CHAIN facility, which allows a program to call another program into memory, so that both programs will not have to be in memory at the same time. The purpose of the features in this section is not to save memory space, but to write programs more reasonably.

In writing and maintaining a large program it is essential to avoid using the same variable name for different purposes in different sections of the program. Failure to do so leads to errors that can be almost impossible to debug. If the program is written in several pieces of reasonable size, the problem is solved by brute force: the variables in each piece are completely independent unless you take the trouble to share them by means of ENTRY and EXTERNAL statements.

A one-line change in a big program means editing and recompiling the whole thing, unless it was written a several small modules. It is much more fun to recompile a small routine than a big one.

A group of related programs, such as a business accounting system, will have many routines in common. These can be handled by the %INCLUDE statement, which copies them at compile time; but if one of these routines needs to be changed, it becomes necessary to recompile every program that uses it. If the routine is compiled separately, on the other hand, the programs that call it need only to be re-LINKed with the new version.

But the most important advantage is the simplest: small programs are better than big ones. A set of small, well-planned modules is easier to write, debug, and change than a great, unorganized heap of code.

12.1. Main programs, subprograms, and modules

In this chapter a MODULE will mean any separately compiled program, whether a main program or a subprogram. A QBASIC program consists of one or more modules: a main program and zero or more subprograms. A simple, CBASIC-compatible program is a main program with no subprograms.

When a program is executed, the action begins at the top of the main program and proceeds according to whatever statements are in the program. Some of these statements may call functions which are defined in another module; this is the only way that control can get into a subprogram. The subprogram itself can call functions that are defined in other subprograms or in the main program.

In addition, a module can make any of its variables available to other modules. This feature is distinct from COMMON, which makes variables available to a new program which is invoked by CHAIN.

12.2. SUBPROGRAM statement

SUBPROGRAM <name>

When two or more separately compiled modules are to be linked together, all but one of them must be subprograms. The SUBPROGRAM statement simply declares that this is not the main program. The <name> is a name of up to eight letters and digits, which will identify this program in the listing generated by LINK.

A subprogram may not have DATA or COMMON statements.

The SUBPROGRAM statement must be the first statement in the program, apart from blank lines and REMARKs.

12.3. ENTRY statement

```
ENTRY <external name>=<internal name>,...
```

The ENTRY statement declares which of the variables and functions defined in this module will be available to other modules. The <internal name> is the name of a variable or function which appears in this module. The <external name> is a name of no more than six letters and digits, beginning with a letter, which uniquely identifies the variable or function to the outside world.

If the <internal name> is followed by parentheses, then the <external name> refers to the subscripted variable, not the simple variable of the same name. For consistency with the COMMON statement, there may be a single Integer constant inside the parentheses, but it is ignored by the compiler. A function name in an ENTRY statement should not be followed by parentheses, regardless of whether or not the function takes arguments.

The following are examples of the ENTRY statement:

```
ENTRY V0001=PROGRAMMER.NAME$,V0002=NUMBER.OF.SNAFUS
ENTRY W001=FN.DOCK.PAY%,BOBBLE=BOBBLE
ENTRY A=X,B=X%,C=X$,D=X(),E=X%(1),F=X$()
```

Any program module, whether a main program or a subprogram, may have ENTRY statements. ENTRY statements must be grouped at the beginning of the program (after SUBPROGRAM if any) with EXTERNAL statements and before COMMON statements (if a main program).

12.4. EXTERNAL statement

```
EXTERNAL <external name>=<internal name>,...
```

The EXTERNAL statement has exactly the same form as the ENTRY statement and the complementary meaning. That is, it declares that <internal name> is not defined in this module, but is defined in some module which has <external name> in an ENTRY statement.

The following statements would allow a module to use some of the variables and functions defined in the ENTRY statements above:

```
EXTERNAL W001=FN.DOCK.PAY%
EXTERNAL V0002=NUMBER.OF.SNAFUS,V0001=PROGRAMMER.NAME$
EXTERNAL A=Y,B=I%,F=VERYLONGNAME$()
```

The examples illustrate that neither the order of declarations nor the choice of internal names is important; the same object is called X in one program and Y in the other, but they are linked by the <external name> "A". Of course, it is good practice to use the same internal name consistently for the same object.

On the other hand, the objects linked by ENTRY and EXTERNAL must be of the same type: if the second program said EXTERNAL A=Y% or EXTERNAL W001=VARIABLE (instead of an Integer function), the results would be disastrous.

The restrictions on the placement of EXTERNAL statements are the same as those for ENTRY statements.

12.5. Linking separate modules

All the modules that go into a program must be specified in the LINK process. For example:

```
QLINKER MAGIC=SORCERER,SPELL1,WAND
```

This example assumes that SPELL1.REL and WAND.REL are files containing compiled SUBPROGRAMs. Note that the name used is always a file name, not an <external name> or the <name> in a SUBPROGRAM statement.

No more than two external routines should be named on the QLINKER call. If there are more than two, use LINK:

```
LINK MAGIC=SORCERER.REL,SPELL1.REL,WAND.REL,HUMBUG.REL,@QLINK
```

Under NOS, use @1:QBASIC/QLINK instead of @QLINK.

By using the LOC and FETCH functions of the LINK routine it is possible to set up a library of QBASIC routines so that the user does not have to type, or even to know, the names of all the modules that his program requires. The use of these functions is described in the LINK section of the manuals for NOS and Disc Executive. For a practical example, see file QLINK on the QBASIC release disc.

12.6. EXTERNAL variables in the library

The runtime library that handles I/O and other standard operations for QBASIC defines some words that QBASIC programs can access as Integer variables by using the EXTERNAL statement. These allow the QBASIC program to determine its operating environment and to change some operating modes of the library.

NOSFLG contains a 0 if the program is executing with the Disc Executive library, and a -1 if it is using the Network Operating System library. A program can include code to handle the file naming conventions of both systems, using NOSFLG to determine which code to use.

CCHECH determines whether CONSTAT% and CONCHAR% will echo characters as they are typed in. Initially it is non-zero, and all printable characters (ASCII value greater than 31) will be echoed. If the

QBASIC program sets it to 0, no characters will be echoed. CCHECH can be examined and modified in the same way as any Integer variable.

CCWAIT determines the operating mode of CONSTAT%. Initially it is non-zero; CONSTAT% will wait for input before returning control to the calling program, and will always return a value of -1. If the QBASIC program sets CCWAIT to zero, CONSTAT% will return immediately after it is called, exactly as in CBASIC; the value will be -1 if an input character is available and 0 otherwise. CONCHAR%, however, will always wait for input to be available, rather than return a nonsense value.

PRSPC controls the extra blank that is edited after a user-specified prompt in the INPUT statement. Initially it is non-zero, and the given prompt is always followed by a blank. If the QBASIC program sets it to 0, prompts will be given without the following blank.

The following are examples of the use of library external variables:

```
EXTERNAL NOSFLG=NOS%,CCHECH=CCHECH%, CCWAIT=C%

IF NOS%=0 THEN PRINT "Don't run this under Disc Exec"
SAVE.ECHO% = CCHECH%: CCHECH% = 0
C% = SAVE.WAIT%
```

13. Reserved words (keywords)

The following words are reserved for the exclusive use of QBASIC. These words may not be used for variable names within QBASIC programs.

ABS	DATETIME	GOTO	ON	SGN
ACOS	DEF	GT	OPEN	SIN
ADRS	DELETE	IF	OR	SIZE
AND	DIM	INP	OUT	SQR
AS	DISMOUNT	INPUT	OVERLAY$	STEP
ASC	ELSE	INT	PEEK	STOP
ASIN	ELSEIF	INT%	POKE	STR$
ATAN	END	LE	POS	SUB
ATN	ENDIF	LEFT$	PRINT	SUBPROGRAM
BUFF	ENTRY	LEN	PUT	TAB
CALL	EQ	LET	RANDOMIZE	TAN
CHAIN	EXIT	LINE	READ	THEN
CHR$	EXP	LOCK	RECL	TO
CLOSE	EXTERNAL	LOG	RECS	UCASE$
COMMAND$	FEND	LPRINTER	REM	UNLOCK
COMMON	FILE	LT	REMARK	USING
CONCHAR%	FLOAT	MATCH	RENAME	VAL
CONSOLE	FOR	MFRE	RESTORE	WEND
CONSTAT%	FRE	MID$	RETURN	WHILE
COS	GE	MOD	RIGHT$	WIDTH
COT	GET	MOUNT	RND	XOR
CREATE	GETFILE	NE	SADD	
CSC	GO	NEXT	SAVEMEM	
DATA	GOSUB	NOT	SEC	

14. Error messages

14.1. Compiler error messages

When the QBASIC compiler detects an error in the program, it prints the offending line with a pointer to the approximate position where the error occurred. In most cases this is enough to show that a reserved word has been misspelled or a statement has been badly formed. In addition, some errors generate an error message before printing the line, while others do not apply to any one line in particular; these are explained here.

Label S$XXXX not defined

A GOTO, GOSUB, or IF END referred to a line number which was never defined. The field S$XXXX in the message is a coded form of the line number. To find the actual line number, remove the "S$" from the beginning and then make the following substitutions:

For	Substitute
$	.
P	+
M	-

Function XXXX not defined

A function was called or (or named in an ENTRY statement) but was never defined in a DEF or EXTERNAL statement. To decode the name XXXX, translate the last three characters as follows:

For	Substitute
$IF	%
$RF	nothing
$SF	$

In the rest of the name, change "$" to ".".

Array XXXX not defined

A reference was made to an array which never appeared in a DIM, ENTRY, EXTERNAL, or COMMON statement, nor as a dummy variable or actual argument to a function. Therefore, it is impossible for the array to be properly defined at runtime. This usually results from a misspelling of an array name. The name XXXX is decoded as in the case of an undefined function, above, except that the last letter of the coded name will be A, not F.

Improper block nesting

A block is a WHILE - WEND, FOR - NEXT, IF - ENDIF, or DEF - FEND sequence. One of these began or ended within a Single-line IF, but

was not entirely contained in it.

XXXX was expected

A block of statements was ended incorrectly. For instance, if a FOR loop ends with a WEND statement (or is not ended before the end of the program), the message will be "NEXT was expected."

14.2. Second pass errors

When the QBASIC compiler detects no errors in the program, it generates the message:

```
QP2 <program>.REL=TEMP1$
```

No messages should appear between this and the next system prompt. If any message does appear, it should be reported to Marinchip Systems with as complete documentation as possible.

14.3. Runtime error codes

When an error is detected by QBASIC at execution time, the program will be terminated and an error code will be printed. The table below lists the error codes given by QBASIC for runtime errors.

Code	Meaning
0100	Memory full
0101	More RETURNs than GOSUBs or function calls
0102	Range error in ON...GOTO or ON...GOSUB
0103	Wrong number of subscripts
0104	Subscript out of range
0105	Array not dimensioned
0106	Null string as argument to ASC
0107	Bad format string in PRINT USING
0108	Negative dimension in DIM
0109	Length of function arguments is wrong
•110	Too many string expressions in a statement
0111	Too many nested GOSUBs or function calls
0130	Incompatible COMMON statements
0131	Could not read COMMON variables from TEMP2$
0132	CHAIN failed
0133	Could not write COMMON variables in TEMP2$
0141	End of file, no IF END given
0142	OPEN: file cannot be found
0143	OPEN: file was already open
0144	FILE: all unit numbers in use already
0145	CLOSE: file was not open
0146	CLOSE: system save error status
0147	Unit number out of range
0148	I/O to non-open unit number
0149	Write error
0150	Read error

0151	Seek error
0152	DELETE error
0153	OPEN: RECL was less than 1
0154	I/O in a function called from an I/O list
0155	READ beyond end of DATA
0156	Attempt to read or write beyond end of fixed record
0157	Random I/O in stream file
0158	READ after PRINT in stream file
0159	Could not open PRINT.DEV
0160	MOUNT failed
0161	File was not opened for record locking
0162	I/O error locking or unlocking record

When a program is terminated with a runtime error, the line number of the last program line executed will be printed in the error message if the %DEBUG statement was present at the beginning of the program in which the error occurred. If the error occurred in a block of source code copied into the program with an %INCLUDE statement, the number of the %INCLUDE (the first is numbered 1) will be given. Note that in programs with several separately compiled routines, it is up to the user to determine which routine the program was in when the error occurred so that the line number can be traced to the routine in which the error was detected.

15. Comparison of QBASIC with CBASIC

This section attempts to summarise all the differences between the QBASIC and CBASIC languages as seen by an application programmer. It is not concerned with implementation differences, such as the difference between an interpretive compiler and a compiler that generates machine code, except where such differences affect the language specifications.

There are four subsections in this section: (1) CBASIC features which are lacking in QBASIC, or require some extra action on the part of the programmer to get the same effect; (2) differences between the ways in which the two languages interpret a statement, where neither language behaves like a subset of the other; (3) restrictions in CBASIC which have been removed in QBASIC; (4) QBASIC features which are extensions to CBASIC. The last two are of no interest to those who merely want to convert CBASIC programs to QBASIC.

In writing QBASIC and the manual we have attempted to maximise compatibility with CBASIC, including features on which the CBASIC manual is silent (the logic of writing EOF on fixed sequential files) or incorrect (continued DATA statements). Marinchip Systems would like to hear of any undocumented incompatibilities which users may discover. New incompatibilities may or may not be corrected, but they will at least be documented in new editions of this manual.

15.1. Restrictions present in QBASIC

The current release (3.0) does not support the following format types in PRINT USING: ^^ for exponential field editing, leading minus sign, and editing a number with a leading % sign if it is too big for the defined field. These will probably be added in a later release.

CHAIN does not preserve the contents of DATA statements.

COMMON variables are passed in a file, not in memory. This is included in this subsection because the programmer must take one extra action to make COMMON work: under the Disc Executive (not NOS) the file TEMP2$ must be created before the program is executed.

The SIZE function does not accept ambiguous file names.

The following new keywords represent extensions to the language, but are restrictions in that they can't be used as variable names: ADRS, ASIN, COT, CSC, DISMOUNT, ELSEIF, ENDIF, ENTRY, EXTERNAL, GET, GETFILE, LOCK, MOD, MOUNT, OVERLAY$, PUT, SEC, SUBPROGRAM, UNLOCK.

Real values (containing decimal points) cannot be read into Integer variables by READ or INPUT.

%INCLUDE cannot be nested, but there can be any number of them on the top level.

The directives %NOLIST and %EJECT are ignored.

There are no binary constants (e.g., 01111010B).

The maximum number of disc files that can be open simultaneously is set by the operating system configuration, not by tables in QBASIC itself. The limit is normally ten. Any number of device files may be open. The range of file numbers is still 1-20.

15.2. Features treated differently in QBASIC and CBASIC

The order of evaluation of arithmetic expressions in QBASIC differs slightly from that in CBASIC: Unary plus and minus are evaluated before multiplication and division.

QBASIC treats a numeric constant as Real if it contains a decimal point, including a leading (illegal in CBASIC) or trailing decimal point. A constant is also Real if it exceeds the allowable range for integers, -32767 to +32767, even if it has no decimal point.

Real arithmetic is performed in binary (IBM 370 format) in QBASIC and in packed decimal in CBASIC. This can cause rounding problems due to the approximate representation of decimal fractions in QBASIC. For instance, to round to the nearest cent, one should use an expression such as INT(100.*X+.500001)/100.

The value returned by COMMAND$ under NOS includes the whole command line; under Disc Executive it returns everything after the program name, including at least one leading blank. Under either system the value returned is the line which invoked the currently executing program; if the current program was invoked by a CHAIN from another program, the value is that which appeared on the CHAIN statement, not the line that invoked the original program.

PEEK, POKE, INP, and OUT are 16-bit operations. 8-bit input/output devices will ignore the extra bits.

Anything which is specified for PRINTing on console or line printer is printed immediately, even if the list is terminated by a comma or semicolon. This means that no printed output is lost upon CHAIN or error termination. It also allows a program to print an arbitrary expression followed by semicolon as an input prompt.

When floating $ is used in a PRINT USING format, a negative number will be displayed with the $ preceding the minus sign; CBASIC suppresses the $ sign. This may be changed in a later release.

Records are terminated with a Carriage Return only, not followed by Line Feed. Therefore, a fixed record length need be only one byte larger than the longest record which is to be written in it.

Hexadecimal numbers are specified with a leading zero, both in program constants and in input. The trailing H for hexadecimal is not recognised.

The %DEBUG directive is used to enable printing of source line number in runtime error messages. CBASIC uses the "E" compiler toggle for this.

15.3. Restrictions removed in QBASIC

DATA, DIM, DEF, and END statements need not be the only thing on a line.

The Single-line IF statement need not be the first on a line. Single-line IF statements can be nested in a limited way. A THEN-clause or an ELSE-clause can consist of a line number; an ELSE-clause may be used after THEN <line-number>.

The FOR statement accepts mixed-mode expressions.

Strings may be of any length, as long as there is enough room in memory.

Re-dimensioning a String subscripted variable will always recover all the space occupied by the strings that it contains. The subscripted variable need not be set to all null strings beforehand.

A user-defined function need not be defined before it is called. Functions are fully recursive. If execution drops through to a FEND statement, the function will return, not give an error message.

The %COMMON statement is not required in order to CHAIN programs. CHAIN gives a standard system command line which may call a non-QBASIC program, though COMMON will be lost in that case.

The % sign on a compiler directive need not be in column 1.

15.4. Extensions to CBASIC

The Block IF statement has been added, allowing completely general nesting of IFs spanning as many lines as desired. Multiple cases are handled by the ELSEIF statement.

The EXIT IF statement allows direct exit from any loop, Block IF, or function.

A function argument can be an entire array or the name of a simple variable. The value of such an argument can be changed by the function.

Arithmetic expressions may use the MOD operator to compute remainders. The following functions have also been added: been added: ADRS, ASIN, COT, CSC, SEC.

An input/output list in READ, PRINT, or INPUT can include a FOR - NEXT construction.

The PRINT statement recognises two new constructions: <integer expression># for hexadecimal output, and <string expression>! to omit quote marks.

The GETFILE statement corresponds to the FILE statement; but with the full syntax of the CREATE statement.

It is possible to READ a stream file, then PRINT in it even if an end-file has not been detected.

A file or a record can be locked to assure that only one of a set of cooperating programs is accessing it at one time.

Non-disc devices, such as console and printer, may be assigned with the OPEN statement and used as files.

Line numbers may contain any number of decimal points, plus signs, minus signs, and E's, provided they begin with a digit or decimal point.

Remarks may be embedded in statements using {curly brackets}.

The CALL statement uses the name of an assembly-language program rather than an address. The program itself is in relocatable form and will load wherever LINK decides to put it; SAVEMEM is not needed. The CALL may take arguments, including the location of program variables.

Functions may be compiled in modules separate from the main program and linked to it with LINK. Variables may be shared between separately compiled modules. Functions can be written in assembly language.

The ability to read the date and time through the DATETIME function has been added.

16. Differences from earlier QBASIC releases

The following sections describe changes made between various release versions of QBASIC from the most recent to the original release. By reading this section, you will be able to establish what has changed between the current version and the last version you used.

16.1. Changes in release 3.0

This section lists changes between level 3.0 and level 2.0.

16.1.1. Transparent changes

The largest change in release 3.0 from previous versions is that the compiler was modified to generate threaded code rather than direct code. This reduces the size of object programs by from 30 to 40 percent. There is usually no measurable increase in execution time as a result of this change. A worst case benchmark, unlikely to ever be encountered in real code, shows that a degradation of 10% is the absolute maximum. In any program containing I/O or floating point, the difference is too small to measure.

Starting with release 3.0, the compiler consists of two phases: QBASIC, which compiles the program to a machine independent intermediate code; and QP2, which optimises this code, assembles it into machine instructions, and writes out the relocatable output file. This eliminates the need to call ASM after QBASIC, and results in the output code always being optimised. Previously, QBOPT had to be called to perform the optimisation, and this took a great deal of time. With release 3.0, a compilation, including optimisation, often takes half the time of an unoptimised compilation in prior releases. Since output code is always optimised, it is more compact and efficient.

The intermediate code in the temporary file will be much shorter than with previous releases. In release 3.0, this code is in a very compact format unique to QBASIC. Previously, it had to be in assembler format and took much more room to store. Thus, the TEMP1$ file need not be as large as with prior releases.

16.1.2. Nontransparent changes

The following reserved words have been added, and hence may not be used as variable names: DATETIME, FRE, MFRE.

On a stream or fixed file READ, a record with an ASCII SUB character (hex 01A) in column 1 will be treated as an end of file. This is compatible with CP/M, but different from prior releases of QBASIC. The standard Marinchip end of file character, EOT (4) is still written and recognised as before by QBASIC. This change allows programs which write their own CP/M end of file marks to work without modification.

QBOPT has been deleted. Output code is now automatically optimised exactly as QBOPT used to do it.

The compiler no longer outputs assembly code. The undocumented

practice of modifying the assembly code output by the compiler is no longer possible.

Separately compiled QBASIC subprograms linked with a main program compiled with release 3.0 should be recompiled with release 3.0. Many subprograms will work without recompilation, but there have been some changes which could result in undefined symbols in the LINK unless the subprograms are recompiled. ASSEMBLY LANGUAGE SUBROUTINES AND FUNCTIONS DO NOT HAVE TO BE CHANGED. IF THEY WORKED WITH 2.0, THEY WILL WORK WITH 3.0. It is also permissible to CHAIN back and forth between programs compiled with releases 2.0 and 3.0.

16.1.3. Extensions

The DATETIME function was added to allow retrieval of the time and date.

The FRE and MFRE functions were added to allow a program to determine the total amount of free space and the largest block of free space available.

The %DEBUG directive was added to allow compilation of line number flags into the output code. This allows printing the source line number in runtime error messages.

The %DIAGNOSTIC directive was added to control whether the source program is copied into the interpass file for debugging. This was previously always done, but now is done only if requested. This reduces the size of the interpass file 30 to 50%.

16.2. Changes in release 2.0

This section lists changes between release 2.0 and the original release, 1.0.

16.2.1. Things to watch out for

The level 2.0 compiler and library go together. Since old relocatables can't be used with the new library, conversion of a program requires recompilation of all its modules. Likewise, new relocatables may not be LINKed with old libraries.

The most important incompatibility involves the treatment of stream files: earlier levels of QBASIC, unlike CBASIC, recognised record boundaries in stream input. Level 2.0 ignores record boundaries in READing without the LINE option and is entirely compatible with CBASIC. (Earlier QBASIC manuals gave warning that this might be changed.)

QBASIC 2.0 introduces some new reserved words, which will affect any program that uses them as variable names. The new words are ELSEIF, ENDIF, EXIT, GET, LOCK, OVERLAY$, PUT, UNLOCK.

The last source language incompatibility is fairly obscure: the LINE option on the disc READ statement formerly caused the entire length of a fixed-length record to be read, including the carriage return and any garbage after it. Thus it would be possible to read arbitrary

binary data which had been packed into a pseudo-String by assembly language routines and written on disc. As the PUT and GET operations now handle binary data in a simpler way, there is no need for this feature. The LINE option now reads up to, but not including, the carriage return, just as in stream files and as in CBASIC.

Some of the library routines, especially disc input/output, are larger than before. As a result, an executable program may increase in size enough to overflow the Disc Executive file in which it was formerly stored. Because of the improved buffering, however, the total memory space used at execution time is likely to decrease, especially for large programs.

16.2.2. Extensions

Several obscure restrictions have been removed, including the lack of a Real MOD and length restrictions in the VAL function and stream input.

Here is a short list of actual extensions to the language:

- Block IF
- EXIT IF
- Record locking and unlocking in NOS/MT
- User control of disc buffering with the BUFF spec
- GET and PUT statements
- FOR - NEXT in input/output lists
- Function call by reference
- Compilation listings