# UniFLEX™ BASIC User's Manual

technical systems consultants, inc.

# UniFLEX™ BASIC User's Manual

## MANUAL REVISION HISTORY

| Revision | Date | Change |
|---|---|---|
| A | 9/80 | Original Release, Basic Version 1.0<br>under UniFLEX Version 1.01. |
| B | 10/80 | Include changes described in Product Bulletins<br>UBASIC0001 and UBASIC0002:<br>p. B-3 Add documentation for error number 42.<br>p. B-8 Description of Error 91 should be for Error 90;<br>Error 91 is not used. |
| C | 1/81 | p. B-2 Add documentation for error number 29. |

# Preface

This manual contains a comprehensive description of the Basic language as implemented under the UniFLEX™ Operating System. This version of Basic is an extension of Basic™ as originally developed at Dartmouth College. (Basic is a trademark of the board of trustees of Dartmouth College, and UniFLEX is a trademark of Technical Systems Consultants, Inc.)

This manual is organized such that elementary material is presented in the first three chapters, with the more advanced features described in later chapters. Beginning Basic programmers should master these first three chapters before proceeding to the advanced material.

Certain conventions are used throughout this manual in the description of the general form (or syntax) of statements in the Basic language. Each statement is described in general terms using the following conventions:

1. Items that are not enclosed in angle brackets (< and >) or square brackets ([ and ]) are "key words" and should be typed as shown. Spaces may be inserted within a key word to inprove readability. For example, "go to" is an acceptable form of the key word "goto".

2. Angle brackets (< and >) indicate essential items. For example:

   goto <line number>

   Here, a line number must be specified in the place indicated by <line number>.

3. Square brackets ([ and ]) indicate optional items. These items may be omitted if their effect is not desired.

Some of the discussions of the more advanced material require a knowledge of the UniFLEX Operating System. It is assumed that the programmer will have become familiar with the UniFLEX Operating System manual before attempting to use these features.

Basic is invoked by typing the command "basic" in response to the UniFLEX command prompt (++). Basic will accept one optional argument. If an argument is specified, it is assumed to be the name of a UniFLEX file that contains a Basic program to be run. Basic will automatically load and execute the program contained in that file. For example:

```
++basic inventory
```

This will cause the loading and execution of the Basic program contained in the file "inventory". If more than one argument is specified, the extra arguments are ignored by Basic; however, they are available to the Basic program as described in the "System Interface Features" chapter.

Table of Contents

# Chapter 1.

# Fundamentals of Basic

## 1.1 Lines

Each line of a Basic program begins with a line number. Lines may be numbered from 1 through 32767 and each one must have a unique number. The length of a line may not exceed 255 characters and the line must be terminated by a carriage return. When a program is executed by Basic, it starts with the smallest line number and progresses toward the largest.

When writing programs it is wise to number lines in increments of 10, 20, or more so that additional lines may be inserted during program debugging or modification.

## 1.2 Statements.

The line number is followed by a Basic statement. The first word of the statement tells Basic what action to perform.

More than one statement may be put on a line. The individual statements must be separated by a colon (:) or a backslash (\). The following are examples of multiple statements on a line.

```
120  input "Speed,Time";s,t:s=s*t:print "Dist=";d
130  a=310*x \ b=k-a \ c=a*2.9/11 \ print "CF=";a;b;c
```

Spaces and horizontal tab characters may be used as desired to make a program easier to read. (The effect of the horizontal tab character is described in the UniFLEX manual.) An example of this feature is given below. Note especially that statements 30 and 40 are equivalent except that line 40 may be a little easier to read. Line 50 demonstrates that spaces may appear anywhere in a Basic line without causing any harm.

```
30letx=32*x/3+7.3*x/2+6.54*x-.1
40 let x = 32*x/3 + 7.3*x/2 + 6.54*x -.1
50 l et a 1=123 . 5
```

## 1.3 Numbers

Numbers, also called numeric constants, can be of two types: floating point constants and integer constants. Either type may also be positive or negative.

### 1.3.1 Floating Point Constants

A floating point constant is a number that either 1) contains a decimal point, 2) is written in scientific notation as described below, 3) is larger than 32767, or 4) is smaller than -32768. All floating point values are internally represented with a seven byte (56 bit) signed magnitude mantissa and a single byte (8 bit) excess 128 notation exponent. Externally this is approximately a 17 decimal digit mantissa with a dynamic range of 10**38.

Often a more convenient form of representing large values is scientific notation. In this form, a signed mantissa is followed by "E" or "e" and the signed exponent. When Basic converts a floating point number into human-readable form, it is automatically converted to scientific notation if its magnitude is greater than or equal to 1000000 (1e6) or less than .01 (1e-2). (This range may change depending on the values set by the "digits" command, described in the second half of this manual.) Some examples of floating point numbers are:

```
98.345              -1.23e+6
3.14159265          1e-04
0.000001e6          1.0
```

### 1.3.2 Integer Constants

An integer constant is a whole number in the range -32768 through 32767 inclusive. Integers are internally represented as a 16 bit two's complement number. Some examples of integer constants are:

```
100                 32000
-12                 1
```

### 1.3.3 String Constants

Another type of constant is the character string constant. It is different from the other constants both in the way it is defined and in the way it is usually used. Character string constants will probably be seen most often in "print" and "input" statements. String constants are defined by placing any ASCII character or group of characters (a string) between single or double quotation marks. A string may be of any length from 0 to 32767 characters long. Some examples of strings are:

```
"What is your name"
"'Friday'"      (string includes single quotes)
'"really"'      (string includes double quotes)
"H"             (single element string)
""              (null string)
```

## 1.4 Variables

A variable is an item that may take on different values. For example, it can be assigned a value by the programmer and later be changed by the program during execution. The three types of variables are "floating point", "integer", and "character string". A "floating point" variable name can consist of a single alphabetic letter or a letter followed by another letter or single digit. Examples of names used to define floating point variables are A, K, G9, E1, and XX. Upper case letters are treated as distinct from lower case letters. Thus, each of following four variables would be treated as a different variable by Basic: AB, ab, aB, Ab.

The second type of variable is the integer variable. It is defined by following any "floating point" variable name with a percent sign (%). Using the same five variables as before, we could define them to be integer variables by writing them as A%, K%, G9%, E1%, or XX%.

The third type of variable is the string variable. It is defined by following any "floating point" variable name with a dollar sign ($). Using the same five variables as before, we could have defined them to be string variables by writing them as A$, K$, G9$, E1$, or XX$.

The same variable name can be used in a program for a floating point variable, a string variable, or an integer variable. For example, the variables "A" and "A$" listed above could both be used in the same program because they are considered different variables since one is a floating point variable and the other is a string variable.

It should be noted that some combinations of double letters intended to be used as variables are not valid. They are keywords in Basic and are reserved. These are: AS, FN, IF, ON, OR, PI, and TO. All upper and lower case variants of these reserved words may also not be used as variable names.

Because Basic tries to find keywords before variable names, and because spaces may be imbedded in keywords and variables names, certain combinations of two letter variables names and keywords may cause unexpected results. For example:

```
10 if a1>1e then 200
```

Basic would see the variable "1e" and the "t" from "then" as the keyword "let". The remaining "hen 200" would be seen as the variables "he" and "n2" followed by the constant "00". This is not detected when the line is entered into Basic, but it will produce an error message when the program is run.

## 1.5 Expressions

An expression is composed of numbers, variables, functions, or a combination of the preceding separated by operators.

There are four different classes of operators available: mathematical operators, logical operators, relational operators, and string operators.

The class of operators most familiar to everyone is that of the Mathematical Operators. These operators are addition, subtraction, multiplication, division, and exponentiation.

The second class is the Logical Operators. They are used to perform bit by bit operations on integer quantities and are used extensively in conditional tests and for masking. Since they operate on integer quantities, the internal "floating point" representation of some variables and constants must first be converted to integer. These conversions are done automatically by Basic.

The third class of operators are called Relational Operators. They are also used in conditional tests. They may be used in an "if" statement, for example, to determine if one quantity is greater than another. In the case of trying to "relate" a floating point number and an integer, the integer is first converted to floating point and then the relation is performed.

The last class of operators is the class of string operators. This class includes the concatenation operator (for joining strings together) and the relational opertors (for comparing strings).

Each of the classes of operators will now be described separately.

### 1.5.1 Mathematical Operators

The mathematical operators are those used for arithmetic. They are summarized in the following table.

Mathematical Operators

| Symbol | Example | Meaning |
|---|---|---|
| + | X+Y | Add X and Y |
| - | X-Y | Subtract Y from X |
| * | X*Y | Multiply X and Y |
| / | X/Y | Divide X by Y |
| ^ | X^Y | X to the Yth power |

The symbol "**" may also be used for the exponentiation operator. Thus, X**Y is also "X to the Yth power".

When an arithmetic expression containing several of these symbols is to be evaluated, it is processed using the following priority scheme.

1. Exponentiation
2. Unary Minus
3. Multiplication and Division
4. Addition and Subtraction

This means that when Basic is evaluating an expression containing a mixture of mathematical operators, it will first do the exponentiation, then take into account any unary minus signs (such as -3.4 or -A). Next it will do multiplications and divisions then, last of all, it does additions and subtractions. When signs of equal priority are encountered, it does the left one first since Basic evaluates expressions from left to right. This order can be altered by the use of parentheses. Basic evaluates quantities in parentheses first and, in the case of nested parentheses, it starts with the innermost set and works its way out. They can and should be used anytime there is a doubt as to how the expression will be evaluated.

When one of the operands is an integer and the other is a floating point number, (called a "mixed mode" operation), the integer is converted to floating point before the operation is performed. When mathematical operations are performed on integers, the rules for integer arithmetic are used. These are discussed later in the section: "Integer Arithmetic".

The exponentiation operator is an exception to these rules and one of two things can happen. First of all, if the "base" (base ^ power) is an integer, it is converted to floating point. Note that the "^" operator always returns a floating point result. Next if the "power" is an integer then a fast algorithm is called that "essentially" performs repeated multiplication. On the other hand, if the "power" is a floating point value, then the following algorithm is used:

$$x^y = \exp(y * \log(x)),$$

where "exp" is the "exponential" function. In this case, "x" cannot be negative because the logarithm of a negative number does not exist.

#### 1.5.1.1 Integer Arithmetic

When mathematical operations are performed on two integers, the result is always an integer. Thus, the integer 5 plus the integer 6 yields the integer 11. If the mathematical operation results in a value that is too large to be an integer (greater than 32767 or less than -32768), then an error is generated (Error 109). Thus, multiplying the integer 4096 by the integer 10 will generate an error since the result (40960) is too large to be represented as an integer.

When division is performed on integers, any remainder is thrown away. Thus, dividing the integer 5 by the integer 2 gives 2 as a result. This is true even if constants are used; for example, the Basic statement

```
10 print 5/2
```

will print a 2 because both the dividend and the divisor are integers. If either the dividend or the divisor is a floating point number, then the result is a floating point number and the remainder is not discarded. For example, each of the following statements prints a result of 2.5.

```
10 print 5./2
20 print 5/2.
30 print 5./2.
```

### 1.5.1.2 Round-off Error

A computer uses the binary (base 2) number system while humans use the decimal (base 10) number system. There are many numbers which cannot be exactly represented, either to a human, or inside the computer. For example, 1/3 (base 10) represented in decimal and 0.1 (base 10) and 0.01 (base 10) represented in binary are very common values; yet they are repeating numbers and cannot be accurately represented in a finite amount of space, ie. 1/3 = 0.33333... in base 10 and 0.1 decimal = 0.000110011001... to the computer in binary. Because values such as 0.1 and 0.01 cannot be represented accurately in binary, this inaccuracy, as small as it may be, is carried or propagated through successive arithmetical operations. That is, by adding, subtracting, multiplying, etc. the value 0.1 (or any other value not accurately represented) a number of times, the small inaccuracy or error grows larger with each arithmetical operation. Therefore, to the computer, 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 does not quite equal 0.8! In situations where you may have several hundreds or thousands of computations, this error (called round-off error or truncation error) could result in false results or in improper testing in "if" statements. For example, if you were testing for the equality of a computed result and a constant, they may not be exactly equal due to round-off error. For example:

```
10 for i%=1 to 1000
20 let f=f+0.1
30 next i%
40 if f=100. then print "EQUAL!"
50 end
```

This program, which merely performs an addition of 0.1 one thousand times, will not indicate that the final result of the additions is equal to 100. In general, one should not test for strict equality between floating point numbers. The proper programming technique is to define a small tolerance value and consider the numbers to be equal if their difference is smaller than this tolerance value. For example:

```
10 for i%=1 to 1000
20 let f=f+0.1
30 next i%
40 if abs(f-100.) < 1e-6 then print "CLOSE ENOUGH!"
50 end
```

In this example, a tolerance value of 1e-6 was chosen. The "abs" is the absolute value function and is necessary in case the difference in the values being compared results in a negative number. This program could also have been written using the "approximately equal" relational operator, described in the section on relational operators.

### 1.5.2 Logical Operators

When Logical Operators are used on one or two numbers they perform the desired operation on the corresponding bits of the number or numbers. If, for example, we assume "a" and "b" are equal to the following binary quantities:

```
a=(1100101111110110)
b=(0111010111100100)
```

Then:

```
not a  =(0011010000001001)
a and b=(0100000111100100)
a or b =(1111111111110110)
```

It can be seen that these are bit-by-bit operations. These operators, when used like this, operate on one or two numbers to give a single numeric result.

The logical operators have a totally different effect when they are used in an expression that is the test condition of a conditional ("if") statement. In this case, the expression is being logically evaluated (not arithmetically evaluated) to see if it is true or false. An expression that is evaluated and determined to be true has a non-zero value and one that is determined to be false has a value of zero. A statement such as:

```
22 if a>0 and a<10 then go to 40
```

will branch to statement 40 if and only if "a" is between 0 and 10. The logical operator "and" specifies that the condition "a>0" be true and the condition "a<10" also be true. The following is a list of the available operators.

Logical Operators

| Symbol | Example | Meaning |
|---|---|---|
| not | not x<br><br>x \| not x<br>0 \| 1<br>1 \| 0 | When operating on integers, this operator simply switches each bit in the binary representation with its complement (1's are replaced with 0's and 0's are replaced with 1's). |

and x and y

| x \ and | y = 0 | y = 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

The result of this is to assign to each bit of the result a "1" if each of the corresponding bits of the two arguments is a "1". When "and" is used in a conditional test, then both "x" and "y" in the example must be true for the logical "and" of them to be true.

or a or b

| x \ or | y = 0 | y = 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

This leaves each bit of the resulting integer a "1" if either of the corresponding bits of "a" or "b" is a one. When used in a conditional test, the test will yield true if either "a" or "b" is true.

### 1.5.3 Relational Operators

As the name implies, this group of operators tests the relation of variables to other variables or constants. The relational symbols recognized by Basic are:

Relational Operators

| Symbol | Example | Meaning |
|---|---|---|
| = | X=Y | X is equal to Y |
| <> | X<>Y | X is not equal to Y |
| < | X<Y | X is less than Y |
| > | X>Y | X is greater that Y |
| <= | X<=Y | X is less than or equal to Y |
| >= | X>=Y | X is greater than or equal to Y |
| ~ | X~Y | X is approximately equal to Y |

These are often combined with the Logical Operators to perform complex tests. The statement:

```
660 if a=0 or (c<127 and d <> 0) go to 100
```

will cause a branch to statement 100 if "a" is equal to zero or if both "c" is less than 127 and "d" is not equal to zero.

#### 1.5.3.1 The "approximately equal" Operator (~)

The "approximately equal" operator may be used to compare floating point values. Two floating point values are considered approximately equal if their difference is "small" compared to their respective values. The value specified by the "digits" statement (described in the second half of this manual) determines what "small" means. The "digits" statement specifies a certain number of significant digits. The larger this number, the closer together the two values must be before they are considered approximately equal. A complicated method is used to determine when two values are approximately equal. This is discussed in the description of the "digits" statement. For now, it suffices to say the the "approximately equal" operator may be used to compare floating point values when there is a chance that round-off error has perturbed the values.

### 1.5.4 String Operators

The string operators consist of the concatenation operator (+) and the relational operators. The "+" operator will concatenate two strings (join them together) to form a new string. The relational operators, when applied to string operands, indicate alphabetic sequence. If one string is "less than" another, it implies it would appear before the other if sorted into alphabetical order. The ASCII collating sequence is used to define "alphabetical order". In this sequence, blank is smaller than numbers which are, in turn, smaller than letters. Upper case letters are smaller than lower case letters. In any string comparison, trailing spaces are ignored. If two strings of unequal length are compared, the shorter string is padded with trailing spaces to make it equal in length to the other. A string of zero length (null string) is considered to be completely blank and is less than any string of length greater than zero unless the string is all spaces, in which case the two are considered equal. All of the standard arithmetic relational operators may be used in connection with strings.

### 1.5.5 Overall Operator Precedence

The overall operator precedence is shown in the table below. The operator at the top of the list has highest precedence, while the one at the bottom has lowest. Operators of equal precedence are evaluated left to right.

1. ()  Expressions enclosed in parenthesis
2. ^  Exponentiation
3. -  Unary minus
4. * /  Multiplication and division

| | | |
|---|---|---|
| 5. | + - | Addition and subtraction |
| 6. | | Relational operators |
| 7. | not | The "not" operator |
| 8. | and | The "and" operator |
| 9. | or | The "or" operator |

# Chapter 2.

## Elementary Commands

Commands are those instructions which may be given to Basic in response to the "Ready" prompt. They cause immediate action to take place. This section discusses those commands which are most frequently used when writing Basic programs. Additional, more advanced, commands are discussed in the second half of this manual. In the following documentation, the commands are given in lower case. Basic will accept either upper case or lower case for the commands.

### exit

The "exit" command causes Basic to terminate, returning control to the UniFLEX Operating System. An optional integer argument may be specified which will be used as the UniFLEX termination code (see the UniFLEX manual for a description of termination codes). Some examples:

```
exit          (return to UniFLEX)
exit 8        (return a code of 8)
```

### list

"List" is used to display lines of a program. If "list" is typed with no arguments, the entire program is listed. If a single line number is specified as an argument to the "list" command, only that line is listed. A range of lines may be listed be specifying the starting and ending line numbers, separated by a hyphen. Some examples of the "list" command are:

```
list          (list entire program)
list 10       (list line 10)
list 50-80    (list lines 50 through 80)
```

### load

The "load" command is used to load a Basic source program from disk. The file may have been created as the result of a Basic "save" command, or it may have been created by another program, such as the text editor. The name of the file containing the Basic source program is specified as an argument to the "load" command and must be specified in quotation marks. Either single or double quotation marks may be used. The name may include whatever UniFLEX path information is necessary to locate the file. Some examples of the "load" command:

```
load "ledger"
load 'usr/source/basic/program'
```

Note that a program which has been saved with the "save" command should load back in without any error. A program that was created by the text editor may contain errors which Basic will detect during the load. If this occurs, Basic will terminate the loading process at the line preceding the line containing the error.

new

When the "new" command is executed, it deletes the current program. After executing this command, you are ready to start typing in a new program.

run

The "run" command instructs Basic to begin execution of the current program. When you run a program, all variables are initialized to zero and "data" statements are restored. ("data" statements are discussed later on in this manual).

An argument may also be specified when using the "run" command. The argument is the name of a file on the disk which contains a Basic program. Using the "run" command with an argument tells Basic to delete any existing program in memory, load the program from the specified file, and then run the program. If a file name is specified, it must be in quotation marks (either single quotation marks or double quotation marks). The name may include such UniFLEX path information as may be required to locate the file. For example:

```
run 'usr/source/basic/program'
```

save

"Save" is used to store a Basic program on disk. The file name under which to save the program is specified as an argument to the "save" command. The name must be in quotation marks (either single quotation marks or double quotation marks), must include such UniFLEX path information as may be required to put the file in the desired directory. For example:

```
save "ledger"
save "usr/source/basic/program"
```

If a file with the same name already exists, the following prompt is issued:

```
Delete existing file?
```

Answering this prompt with a "y" followed by a carriage return will cause the existing file to be deleted, and a new file, containing the saved program, to be created. Answering this prompt with an "n", a "control-d", or a "control-c" will cause the existing file to not be deleted, and the "save" command to be aborted. A Basic program that has been saved is stored as a UniFLEX text file and may be manipulated by other programs, such as the text editor.

control-d Typing a "control-d" in response to the "Ready" prompt has the same effect as the "exit" command. Basic will return control to the operating system.

# Chapter 3.

# Elementary Basic

This chapter describes the simpler forms of some of the most frequently used Basic statements. Enough information is given so that these statements may be used to solve many problems. Once the programmer has become familiar with these simpler statements, the more advanced forms and techniques, described in subsequent chapters, may be studied.

## 3.1 Remarks - The "rem" statement.

The "rem" statement is used to place remarks and comments throughout your program. The form of the "rem" statement is:

```
<line number> rem [message.... ]
```

For example:

```
40 rem This could be the name of the program
50 rem
```

Anything following the "rem" statement is ignored, including multiple statement per line characters, so remarks should be the last statement on multiple statement lines. Any character is legal following the "rem" and you can even leave the rest of the line blank as in statement 50. It's very likely you'll find "rem" statements at the beginning of programs to give the name of the program and what it does. Also they are often placed at different points in programs to explain how unclear or complicated sections of the program work.

## 3.2 Assignment - The "let" Statement

The "let" statement assigns a value to a variable. The form of the "let" statement is:

```
<line number> let <variable>=<expression>
```

For example:

```
10 let x=3.5
25 let h1=27.2*h1/(5.4e7-x)
70 let da$="MONDAY"
80 y=12.314                    (implied "let")
90 y%=y%+1
95 z=y%                        (type conversion)
96 y%=a+2.4*g1
```

Any variable can be assigned a value using this statement. The value can be a constant as in statement 10 or may be a complex expression as in statement 25. Notice that statements 80 through 96 are missing the word "let". This is no accident but is an example of an implied "let". This is a convenience feature, and any "let" statement may be written this way. In line 95 note that the assignment variable is a different type than that of the resulting expression. In this particular example, the integer expression is converted to floating point and the assignment is performed. In line 96 just the inverse is performed. The floating point expression is converted to an integer and then the assignment is performed. In this case, any fractional part of the result is discarded.

## 3.3 Input and Output

Data may be given to the program either through an array of pre-defined constants stored in the program itself, or by entry from the keyboard. The program may also display information to the user as it is running. In this section, the most simple forms of input and output statements are presented. More advanced forms are discussed in the second half of this manual.

### 3.3.1 The "read", "data", and "restore" Statememts

The "read" and "data" statements work together. Neither statement is used without the other. The "read" statement is used to read data from a "data" statement. The format of the "read" statement is:

<line number> read <variable> [,<variable>,<variable>,... ]

For example:

```
200 read v,a1,cc
```

The format of the "data" statment is:

```
<line number> data <number> [,<number>,<number>,... ]
                   <string> [,<string>,<string>,... ]
```

For example:

```
50 data -3.556E-5,0,2,4.59E11
60 data April, May, Thats all
70 data "  100"," 1000","10,000"
```

The "data" statement specifies information that will be read in by the program. The "read" statement specifies which variables are to receive the data. The data is read in from left to right and begins with the first piece of data listed. Each time a "read" statement is encountered in the program, the next item in the data list is read. Statement 200, for instance, will read the next available piece of data and assign it to the variable "v". The next data entry will be assigned to "a1" and the next one will be assigned to "cc". The "read" statements will begin taking data from the first "data" statement that appears in the program and, when it has read all that is in this statement, it will drop down to the next data statement and so on. Subsequent "read" statements will pick up data where the last "read" statement left off.

The "data" statement cannot appear in multiple statement lines. If a string is needed that contains an embedded comma or it is preceded by a space or spaces then it must be enclosed in quotation marks as in statement 70 above. (Note that the values in statement 70 are strings, not numeric data.)

It is important that the program not try to read beyond the last data item in the last "data" statement because an error (number 31) will be issued if this is attempted.

The "restore" statement is used to reinitialize "data" statements so that the data in them may be re-read by "read" statements. The format of the "restore" statement is:

```
<line number> restore [ <line number> ]
```

For example:

```
440 restore
500 restore 60          (restore to line 60)
```

When a "restore" is executed it causes the next available data entry to be the first one appearing in the first "data" statement or, in other words, it resets the "next available data item" pointer to the beginning of the data that appears in the "data" statements of the program. Optionally one can specify a line number at which to restore such that the next read will start at that line looking for a data statement

instead of searching from the beginning of the program. Thus, in the second example above, the next data item will be the first one in statement 60.

### 3.3.2 The "input" Statement

The "input" statement is used to read data from the user's terminal keyboard and enter the values into Basic variables. The format of the "input" statement is:

```
<line number> input ['string',] <variable list>
                    ["string",]
```

For example:

```
556 input "What is your age",a
600 input '"Enter a number"',n
650 input W,X,Y,Z
700 input 'Give me an integer',i%
```

The "input" statement, when executed, prints out the character string enclosed in single or double quotation marks (if there is one). Then it prints a question mark and waits for the user to enter the value or values requested. If no string has been specified, only the question mark is printed. If a string was specified, either single or double quotation marks may be used to delimit the string. If one form of quotation mark is to be part of the actual string, the other form should be used to enclose the string as in line 600 above. In statements that request more than one variable, the entries from the keyboard must be separated by commas and the last one is followed by a carriage return. If you do not enter as much data as the "input" statement requests, Basic will respond to your carriage return with "?" which is a prompt for more input. If too many entries are made then the extra entries will be ignored.

The user may respond to an "input" statement by typing a "control-c". If this is done the program will stop running and Basic will be in a mode to accept new commands. The program will resume execution at the "input" statement if the "cont" command is typed. This is discussed in more detail under the description of the "cont" command.

### 3.3.3 The "print" Statement

The "print" statement is used to display information at the user's terminal. The format of the "print" statement is:

```
<line number> print  [variable, string ;... ]
                     [ string ;variable,... ]
```

Some examples:

```
10 print                          (prints blank line)
30 print "WHAT"                   (prints "WHAT" on the terminal)
45 print "Speed=";s
50 print a,b;x;y
75 print "Solution=";h*g/3.2      (expression)
```

The information following the "print" statement can be left out or it can be an arbitrarily arranged string of constants, variables, expressions, and character strings. If the argument string is ended with a comma or semicolon, then the next "print" statement will continue where the previous one leaves off. Otherwise, the next "print" statment will start on a new line.

The individual items in the argument string are separated by either a comma or a semicolon but a short explanation of Basic's printing format is necessary before the significance of each one can be understood. In the following discussion, we will refer to a "cursor" as indicating the position at which the data is going to be printed. If the user is printing to a CRT terminal, then this should correspond to the actual cursor being displayed. If the user is using a hard-copy terminal, then "cursor" should be interpreted as the "print head" of the terminal.

The maximum length of a line is determined by a "width" value which is set by the "width" statement. A width of zero means indefinite length. Basic divides each output line into fields with each field containing 16 character positions (columns). Columns on the print line are numbered starting with column zero. When arguments are separated by commas, the comma after one item will cause the printer to jump to the beginning of the next field before printing the next value or string. This positioning takes place as soon as the comma is seen. Thus, a comma will cause the cursor to jump to column 16, 32, 48, 64, etc., depending on the current location of the cursor. (Remember that the columns are numbered starting at zero, so that the first field includes columns 0 through 15, the second, 16 through 31, etc.) If the cursor is already closer than 16 columns to the end of the line (as defined by the "width" statement) when a comma is seen, then a new line is started, and the data will be printed in the first column of the new line. You can type two or more commas next to each other. This will cause the cursor to skip one field just as three commas would cause two fields to be skipped and so on.

When items are separated by a semicolon, they will be printed next to

each other. There will still be a space or two between them if one or more of the adjacent arguments are numbers. This is because numbers are printed with one trailing space and a leading space if the number is not a negative number (negative numbers have a leading minus sign). The last necessary bit of information is that character strings do not have leading or trailing spaces added. By knowing how to use commas and semicolons and by being aware of the format used to print numbers and character strings, you can tailor printed output to suit most needs. More sophisticated output formatting is discussed under "print using" in the second half of this manual.

### 3.3.4 The "width" Statement

The "width" statement is used to control the number of characters that are printed on a line by "print" statements. The format of the "width" statement is:

<line number> width <size>

For example:

```
10 width 80
20 width 0        (sets indefinite width)
```

The first example sets the width to 80 columns. The second example sets an indefinite width. In this second case, Basic will never automatically start a new line until a "print" statement that does not end in a comma or semicolon is seen. It is possible for a data item, such as a string, to print beyond the declared width. All that is important is for the data item to start at least 16 columns away from the declared width; any closer and a new line will be started before printing the data.

## 3.4 Unconditional Transfer of Control - The "goto" Statement

The "goto" statement simply causes a branch to the specified line.

<line number> goto <line number>

For example:

```
100 goto 50
```

When statement 100 (in the example) is executed, it forces a branch to line 50. The "goto" statement should only be used in the last position in lines containing multiple statements. This is because this statement

always causes a branch and any statements following it would never be executed.

## 3.5 Conditional Statements

Conditional transfers of control cause a jump to another line, or execution of a specific statement, to take place only if certain specified conditions are satisfied. We have seen examples of this in some of the previous examples in which an "if" statement has been used.

### 3.5.1 The Conditional Jump - "if-goto"

The "if-goto" statement is a means of causing a jump to another line if certain specified conditions are met. The form of the "if-goto" statement is:

```
<line number> if <expression> goto <line number>
```

Some examples are:

```
5 if C=1 goto 110
1000 if a>b and f<>6.0 goto 300
```

The expression is evaluated and if it is true (has a nonzero value), Basic jumps to the line number following the "goto". If the expression is not true, the next sequentially numbered statement after this "if-goto" will be executed. In other words, this statement would not cause Basic to take any action if the expression is false.

### 3.5.2 Conditional Execution - The "if-then" Statement

The "if-then" statement is used to cause a jump to another line or execution of another statement only if certain specified conditions are met. The form of the "if-then" statement is:

```
<line number> if <expression> then <line number>
                                   < statement >
```

Some examples of the "if-then" statement are:

```
30 if A+B=10 then 50
80 if A=0 then print "A=0"
99 if X=Y then if X>Z goto 40
```

This works similar to the "if-goto" statement except that you do not necessarily have to branch to another line. If, as in the example, you follow the expression by "then 50" this has the same effect as "goto 50". Each will cause a branch to line 50 if the expression is evaluated as being true.

In line 80 of the example, we specified the execution of the statement 'print "A=0"' if the expression is true. Executing another statement is an alternative to jumping to another line. The statement could even be another "if-goto" or "if-then" statement as in line 99 of the example.

If multiple statements are included after the "then", they are all considered to be under the control of the "then". If the expression is false, none of the statements on that line will be executed.

A more advanced form, the "if-then-else" statement, is described in the second half of this manual.

## 3.6 Program Loops

Many times, in a program, it is desired to perform a sequence of statements repeatedly. One way of doing this is to duplicate those statements in the program the required number of times. This, however, is quite cumbersome. Another way is to have one copy of the statements and use a conditional statement ("if-goto" or "if-then") to jump back and re-execute the statements the proper number of times based on the value of a counter. There is a more convenient way of doing this, and that is with "for" and "next" statements.

In many cases, when a program has need for loops, it also has need for a way to store and process data in arrays. Indeed, it is usually very cumbersome to process arrays of information without loops. This section also describes how to declare and process arrays of information.

### 3.6.1 The "for" and "next" Statements

The "for" and "next" statements work together to define and execute a seqment of a Basic program that is to executed several times in succession. The form of the "for" statement is:

<line number> for <variable>=<expr1> to <expr2>[step<expr3>]

Some examples of "for" statements are:

```
73 for H%=3 to 600
88 for B1=3.2*(X-7) to 200+X step 5.5
116 for I%=10 to -10 step -1
```

The form of the "next" statement is:

<line number> next <variable>

The "next" statement is only used in a program with a companion "for" statement. The variable specified in the "next" statement is the same one that is in the "for" statement with which this "next" is used. The "for-next" pair specifies the boundaries of the loop. The "next" statement is just a signal to the "for" statement that it has reached the end of the code it is to execute. Here is an example of a "for-next" loop that prints "Hello" ten times.

```
10 for I%=1 to 10
20 print "Hello"
30 next I%
```

When Basic executes a "for" statement, this will cause all the following instructions to be executed until a "next" is reached. Execution then loops back to the "for" and if the condition specified in the "for" is still true, then the statements will all be executed again. Once more control jumps back to the "for", and this cycle continues until the test finally fails, at which time execution resumes following the "next" statement. The "variable" in the "for" statement is used to keep track of how many trips or loops have been made through the "for-next" statements. The variable is initially assigned the value of "expr1" and each time the "next" statement loops back to the "for" statement, "expr3" is added to the current value of the variable. Since "expr3" can be positive or negative, this can have the effect of incrementing or decrementing the value of the variable. If no "step" is specified, then a value of positive one is assumed. When no step is specified (+1 assumed), or when a positive step is given, then the test that is specified in the "for" statement is determined to be true as long as the value of the variable is not greater than that of "expr2". When a negative step is specified, the test yields true as long as the variable is not less than "expr2". Once again, the statements between a "for" and "next" will always be executed a first time but subsequent executions only occur while the index variable in the "for" statement is within the bounds of "expr2".

When a floating point variable is used for a control variable, there is a possiblilty for round-off error to occur when the control variable is incremented or decremented by the step size. This is especially likely to occur if the step size is not a whole number. The user should be aware that under these circumstances, the loop may run for one iteration more or less than would normally be expected.

"For" Statements may be nested ("for" statements located within other "for" statements), but they can't use the same index variable. The number of levels of nesting is limited only by the available memory space. "For" statements can be exited abnormally by using a "goto" statement in the statements between the "for" and "next". They should not be entered like this unless they have previously been left abnormally. If you enter a "for" loop like this, the results are unpredictable and probably not what is desired because no initial value for the index variable has been specified.

It is to the user's advantage to use integer "for-next" loops whenever possible. Integer "for-next" loops execute approximately 3 times faster than floating point and typically use 12 bytes less storage.

### 3.6.2 Arrays, Subscripted Variables, and the "dim" Statement

Any type of variable can be subscripted (dimensioned) using the "dim" statement. A subscripted variable is also called an array variable. A variable is dimensioned in a "dim" statement by following the variable name with an integer that is enclosed in parentheses or two integers that are also enclosed in parentheses but separated by a comma. More formally, the format of the "dim" statement is:

```
<line number> dim <variable 1> (n) [,<variable 2> (m),.. ]
<line number> dim <variable 1> (k,l) [,<variable 2> (m,n),.. ]
```

An example of a "dim" statement is:

```
20 dim x(20),Y(30),z(30,40)
```

The numbers specified in parentheses are the largest values that a subscript may assume. These largest values may not exceed 32767. In the case where two dimensions for a variable are specified, the product of the two maxima may not exceed 32767. All subscripts start at 0. To use a subscripted variable you write it just as it appeared in the dimension statement except that you may use any legal expression to describe the subscripts. For example, suppose a one dimensional array has been created in a program with the statement:

```
100 dim x(4)
```

This will create 5 new variables that are called:

x(0), x(1), x(2), x(3), and x(4)

They may be used in program operations just as they are written above or the "subscript" could be a variable such as x(A) where "A" has a value of 0, 1, 2, 3, or 4. The subscript could also be an expression such as x(A+2). When this is encountered in a program, the subscript expression

is evaluated using the current value of "A". Basic will also support two-dimensional arrays.

In another example, a two-dimensional array defined by the statement:

```
30 dim X(3,2)
```

specifies a four by three element matrix. This matrix has the following elements:

```
X(0,0)  X(0,1)  X(0,2)
X(1,0)  X(1,1)  X(1,2)
X(2,0)  X(2,1)  X(2,2)
X(3,0)  X(3,1)  X(3,2)
```

Just as with the one-dimensional array, this can also be used in a program with subscripts that are expressions.

The same variable name can be used to specify a non-dimensioned and a dimensioned array. Basic will consider these to be separate but one-dimensional and two-dimensional arrays cannot share the same name. All dimensioned variables must be declared in a "dim" statement before they can be referenced.

Whenever possible integer arrays should be used. For example, assume you have a two dimensional array of of 200 elements with dimensions (24,3). (Remember all arrays in Basic start with indices of 0; so this is really an array containing 25*4 elements.) If the array is of type integer then it will need a total of 2*200 = 400 bytes of storage. On the other hand if the array is of type floating point then it will need a total of 8*200 = 1600 bytes of storage. One can easily see that a large floating point array can easily use all of memory and result in an error #80.

### 3.6.3 Variable Dimensions

The "dim" statement, as described above, also accepts an expression to indicate the maximum subscript. For example:

```
10 let X2=2
20 let Y2=3
30 dim A(X2*5,Y2*10)
```

This declares the array "A" to be an 11 by 31 array (remember, all subscripts start at 0). However, once an array is dimensioned, it cannot be re-dimensioned. If the expression does not result in an integer value, the result is automatically truncated to an integer before it is used as the dimension.

## 3.7 Functions

Since, in programming, there is often a need for many of the same functions to be used over and over again, several of the more common ones have been included as part of Basic. The supplied functions are divided into five groups which are: trigonometric, mathematical, character, output, and system. The first four categories will be covered in this section. The system functions will be covered in the second half of this manual.

Most of the functions require little explanation for what they do but a few words are necessary on the way that the functions in general are called. To call a function it is necessary only for its name and the required arguments to appear. For example, when the statement:

```
100 y=sqr(9)
```

is executed, "y" will have a value of three since "sqr" is the Square Root function.

There are also character functions such as "chr$(i%)" which return a character string.

There are some functions which do not require arguments, such as "pi", "second", "date$", and some others. These should be used without arguments in a Basic program. For example:

```
10 c=d*pi
```

Also discussed in this section are user-definable functions and the "randomize" statement, which is related to the random number function, "rnd".

### 3.7.1 Trigonometric Functions

For each of the trigonometric functions, the accuracy of the value returned is dependant on the magnitude of the argument passed to it. More accurate values will be returned by these functions when arguments with smaller magnitudes are used. In general, the trigomentric functions have an accuracy of thirteen and one-half digits.

atn(x) Returns the arctangent of x, in radians. The value returned will be between -pi/2 and pi/2, where pi/2 is approximately equal to 1.5707963267948966.

cos(x) Determines the cosine of the angle x. The argument x is assumed to be in radians.

sin(x) This calculates the sine of the angle x where the argument is assumed to be in radians.

tan(x) The tangent of the argument x is calculated by this function. x is assumed to be in radians.

### 3.7.2 Mathematical Functions

The logarithmic and square root functions are generally accurate to sixteen decimal places.

abs(x) The absolute value of x is returned by this function. If x is positive, then the value returned is x, and if it is negative, the value returned is -x.

exp(x) The mathematical operation e^x (e raised to the x'th power) is peformed. Here "e" is the base of natural logarithms and is approximately equal to 2.718281828459045. The maximum allowable value of x is 88.02969193111306. An argument any greater than this will result in overflow and error 102 will be issued.

int(x) The value returned in the largest integer that is not greater than x. This function is often called the "floor" of x. Several examples of the values returned are shown below.

```
int(70) =70          int(-.01)=-1
int(5.7)=5           int(-3)  =-3
int(0)  =0           int(-4.2)=-5
```

log(x) This is the natural logarithm (to the base "e") of the number x. This is the only logarithm supplied with Basic, but it is the only one needed because logarithms to any other base can be calculated from it. The following formula is used to translate to logarithms of other bases where the logarithm to the base B is desired.

log of x to the base B = log(x)/log(B)

Probably the most frequent use of this will be to convert to common logarithms (base 10). As an example suppose we wanted to find the common log of the number 327. The following expression will accomplish this.

```
Common log of 327 = log(327)/log(10)
```

pi — This returns the value of "pi" = 3.1415926535897933. Note that this function has no arguments.

rnd(x) — The function returns a random number that has a value between zero and one. The programmer can use this to generate random numbers between any desired limits using the formula:

```
Random Number=(ML-MS)*rnd(0)+MS
```

Where ML is the larger number and MS is the smaller number. The resulting number that is generated will range from MS to ML.

The argument x has an effect on the number that is generated according to the following rules.

- x<0 A new series of random numbers is started. For different negative values of x, a different sequence is started each time, but if the argument retains the same value, the function will keep starting the same random sequence so the value returned will be the same each time the function is called. (See also the description of the "randomize" statement for another method of starting a series of random numbers.)
- x=0 Causes the function to generate a new random number when it is called. This is the argument that will normally be used with the "rnd" function.
- x>0 This returns the last random number that was generated.

second — This function returns the number of seconds elapsed since midnight, January 1, 1980. Note that this function has no arguments. The "second" function may be used to time parts of a program by calling it before and after the program segment to be timed, and then subtracting the two values. The value returned by "second" is a floating point number and may not be assigned to an integer variable as this would cause an overflow.

sgn(x) — This is the sign function. This is also called the "signum" function. It returns "1" if the argument is greater than zero. A zero is returned if the argument is zero and a minus one is returned if the argument is negative.

sqr(x) — The Square Root function returns the square root of the argument x. If the argument is negative, error message 107 will be issued.

### 3.7.3 Character Functions

The character functions are those that either produce a string result or operate on string arguments, producing a numeric result. In the following descriptions, if an argument ends in a dollar sign, it must be a string. If an argument ends in a percent sign, it should be an integer, but a floating point value is also acceptable (the value will be truncated before being used). An argument that has neither a dollar sign nor a percent sign must be either an integer or a floating point number.

Function names that end in a dollar sign produce string results; all others produce numeric results.

asc(x$) — The argument x$ is a string expression. The value returned by the function will be the ASCII numeric value of the first character in the string. Zero will be returned if the argument is the null string.

chr$(i%) — This returns a single ASCII character (a one character string) whose ASCII value is the argument i%. The argument "i%" must be a number within the limits: 0 <= i% <= 255.

clock$ "clock$" returns the current 12-hour clock time in the format: hh:mm:ss xx, where "xx" is "AM" or "PM". Midnight is considered 12:00:00 AM and noon is considered 12:00:00 PM. Note that this function does not have an argument.

clock$(x) The "clock$" function, with an argument, returns the 12-hour clock time represented by the argument "x". The format of the time is that same as that returned by clock$ when called with no argument. The argument is the time specified in terms of elapsed seconds since midnight, January 1, 1980.

date$ "date$" returns the current date in the format: dd-Mmm-yy. The month portion "Mnn" is the first three letters of the name of the month. Note that this function does not have an argument.

date$(x) The "date$" function, with an argument, returns the date represented by the argument "x". The format of the time is the same as that returned by "date$" when called with no argument. The argument is the time specified in terms of elapsed seconds since midnight, January 1, 1980.

hex(x$) The "hex" function converts a hexadecimal character string into its decimal equivalent. If the statement:

```
180  print hex("100")
```

is executed, then the value printed would be "256" which is the decimal equivalent of the hexadecimal value of 100. The conversion stops when the first non-hexadecimal digit is encountered. This implies that there may not be leading spaces in the string.

inch$(i%) The "inch$" function inputs a single character from the file or device specified. The argument passed specifies the internal file channel. (File channels are discussed in the second half of this manual when disk files are introduced.) A channel number of 0 is the user's terminal.

instr(i%,s$,p$) The "instr" function searches for sub-string p$ in the string s$. The first argument specifies the number of the character in string s$ at which to start the search (the first character in a string is considered character

number 1). "Instr" returns an integer value specifying at which character the sub-string started. If the sub-string was not found then zero is returned.

left$(x$,i%) Character function "left$" returns a string that consists of the i% left-most characters of the string x$. The value of i% must be be a positive number less than 32,767. If the argument contains fewer than i% characters, the entire string is returned. If the character count, i%, is zero, the null string is returned.

len(x$) "len(x$)" returns with the number of characters in the string x$. All characters in the string are counted, even spaces and non-printing characters. A null string returns a count of zero.

mid$(x$,i%) This returns a character string that is a portion of the string x$. The returned string starts at position i% in the string x$ and includes everything until the end of the string. Position i% must be a positive number less than 32,768 or an error will result. If the starting character position is beyond the end of the string, the null string is returned.

mid$(x$,i%,j%) With a three variable argument, "mid$" functions the same as it does with a two variable argument except that the returned string only includes j% characters of the string x$. If the count specified is zero, the null string is returned. If the starting character position is beyond the end of the string, the null string is returned.

right$(A$,i%) Returns a character string that is the rightmost i% characters of the string A$. If the value of i% is greater than or equal to the length of the string x$, then the entire string is returned. The count, i%, must not exceed 32767. If a count of zero is specified, the null string is returned.

string$(A$,i%) The result of this function is a string that consists of i% copies of A$ concatenated together. For example:

```
10 B$=string$("XY",5)
20 s$=string$(" ",25)
```

The first example results in B$ being set to

"XYXYXYXYXY". The second example set s$ to a string containing 25 spaces.

str$(x) This will return a character string that represents the numerical expression x. In other words, it takes a number and transforms it to a character string. The string is constructed just as it would be printed, honoring the "digits" statement, if any, and including the leading space or minus sign and a trailing space.

time$ time$ returns the current date and time. The format is best described through an example:

13:24:44 Thu May 08 1980

Note that the time is a 24-hour time. Note also that this function does not have an argument.

time$(x) The time$ function, with an argument, returns the date and time represented by the argument "x". The format of the time is the same as that returned by time$ when called with no argument. The argument is the time specified in terms of elapsed seconds since midnight, January 1, 1980.

val(x$) "val(x$)" does just the opposite of "str$(x)". "val(x$)" takes a numerical character string (a string composed entirely of numbers, plus or minus sign, and decimal point) and converts it to its numerical value. Zero will be returned if the first non-space character is anything other than a digit or a decimal point, plus sign, or minus sign.

### 3.7.4 Output Functions

The output functions are those related to the printing of data by Basic.

pos(i%) The value returned will be the current column position of channel i%. (Channel numbers will be discussed in the second half of this manual when disk files are described. At this point, it suffices to note that channel 0 is the user's terminal.) Column numbering starts at zero, so if the value returned is zero, then the cursor is in the first position (column) of a line.

spc(i%) This function may only be used in a "print" statement. It causes "i%" spaces to be printed. Basic will respond with error number 74 if the value of i% is negative or greater than 255.

tab(i%) This function also must only be used in "print" statements. It moves the cursor to column i% by printing space characters until the specified column is reached. If the cursor is already past this position, then no action is taken. (Moving backwards is not allowed.) The argument must be a positive number less than 256. Column numbering starts at column 0.

### 3.7.5 User-defined Functions, the "def" statement

Basic provides the capability for the user to define functions which compute the result of a mathematical formula applied to an argument. This is most used in those circumstances where a formula must be evaluated at several different places in a program, each with a different argument. Once defined, these functions may be called in the same manner as the intrinsic functions, such as the "sin" function.

A function is defined by using the "def" statement. The format of the "def" statement is:

<line number>def fn<variable>(<dummy variable>)=<expression>

The function is defined (understood by the computer) as soon as the "def" statement has been executed. The same function may be redefined at any time in the program because only the most recent definition is used. The defined function may contain only one argument. The dummy variable used to represent this argument is local to the function definition and has no other meaning to the program.

A legal function name is formed by preceding any floating point variable name with the letters "fn". If, for example, we want to use the variables x, h1, and EE as function names, we would write them as fnx, fnh1, and fnEE respectively. A function is called just by the appearance of its name with an argument. When program execution detects a function, execution branches to the function definition where the defining expression is evaluated, using the supplied argument, and the value is assigned to the function name. Execution then continues in the line where the function was called with the function name having taken on the value of the function.

For example, if we wanted to define a function called "fntt" which would multiply a variable passed to it by 10, we could use the following definition:

```
180 def fntt(V)=10*V
```

Now suppose the definition has already been executed and the following two lines of code are executed:

```
300  X=3
310  Y=100+fntt(X)
```

After they have been executed the value of Y will be 130. In statement 300, "X" is assigned a value of three. The next statement first calls the function "fntt". It is passed the variable "X" which is multiplied by ten as instructed by the function definition. Control now returns to statement 310 where the function name (which now has a value of 30) is added to 100 and this sum of 130 is assigned to the variable Y. Restrictions on the use of the "def" statement are that the definition must consist of only one line, only one argument is permitted, the returned value must be floating point; integer and string functions are not allowed.

### 3.7.6 The "randomize" statement

The "randomize" statement is used to make the random number generator produce a different sequence of random numbers each time that the program is run. The format of the "randomize" statement is:

```
<line number> randomize
```

For example:

```
30 randomize
```

The operating system clock, as well as other time-related information, is combined to generate a seed for the random number generator. It is very unlikely that successive runs of a program containing a "randomize" statement will produce the same sequence of random numbers. The "randomize" statement should be executed prior to the first use of the "rnd" function.

## 3.8 Subroutines

When writing a program, it is sometimes desired to execute the same sequence of statements in different parts of the program. It would not be efficient to put a copy of those statements at each place where it is needed. Basic provides the capability for placing these statements at one location in the program, and calling on them from other parts of the program. A group of statements that is used in this way is called a "subroutine". There are two statements that are used with subroutines: the "gosub" statement, and the "return" statement.

### 3.8.1 Using a Subroutine - The "gosub" Statement

The "gosub" statement is used to transfer control to a subroutine (also known as "calling" the subroutine). The format of the "gosub" statement is:

<line number> gosub <line number>

For example:

```
5 gosub 250
```

This example will call the subroutine at line 250.

All subroutines should end with a "return" statement which will pass control back to the statement following the "gosub" which called the subroutine. The format of the "return" statement is:

<line number> return

For example:

```
34 return
```

"Return" instructs the computer to return to the calling routine from the subroutine that is now being executed. This is how subroutines should be exited. When control returns to the routine that called the subroutine, execution resumes at the first statement following the statement that caused the branch to the subroutine. The statement that branched to the subroutine will normally be a "gosub" or an "on gosub" statement. (The "on gosub" statement is described in the second half of this manual.) The "return" statement must be the last statement in lines containing more than one statement since statements following this one (on the same line) would never be reached and executed.

## 3.9 Termination Statements - "end" and "stop"

Termination statements are used to stop execution of a program after the desired computations are performed. The two termination statements described in this section are "end" and "stop". There is another termination statement, "exit", which is described in the second half of this manual.

The "end" statement causes the program to stop executing without printing any message. When the Basic detects an "end" statement, it stops executing the program, prints the "Ready" prompt, and waits for a command from the user. The format of the "end" statement is:

<line number> end

For example:

300 end

"End" can be placed anywhere in a program; it does not have to be the last statement in the program.

The "stop" statement is another way to stop the execution of a Basic program. The format of the "stop" statement is:

<line number> stop

For example:

50 stop

When a "stop" statement is executed, program execution is terminated and a message is printed, informing the user where the break in execution occurred. If statement 50 above were executed it would halt the program and print the message:

stop at line 50

The program can be restarted with a "cont" command and execution resumes following the "stop" statement. (The "cont" command is described in a subsequent chapter.) In short, the "stop" statement works just like the "end" statement except that "stop" causes a message to be printed. Neither the "end" statement nor the "stop" statement should be followed by another statement on the same line.

# Chapter 4.

# Advanced Commands

This section contains descriptions of those commands that are of use to the more experienced programmer. This section also includes a discussion of "immediate mode" statements.

## 4.1 Advanced Commands

compile The "compile" command is used to save a program on disk in a special format. In this format, all keywords are replaced by single character values and all remarks and other non-essential text are removed. The resultant saved program will in most cases be smaller than the same program saved with the "save" command, and will load into memory faster. A compiled program can not be loaded, listed, or edited. It can only be executed by using the "run" command with a file name specified. Since a "compiled" program cannot be edited, you should not delete or othewise destroy any copy of the program that you may have created with the "save" command, in case it becomes necessary to make changes to the program. The "compile" command is intended to be used to save programs that are ready to be put into production use.

The "compile" command requires a single argument which is the name of the file in which to save the program. This file name should be enclosed in quotation marks (either single or double quotation marks may be used), and must include such UniFLEX path information as may be required to put the file in the desired directory. For example:

```
compile "ledger"
compile "usr/source/basic/program"
```

If a file with the same name already exists, the following prompt is issued:

```
Delete existing file?
```

Answering this prompt with a "y" followed by a carriage return will cause the existing file to be deleted, and a new file, containing the saved program, to be created. Answering this prompt with an "n", a "control-d", or a "control-c" will cause the existing file to not be deleted, and the "compile"

command to be aborted.

cont

The "cont" command is used to restart a program after it has been stopped by either a "stop" statement or a "control-c". The "stop" statement may have occurred anywhere in the program but the "control-c" would have had to have been typed while the program was waiting for input at an "input" statement or "inch$" function. The command can not restart your program if you got an error during program execution or if you type in more program lines after you have stopped. If the program was stopped by a "stop" statement, then "cont" will cause the program to continue at the next line following the "stop" statement. If the program has been stopped in an "input" statement or "inch$" function by a "control-c", then the "cont" statement will cause execution to resume at that line. Note that the entire line containing the "input" and "inch$" will be re-executed, including any preceding statements if there are multiple statements on that line.

renumber

The "renumber" command is used to resequence a Basic program in memory. The general form of this command is:

renumber [, <first number> [,<increment>]]

Thus, this command may be called in one of three ways, as demonstrated in the following three examples:

```
renumber
renumber,100
renumber,100,20
```

In the first example, the "renumber" command was invoked without arguments. In this case, the line numbers will be resequenced with the first line given the number 10 and incrementing by 10 for subsequent lines. In the second example, the first line will be given the number 100, and subsequent line numbers incremented by 10. The third example specifies a starting number of 100 and an increment of 20.

The largest line number may not exceed 32767. This is checked by the "renumber" command, and a message issued if the last line number would be larger than this value. In this case, no renumbering takes place.

The "renumber" command also checks for line numbers which refer to lines not in the program. This is considered an error in the program. If one is detected, a message is issued indicating the line number which was not found and the number of the line containing the reference. In this case, the resequencing of the program continues.

The "renumber" command cannot detect when a line number is used as a constant; for example when comparing it to the "erl" variable. These must be changed manually by the programmer.

scale

The "scale" command specifies to Basic the number of digits to the right of the decimal place that are to be preserved. A scale factor of zero turns off the scaling feature. The maximum scale factor is six (6). The scale factor is specified as an argument to the command. For example:

```
scale 2
scale 5
scale 0
```

If "scale" is typed without an argument, the current value of the scale factor is printed.

The "scale" command must be used before the program is loaded or typed into Basic. It cannot be used if a program already is in memory. If it is necessary to change the scale factor with a program in memory, it must first be saved with the "save" command, then "new" must be typed to delete the program. The scale factor may now be set and the program reloaded from the saved copy.

With the scale factor set to non-zero, Basic scales all floating point values by 10**scale factor and rounds to a whole number. In effect all floating point numbers become integers; the fractional parts disappear. The not-so-obvious advantage is after many floating point operations, round-off error does not become significant because internally all numbers are integers. A short example will help clear this up.

```
10 for i%=1 TO 10000
20   f = 0.01 + f
30 next i%
40 print 100.0 - f
99 end

run

 1.77813319624e-12

Ready
```

Note that because of round-off error, the expected result of zero is not achieved.

This round-off error can be eliminated through the use of the

"scale" command. If we "save" the program, set the scale factor to at least 2 (if we set it to 1 the constant 0.01 will be scaled to 0), load the program back in and run it, we will get the following result:

```
run

 0

Ready
```

In the above case, the scale factor was set to 2, therefore, the constant "0.01" was converted internally to

$$0.01 * 10^2 = 1.0$$

Repeatedly adding 1.0 does not generate any round off error since 1.0 can be exactly represented in binary floating point notation. Thus, the round-off error normally present has been eliminated.

The scale factor is automatically saved with the program if the "compile" command is used. Thus it becomes effective whenever the "compiled" program is run. It is not possible to change the scale factor on a "compiled" program. It would be necessary to set the new scale factor, load in the source for the program, and issue the "compile" command again to save the program with the new scale factor.

troff — "troff" turns the trace feature off. The trace feature is described in the description of the "tron" command.

tron — The "tron" command enables the program trace feature. When enabled, this feature causes the line number of each line to be printed whenever that line is executed. This provides information on the program flow that may be used for debugging purposes. The trace feature is disabled by the "troff" and "new" commands.

+ — The "+" command tells Basic to send the rest of the command line to the UniFLEX operating system to be executed as a separate task. For example:

```
+list usr/source/program
```

This command would send the command "list usr/source/program" to UniFLEX for execution. Control will not return to Basic until the task has completed.

## 4.2 Immediate Mode Statements

Immediate mode statements are similar to those Basic statements discussed earlier, with the exception that they are entered without a line number in response to the "Ready" prompt. Under these conditions, Basic will execute the statement immediately. For example, if

```
print "Hello"
```

were typed in response to the "Ready" prompt, Basic would immediately respond by printing "Hello". The most frequent use of immediate mode is to print the results of some calculation. For example:

```
print sqr(sin(14.*pi/180.))
```

This would print the square root of the sine of 14 degrees. Note that the constants 14 and 180 were specifically represented as floating point constants by including the decimal point. As described earlier in the section on integer arithmetic, Basic will perform integer arithmetic if both of the operands of an operation are integers.

If more than one statement is required to compute the calculation, it is recommended that they be placed on one line separated by colons or backslashes. It is not guaranteed that Basic will remember the results from one immediate mode statement to another.

# Chapter 5.

# Advanced Statement Features

This chapter discusses some of the more advanced statements available in Basic. Most of these are extensions to those statements discussed in the first half of this manual. Advanced features of input and output statements are covered in a separate chapter, as are system interface features.

## 5.1 Conditional Execution - The "if-then-else" Statement.

An expanded form of the "if-then" statement is provided in Basic. This is the "if-then-else" statement. The general form of this statement is:

```
<line number> if <expr> then <line number> else <line number>
                            <statement>       <statement>
```

For example:

```
20 if h/(y+5)<8 then goto 50 else print "Out of range"
```

This is identical to an "if-then" statement except that when an "if-then" statement evaluates an expression to be false it does not execute the "then" part of the statement and execution passes to the next sequentially numbered statement. In an "if-then-else" statement, if the expression has been evaluated to be false then whatever follows the "else" is executed. To simplify the explanation slightly, the basic form for the "if-then-else" is:

```
<line number> if <expr> then <statement 1> else <statement 2>
```

Considering this form, we will assume the expression has been evaluated. If it was true, then statement 1 will be executed; and if it wasn't, then statement 2 will be executed. Looking at the example, if the expression is true then the statement "goto 50" will be executed. If the expression is false, then the message "Out of range" will be printed. Just as with the other conditional statements, "if-then-else" statements may be nested.

### 5.1.1 The "ambiguous else" Problem

Since there are two forms of conditional, the "if-then" and the "if-then-else", a problem of interpretation can arise. Let us consider the following example:

```
if a<5 then if a<2 then 500 else 600
```

It appears that this statement may be interpreted in two ways. The first interpretation assumes that the part

```
if a<2 then 500 else 600
```

is a complete statement that is subordinate to the first "then". In this case, if "a" is greater or equal to 5 then the rest of the statement is ignored and execution continues with the next sequentially numbered statement. The other interpretation is that

```
if a<2 then 500
```

is under the control of the first "then" and the "else" belongs to the "if a<5". This is called the "ambiguous else" problem. Basic resolves this problem by always assuming that an "else" is associated with the nearest previous "if-then" construction (the first case above).

## 5.2 Computed Jump - The "on-goto" Statement.

Basic provides a mechanism whereby a program can jump to different line numbers based on the value of an expression. This construction is often called a "computed jump" or a "computed goto". In Basic, this form of jump is called an "on-goto" statement. The general form of this statement is:

```
<line number> on <expression> goto <list of line numbers>
```

For example:

```
200 on i%+1 goto 500,600,700
```

The expression is evaluated and the integer portion of the result (any fractional portion will be discarded) determines the line number at which Basic will start executing. The "list of line numbers" has positions corresponding to 1,2,3,4,... So if the expression is evaluated to have a value of 1, then this statement will cause a jump to the first line number in the list. A value of two for the expression will cause a jump to the the second line number in the list and so on for as many numbers as there are listed. In the example above, if the expression evaluates to 1, Basic will go to line 500. If the expression evaluates to 2, control will go to line 600, and so forth.

If the expression evaluates to a number which is either less than one or greater than the number of line numbers listed, an error message will result. No statements should follow the "on-goto" in lines containing multiple statements.

## 5.3 Computed Subroutine Call - The "on-gosub" Statement.

Basic provides a mechanism whereby a program can call different subroutines based on the value of an expression. This construction is called a "computed subroutine call". In Basic, this form of subroutine call is termed an "on-gosub" statement. The general form of this statement is:

<line number> on <expression> gosub <list of line numbers>

For example:

```
20 on i%+1 gosub 30,40,50,60
```

The expression is evaluated and the integer portion of the result (any fractional portion will be discarded) determines the line number of the subroutine that is to be called. The "list of line numbers" has positions corresponding to 1,2,3,4,... So if the expression is evaluated to have a value of 1, then this statement will cause a subroutine call to the first line number in the list. A value of two for the expression will cause a subroutine call to the the second line number in the list and so on for as many numbers as you have listed. If the expression evaluates to a number which is either less than one or greater than the number of line numbers listed, an error message will result. The "on-gosub" statement should be the last statement in a line containing multiple statements.

## 5.4 Control of Precision - The "digits" Statement.

The "digits" statement specifies to Basic how many digits should be printed by a "print" statement. It also determines when two values are considered equal by the "approximately equal" relational operator. The general form of the "digits" command is:

<line number> digits <total> [,<fractional>]

Some examples:

```
10 digits 12      (set total digits to 12)
11 digits 12,3    (set total digits to 12,
                   of which 3 are fractional digits)
```

The maximum number of digits that may be specified is 17; the minimum is 1. The default value is 13 digits.

### 5.4.1 The Effect of "digits" on printing.

When printing, the "digits" statement determines the maximum number of digits that will be printed, and the number of digits that will appear to the right of the decimal point. In some cases, using the maximum number of digits (17) will result in an incorrect value being printed since this is slightly beyond the limit of precision. The second argument specifies how many digits to the right of the decimal point are to be printed. This number must be less than or equal to the total number of digits or an error will be generated. If the second argument is left off then Basic will print all fractional digits up to the total specified by the first argument. For example:

```
digits 4,3
print pi
 3.142
```

If a number is to be printed that is greater than the total number of digits specified, then the number will be printed in scientific notation. For example:

```
digits 1
print 10
 1e+01
```

### 5.4.2 Effect on the "approximately equal" relation.

The "approximately equal" relational operator uses the "total" value of the "digits" command to determine when its operands are to be considered equal. Of course, if the two operands are exactly equal, they are assumed to be "approximately equal". The algorithm used considers two cases: 1) one of the operands is zero, and 2) neither operand is zero. It should be noted that the larger possible values for the "total digits" (15, 16, 17) should not be used since round-off error will quickly render meaningless any comparisons at this level of significance.

### 5.4.3 Case 1: One operand is zero

In the case where one of the operands of the "approximately equal" relation is zero, the second must be within 10**(-"digits") of zero to be considered approximately equal to zero. For example, if the "total digits" value is 10, then a number that is between -10**-10 and +10**-10 is considered approximately equal to zero.

In the interests of speed, Basic uses techniques in computing the "approximately equal" relation that may result in an error in the neighborhood of one-half of an order of magnitude. Since the "approximately equal" relational operator is used for approximations, this error is not considered significant.

### 5.4.4 Case 2: Neither operand is zero

If neither operand to an "approximately equal" relation is zero, then the operands are considered equal if their difference is "digits" orders of magnitude smaller than the larger of the two operands. As in the previous case, Basic uses techniques which may result in an error of one-half of an order of magnitude.

## 5.5 Exchanging Variables - The "swap" Statement.

The "swap" statement exchanges the values of two variables. The general format of the "swap" statement is:

<line number> swap <variable 1>, <variable 2>

For example:

```
10 swap a,b
20 swap A%(i%), A%(i%+1)
```

The variables must be of the same type and may not be elements of a virtual array. The primary purpose of the "swap" statement is to decrease the time necessary to perform a sort. In typical applications, using the "swap" statement can result in a savings of 20-30% in sort time. It is especially advantageous when sorting strings since only internal pointers are exchanged, not the strings themselves.

## 5.6 Error Trapping in a Basic Program

During the running of a program, Basic may detect an error in the program or the data. It may detect a mathematical error such as an attempt to divide by zero or an attempt to take the logarithm of a negative number. Basic also checks for data consistency when requesting input; for example, typing a string in response to a request for a number will be seen as an error. There are also many circumstances which may cause an error when a program is manipulating data files; for example, writing to a file that was opened only for reading.

Normally, when Basic detects an error, it prints the error number and the line number of the statement that caused the error, then execution of the program is stopped and the "Ready" prompt is displayed. For example:

```
Error #30 at line 520
```

An annotated summary of the error numbers is in an appendix at the end of this manual.

In some cases, however, the user may wish program execution to continue when an error is detected. Basic provides a mechanism for trapping errors detected while a program is running. There are statements which allow a program to execute an error-handling routine whenever an error is detected, and to return control back to the main program if such is desired. Read-only variables are available that indicate which error was detected, and the line in which the error occurred, so that the error handling routine can determine what action to take in response to the error.

### 5.6.1 Defining an Error-Handling Routine

An error handling routine can be any set of Basic statements. The user tells Basic the line number to which to go if an error is detected by using the "on error goto" statement. If the error handling routine wants to return control back to the main program, the "resume" statement is used.

The general form of the "on error goto" statement is:

```
<line number> on error goto [<line number>]
```

For example:

```
on error goto 1000
```

If the line number is omitted or is zero, the error trapping feature of Basic is disabled and any error will then cause Basic to print the error

number and return to the "Ready" prompt. If an "on error goto" statement refers to a line that does not exist, an error 60 (Line cannot be found) will be generated when Basic tries to find the error handling routine in response to some other error.

The general form of the "resume" statement is:

<line number> resume [<line number>]

For example:

resume 100

The resume statement is used to pass control back to the main program after an error routine has been executed. If a line number is specified, control will pass to that line. If no line number is specified, the line that caused the error will be re-executed. If there are multiple statements on a line, all of them will be re-executed. If an error handling routine is to return control to the main program, a "resume" statement must be used. If Basic detects another error while executing the error handling routine (that is, before a "resume" statement is executed), execution will stop and the new error number and failing line number will be printed, followed by the "Ready" prompt.

### 5.6.2 The "err" and "erl" System Variables

When any error occurs, the two variables named "err" and "erl" are set to values that describe the error. The "err" variable will contain the error number and "erl" will contain the line number of the line which was executing at the time the error happened. These variables may not be set by the user but may be read at anytime. As an example:

130 if err=4 then print "No such file"

This line will cause Basic to print "No such file" if the last error was an error number 4. The variable "erl" is used in a similar manner.

IMPORTANT NOTE: If the "erl" variable is used in an "if" statement or an expression, the line numbers against which "erl" is being compared will not be changed by the "renumber" command. The line numbers in such statements must be changed manually after the new line numbers have been determined.

### 5.6.3 Error Handling Examples

Following are examples which demonstrate typical applications of the "on error goto" and associated statements. The first example deals with "data type mismatch" errors (error number 30).

```
  10 on error goto 1000
  20 input "Please type three numbers",A,B,C
  30 print "The sum is";A+B+C
  40 goto 20
1000 if err<>30 then on error goto 0
1010 print "Please type numbers only!"
1020 resume
```

Line 10 tells Basic where the error routine is located (line 1000). When the "input" statement is executed on line 20, it expects only numeric data to be typed. If the user enters a letter instead of a number, Basic would normally stop and print Error #30 at line 20. Since the "on error goto" statement has been executed, Basic instead would transfer control to line 1000 if an error occured. Line 1000 checks to see if the error is number 30. If "err" is not 30, the "on error goto 0" is executed which disables the error handling routine and causes Basic to print the error number on the terminal and stop. If it is error 30, the message on line 1010 is printed. Line 1020 will cause the program to re-execute the line which caused the error (line 20), and the input prompt will be reissued.

The next example demonstrates how the "on error goto" mechanism is used to detect the "control-c" interrupt and the "control-d" keyboard end of file signal. When a "control-c" is typed, Basic will generate an error number 34. This value may be tested by the error handling routine. A "control-d" typed in response to an "input" statement or "inch$(0)" function call will generate an error number 8. The following program is a simple calculator that adds the numbers that are typed by the user and prints the sum when a "control-d" is typed. The program is terminated by typing "control-c".

```
  10 on error goto 1000
  20 t=0
  30 input "Number",n
  40 t=n+t
  50 goto 30
1000 if err=34 then end
1010 if err<>8 then resume 30
1020 print
1030 print t
1040 resume 20
```

Line 10 tells Basic that the error handling routine is at line 1000. Line 20 initializes the sum and lines 30 through 50 request numbers and add them. In the error handling routine, line 1000 checks the error number to see if a "control-c" was typed. If so, the program exits.

Line 1010 checks for a "control-d" having been typed. If not (the error number is not 8), the program resumes execution by requesting another number. Lines 1020 through 1040 are executed if the error number was 8, meaning that a "control-d" was typed, and they print the sum and restart execution at line 20.

When writing programs that have an error handling routine, it is wise to check the error number for the "control-c" interrupt value. In the above example, if no check were made for error 34, then there would be no way to stop the program because the error handling routine would always go to either line 20 or line 30. In this case, the only way to stop the program would be to use the UniFLEX "quit" signal, a "control-\". However, this would also cause Basic to terminate and the program would be lost. So it is always a good programming practice to check for error 34 in the error handling routine and exit if it is detected.

Perhaps the most frequent use of the "on error goto" mechanism is in connection with the reading and writing of data files. This is discussed further in later chapters.

# Chapter 6.

# Advanced I/O Features

## 6.1 Literal Input - The "input line" Statement

The "input line" statement is used to read an entire line into a single string variable. All characters on the line, including spaces, punctuation marks, and most control characters, are stored in the string variable. The only control characters that cannot be read are the carriage return and those that have special meaning to UniFLEX, such as control-c, control-d, control-\, the backspace character, the line delete character, and the escape character. The general form of the "input line" statement is:

<line number> input line <string variable name>

For example:

```
10 input line a$
20 input line b1$(5)
```

Note that no prompt may be printed as is the case with the "input" statement. A separate "print" statement must be used to print the prompt, if one is desired. Also, only one variable name may be listed.

## 6.2 Timed Input - The "wait" Statement.

The "wait" statement is used to set a time limit on all requests for input from the user's terminal. The general form of the "wait" statement is:

<line number> wait <time limit>

The time limit is specified in seconds and may range from zero (which turns off the time limit) through 32767. Whenever a request for input from the user's terminal is encountered, Basic will wait no longer than the time limit for the input request to be satisfied. An input request is considered satisfied when the carriage return is typed. (In the case of the "inch$(0)" function, typing any character will satisfy the input request.) If the input request is not satisfied before the time limit expires, an error number 35 is generated.

## 6.3 Formatted Output - The "print using" Statement.

The "print using" statement gives the user more control over the format of data being printed. The general form of the "print using" statement is:

```
<line number> print using <string>, <print list>
```

Several examples:

```
10 print using '####.##', 1234.56
20 print using '###.### is the square root of ###',sqr(x),x
30 print using '\  \', 'String field'
40 print using a$, i, j, k%, b$
```

The "print using" statement is a highly flexible form of the print statement which gives the user a great deal of control over the format of the output. The <string> is called the "format string" and consists of special characters that define the format and size of the print fields. These special characters are described below. All other characters are also allowed in the format string. These are called "literal characters" and have no special meaning. They are printed as soon as they are seen during the processing of the format string. One restriction on the content of the format string is that it may not end with a literal character.

The <print list> is the same as for a normal "print" statement where expressions are separated by either commas (,) or semicolons (;). "Print using" normally ignores the meaning of these separators unless it has reached the end of the format string. Then and only then do the separators become meaningful. If the end of the format string is reached before all of the data items are printed, the format string is re-used from the beginning to print the remaining data items.

### 6.3.1 Exclamation Mark (!)

The exclamation mark is used to denote a single character string field. For example:

```
print using '! ! !', '01', 'AB', '()'
```

The result of this statement is:

```
0 A (
```

Note that only the first character of each string was printed. Also note that the spaces between the exclamation marks in the format string were printed.

### 6.3.2 Backslash (\)

A pair of backslashes (\) are used to denote a string field of 2 or more characters. The size of the field is determined by the total number of characters between the backslashes PLUS the two backslashes. Any characters may be between the backslashes as they are ignored. It is recommended as a good programming practice to include a count or a string of numbers inside the slashes for programming documentation. For example:

```
print using '\23456\', 'This is a test'
```

The result of this statement is:

```
This is
```

Notice that only seven characters were printed. This is because the total number of characters in the string field format is seven: the two backslashes plus five characters between them.

If the string being printed is shorter than the format, trailing spaces are added. For example:

```
print using "\23456\","ab"
```

will print: ab followed by 5 spaces.

### 6.3.3 Number Sign (#)

The number sign (#) is used to denote a numeric field. For example:

```
print using '####.##', 124.555
```

The result of this statement is:

```
124.56
```

Note that the format string specified the location of the decimal point as well as the number of digits to print on either side of it. If the number to be printed will not fit in the field defined, a percent sign will precede the number and it will be printed as though no "print using" statement were being used. For example:

```
print using '#.#', 10.3
```

results in % 10.3 being printed.

If the fractional digits of the number does not fit into the field defined, the number will be rounded and then printed. If the resulting rounded number is too large to fit in the field, a percent sign will be printed before the number. For example, the statement

```
print using '#.#      #.#', 1.99, 9.99
```

will print

```
2.0      %10.0
```

Note that both values are rounded, but the second became too large for the specified format.

If the number being printed is negative, a leading minus sign is printed. This minus sign takes up one of the print positions. For example:

```
print using "####", -235
```

will print the value

```
-235
```

### 6.3.4 Dollar Sign ($)

The dollar sign ($) is used when printing amounts of money. The field is defined the same as with the pound sign (#) except that the first two characters of the field must be the dollar sign. The dollar sign that is printed "floats"; that is, it appears immediately to the left of the number. For example, the statement

```
print using '$$###.##', 123.92
```

will print the value: $123.92

The two leading dollar signs not only indicate that a dollar sign is to be printed, they also reserve an extra place for the number being printed. In the example above, there are actually 4 places reserved to the left of the decimal point that may be used by digits. We could use the same format to print a value that had four digits to the left of the decimal point; for example:

```
print using '$$###.##', 7123.92
```

will print the value: $7123.92.

If the value being printed is too large for the numeric field specified, or if the value is negative, a percent sign will be printed followed by the number without a leading dollar sign. For example, the statement

```
print using '$$##.##  $$##.##', 1234, -1.23
```

will result in the following being printed:

```
% 1234.00  % -1.23
```

In the first case, the value was too large for the field as defined by the format. In the second case, the number was negative. If it is desired to print a negative dollar amount, a trailing minus sign must be used. This is discussed later on.

### 6.3.5 Asterisk (*)

The asterisk (*) is used to fill the leading blanks of any numeric field with asterisks. This is especially useful when printing a numeric field that should not be easily altered (for example, when printing checks). This format is specified in the same manner as the floating dollar sign format described above. That is, two asterisks are specified in front of the numeric field. For example, the statement

```
print using '**##.##', 10.2
```

will print **10.20. As is the case with the floating dollar sign described above, the two asterisks also reserve an extra place for the digits. If there is not enough room to print at least one asterisk, a percent sign will be printed followed by the number.

The leading dollar and the asterisk field cannot both be specified for the same field. Instead a literal dollar sign can be used preceding the asterisk field. For example, the statement

```
print using '$**##.##', 1.15
```

will result in $***1.15 being printed.

### 6.3.6 Comma (,)

The comma (,) is used to insert commas in a numeric field every three places to the left of the decimal point. If at least one comma is embedded in a numeric field and before the decimal point then commas will be inserted in the appropriate places. It is not necessary to place commas in the format at the exact locations required. A comma

before the numeric field or after the decimal point is considered to be a literal character and will be printed as soon as it is seen. If, while filling the numeric field with commas, Basic runs out of field room, a percent sign will be printed followed by the number just as far as Basic was able to fill it. The following are some examples of the use of commas:

```
print using '#,,,#.##', 1234.56 results in 1,234.56
print using '###,###,###', 1E6 results in 1,000,000
print using '####,###', 1E6 results in %1000,000
```

Note in the last example that the resulting number did not fit in the specified field width of eight characters (the comma counts as a character).

### 6.3.7 Trailing Minus (-)

The trailing minus is used when a floating dollar sign or asterisk fill field is used with negative numbers. Under these conditions, the minus sign is printed after the number. For example:

```
print using '$$##.##-', -10.23 results in  $10.23-
```

### 6.3.8 Circumflex or Caret (^)

The circumflex or caret (^) is used to denote scientific notation format for numeric data. Four and only four circumflexes are allowed and must trail the numeric field. The four circumflexes (^^^^) are used to reserve space for the "e+xx" notation used in the scientific notation format. None of the other formats may be used with the scientific notation format since it uses print positions both before and after the number. This format is particularly useful when printing tables of numbers that are quite large and have many decimal places. For example, the statement:

```
print using '#.##########^^^^', sin(x)
```

will print 2.55063602616E-01.

# Chapter 7.

# System Interface Features

This chapter discusses the statements, functions, and variables that are available to the Basic programmer for use in interfacing with the UniFLEX Operating System, or for determining some characteristics of Basic itself.

## 7.1 Invoking A Task - The "exec" Statement and "tstat%" Variable.

The "exec" statement allows the Basic programmer to call upon another UniFLEX program from within an executing Basic program. The program called may be any UniFLEX program that can be called from the "shell" program. The general form of the "exec" statement is:

```
<line number> exec,<UniFLEX command string>
```

Some examples:

```
10 exec,"ttyset,+raw"
50 exec,"list test"
100 exec,"kill testdata*;rename newdata testdata1"
```

In the third example, note that it is possible to specify more than one command by using the UniFLEX command separator character (;). The <UniFLEX command string> is executed as a child task to Basic. Basic will wait until the child task is finished before continuing with the rest of the program. During the time that it is waiting, any "control-c" interrupt will be pended, and processed when the task has completed.

When the task has completed, the task termination status is stored in the integer variable "tstat%". This may be tested by the program to determine if the task terminated normally. See the UniFLEX Operating System manual for a description of the task termination codes.

The only mechanisms available to Basic for passing information to a generated task are through parameters to the task and through disk files.

NOTE: It is possible to generate a task that runs in the background by using the "shell" command terminator "&". If this is done, the variable "tstat%" will not reflect the status of the background task. It is not possible for a program to determine the termination status of such a

background task.

## 7.2 Terminating Basic from a Program - The "exit" Statement

The "exit" statement is analogous to the "exit" command described earlier. This statement causes Basic to terminate and return control to the program that called it, usually the "shell" program. The general form of the "exit" statement is:

<line number> exit [<termination code>]

For example:

```
10 exit
50 exit 10
```

The termination code is an integer between 0 and 255 inclusive. If no termination code is specified, 0 is assumed. See the UniFLEX Operating System manual for additional information on termination codes.

The primary use for the "exit" statement is in writing utility programs in Basic. By using the "exit" statement, the utility program will terminate the task rather than returning to the "Ready" prompt, which would require the user to type "exit". Additional information on writing utility programs in Basic is described in an appendix to this manual.

## 7.3 The "fre(0)" Function

The "fre(0)" function returns as its value the number of bytes available for use by Basic when running a program. This "free space" is used by Basic to store strings, arrays, file buffers, and miscellaneous information. If Basic runs out of free space, it will request additional memory from the operating system. Thus, the value returned by the "fre(0)" function may understate the amount of memory actually available to Basic. If Basic has been allocated its maximum amount of memory permitted to it by the system, and it runs out of free space, either error 80 or error 82 will be generated.

## 7.4 The "sleep" Statement.

The "sleep" statement is used make a program pause for a specified period of time. The general format of the "sleep" statement is:

<line number> sleep <time in seconds>

For example:

```
10 sleep 30
```

The sleep time specified may not be longer than 32767 seconds. If a zero or negative sleep time is specified, no pause is performed. A "control-c" interrupt will interrupt a sleeping program.

## 7.5 Determining the Task Number - The "task$" Constant.

The constant "task$" has, as its value, the task number of Basic. There are no leading or trailing spaces in the string. The primary purpose of this constant is to aid in creating unique file names, since, at any one time, no two tasks may have the same number. The following statement generates such a file name:

```
10 f$="file"+task$
```

## 7.6 Processing Arguments from Basic

When Basic is called, additional arguments may be specified in addition to the name of a file containing a program. These additional arguments are accessible from a Basic as an array of strings, "arg$". The first element of this array, "arg$(0)" is the first argument passed to Basic, which is the name of the file containing the program. A count of the number of arguments is available in the integer constant "argc%". The major application of this argument processing capability is in the writing of utility programs in Basic. This is described in an appendix to this manual.

As an example, let us assume that the following program is stored in the file named "print-hex".

```
10 if argc%<2 then exit
20 for i%=1 to argc%
30 print hex(arg$(i%))
40 next i%
50 exit
```

This program assumes that its arguments are hexadecimal values. The decimal equivalent of each of these values is printed. Line 10 checks for any arguments having been supplied. If there are none, the program exits. Lines 20 through 40 prints the decimal equivalent of each argument. Assume that the program was called as follows:

```
++basic print-hex a,60,3f
```

The output would be:

```
10
96
63
```

A reference to an argument beyond the number actually passed returns the null string.

# Chapter 8.

# Disk Files

Basic provides the capability for a program to read and write disk files. There are two types of disk files, sequential files, and record I/O files. Record I/O files are also called "random" files. This chapter is concerned with sequential files. Record I/O files are discussed in a later chapter.

## 8.1 Introduction to Sequential Files

A sequential file is one in which the data must be read in the order that it appears in the file. For example, if it is desired to access the fifteenth data item, the fourteen items preceeding it must be read in order to reach the fifteenth item. Data is written to sequential files by "printing" lines of information, in the same way as one would "print" to the terminal. Similarly, data is read from a sequential file by using a form of the "input" statement, just as one would read from a terminal. Data is stored in a Basic sequential file in the form of "lines", terminated by a carriage return. A Basic sequential file is an ASCII file; all information is stored as ASCII characters. The "lines" in a sequential file need not all be the same length. It is convenient to think of a sequential file as one would think of the terminal.

## 8.2 The "open" Statement

Before a file may be used by Basic, it must first be opened by using the "open" statement. The "open" statement associates the name of the file that is going to by read or written with a "channel number". All references to the file subsequent to the "open" statement are made by specifying the channel number rather than the file name. One of the three following general forms of the "open" statement is used to open the file:

```
<line number> open old <string expression> as <file expression>
<line number> open new <string expression> as <file expression>
<line number> open <string expression> as <file expression>
```

Some examples:

```
open old "test" as 1
open new A$ as F%
open "/usr/marie/data-file" as 6
```

The "string expression" mentioned above specifies the name of the file. UniFLEX path information may be specified if necessary to accurately identify the file. The "file expression" specifies the channel number that is to be associated with the file. The channel number is an integer between 1 and 12 inclusive. (Channel 0 is a special case, described later.) If the "file expression" results in a floating point value, it is truncated before being used.

The first form of the "open" statement, the "open old" statement, is used for opening a file that is only going to be read (a "read-only" file). The file specified must already exist; if it does not, an error will be generated. An error will also occur if the program tries to write data into a file that has been opened "old". When a file is opened for reading, it is automatically positioned at the beginning of the data in the file. Thus, the first item of information read will be the first item of information in the file. The first example above opens the file "test" on channel 1 as a read-only file.

The "open new" form of the "open" statement is used for creating a new file and preparing it for writing. The file specified in the string expression will be created if it does not already exist. If it does exist, the original file will be deleted, without any message to that effect, and a new file with the same name will be created. A file that is opened as a "new" file may be both written and read (after having been "rewound", see the "position" statement). The second example above will open the file whose name is contained in the string variable A$ for writing. The I/O channel used will be the value of the variable F%.

In the third form of the "open" statement, the one that specifies neither "old" nor "new", the file specified will be opened for both reading and writing if it already exists. If the file specified does not already exist, a new file will be created, just as if it were opened as a "new" file. The third example above opens the file "test-data", in the directory "/usr/marie", for both reading and writing. When a file is opened using this form of the "open" statement, it is positioned at the beginning of the file. Thus, if the file is first read, the first line in the file will be read. If the file is being written, the new information will overwrite some of the old information on the file. This is discussed further when "writing to a sequential file" is discussed.

Since UniFLEX is a multi-user and multi-tasking operating system, it is possible for more than one task to want to access the same data file. If the file is only being read, there is no problem. If one or more of the tasks wishes to write into the file, there may be a conflict with other users of the file. Basic does <u>not</u> have a mechanism to prevent

this conflict for sequential files.

## 8.3 The "close" Statement

The "close" statement is used to inform Basic that the program is finished using a file. Closing a file frees up the channel it was using thus allowing another file to be opened on that channel. The general form of the "close" statement is:

<line number> close <expression> [,<expression>...]

Each expression specifies the channel number of a file that is to be closed. As you can see, more than one file may be closed with a single "close" statement. Closing a file that has not been opened will generate an error. Some examples will demonstrate its use.

```
200 close 3
520 close 1,6
```

Line 200 will close the file on I/O channel 3, and line 520 will close the files open on channels 1 and 6. All files should be closed by a program, before it ends. If a program terminates abnormally (by generating an error), Basic will automatically close all files.

## 8.4 Reading and Writing Files

Sequential files are read and written by using modified forms of the "input" and "print" statements, respectively. These forms specify the channel number associated with the file being read or written. The file must have already been opened with an "open" statement before it may be used. If it has not been opened, an error is generated.

### 8.4.1 Writing Sequential files

Modified versions of the "print" and "print using" statements are used to write data to sequential disk files. These modified forms include the channel number associated with the file being written. In addition, a "print line" statement is available which facilitates the insertion of separators between data items when more than one data item is written on a line. (These separators are necessary if the data is to be read from

the file.) The general forms of these statements are:

```
<line number print #<expression>, <print list>
<line number> print #<expression>, using <string>, <print list>
<line number> print line #<expression>, <print list>
```

In these forms, the "expression" is the channel number specified in the "open" statement that was used to open the file. As error is generated if the file has not been opened by an "open" statement. The "print list" specifies the information to be written to the file and follows all the rules of the normal "print" and "print using" statements. An example will demonstrate its use.

```
10 open new "testfile" as 3
20 print #3, "This is a test file"
30 print #3, using "PI to 5 decimal places is #.#####",pi
40 close 3
```

The above program creates a new file called "testfile" using channel 3. The resultant file will have two lines. The first will be "This is a test file" and the second will be "PI to 5 decimal places is 3.14159". As mentioned earlier, the file created by using these modified forms of the "print" and "print using" statements are ASCII files that may be processed by any program that manipulates ASCII files; for example, the UniFLEX Text Editor can be used to change the data in a sequential file written by Basic.

Another example will demonstrate a method of creating a table of the integers 1 through 5 and their squares. The program might look as follows:

```
100 open new "squares" as 1
110 for i=1 to 5
120 print #1, i, i^2
130 next i
140 close 1
```

Listing the file "squares" as created above would display the following:

```
1           1
2           4
3           9
4           16
5           25
```

Notice that since commas were used in the "print" statements, the resultant lines have the numbers spaced accordingly. This demonstrates that "columns" exist in sequential files, just as they exist for a terminal. The "pos" function can be used to determine the current column of a file, just as it can be used to determine the current column of a terminal. The argument to the "pos" function should be the channel number associated with the file.

The "print line" statement functions like the "print" statement with the exception that the separators between the data items are written to the file instead of spacing over to the next field. These separators are necessary if the data that is written is to be read from the data file by an "input" statement. The following program is a modification of the example above that wrote the "squares" file. In this version, however, the "print line" statement is used.

```
100 open new "squares-x" as 1
110 for i=1 to 5
120 print line #1, i, i^2
130 next i
140 close 1
```

If the "squares-x" file generated by this program is listed, it would look like this:

```
1 , 1
2 , 4
3 , 9
4 , 16
5 , 25
```

Note that instead of the normal spacing between the data items on each line the separator between them has been printed. This allows the information to be read by an "input" statement, as described below.

### 8.4.2 Reading a Sequential File

Modified versions of the "input" and "input line" statements are used to read the information from a sequential file. The general forms of these statements are:

```
<line number> input #<expression>, <variable list>
<line number> input line #<expression>, <string variable>
```

In these forms, the "expression" is the channel number specified in the "open" statement that was used to open the file. The "variable list" and "string variable" specified above are the same as in the normal "input" and "input line" statements.

The "input" statement reads a line of data from the specified file. Individual data items on the line will be assigned to the variables specified in the variable list. If more than one data item appears on the line, they must be separated by commas or semicolons. When writing multiple data items per line that will be read from the file by an "input" statement, it is the responsibility of the program to insert these separators appropriately. The "print line" statement, discussed earlier, is a simple mechanism to use for inserting the necessary

separators. The following program will demonstrate the use of the "input" statement by reading the "squares-x" file that was generated by using the "print line" statement.

```
10 open old "squares-x" as 2
20 for i = 1 to 5
30 input #2, a,b
40 print a,b
50 next i
60 close 2
```

This will cause the values to be read from the disk file "squares-x" and printed at the terminal.

Since Basic sequential files are ASCII files, it is not necessary that Basic have written the file that is being read. Any program that writes ASCII files, with each line terminated by a carriage return, can create files that Basic can read. The only restriction that is imposed on files that Basic reads is that each line may not contain more than 255 characters, including the carriage return. If a line contains more than 255 characters, the excess are discarded.

The modified form of the "input line" statement will read an entire line from the data file and assign it to the specified string variable. The trailing carriage return is not stored with the data. As with the "input" statement, a line read from a file cannot be more than 255 characters long, including the carriage return.

The "inch$" function may also be used to read data from disk files. The argument to the "inch$" function is the channel number associated with the file that is being read. Each call to the "inch$" function will read the next character from the specified file.

If an "input" or "input line" statement, or "inch$" function is used to read from a file from which all data has been read, an "end of file" error (error number 8) will be generated. The "on error goto" mechanism may be used to detect this condition and handle it appropriately. This is discussed later in the chapter.

## 8.5 The Special Case of Channel 0

There is another I/O channel besides those numbered 1 through 12. This is channel 0. It may be used to read from or write to the terminal, or it may be used to write to a disk file, but it may never be used to read from a disk file.

### 8.5.1 Reading from Channel 0

Reading from channel 0 reads from the user's terminal. The difference between performing an "input" from channel 0 and an input directly from the terminal, is that the question mark prompt is not issued when reading from channel 0. For example:

```
10 input #0, B$
```

will request input from the terminal just as a normal "input" statement, but no question mark prompt is printed. In addition, only a carriage return is echoed to the terminal after the data is entered; no line feed is printed. This mechanism allows a limited amount of cursor control for those programs that perform cursor manipulation. A user-supplied prompt is allowed when reading from channel 0. This prompt follows the channel number as in the following example:

```
10 input #0,"Type something: ",a$
20 print
30 print a$
```

The extra "print" statement at line 20 is necessary because the input from channel 0 did not cause a line feed. This "print" statement generates the necessary line feed to prevent the "print" statement at line 30 from overwriting the prompt.

The "input line" statement may also be used on channel 0 to suppress the question mark prompt.

### 8.5.2 Writing to Channel 0

It is possible to write to channel 0 without having opened a file on it. When this is done, the output goes to the user's terminal as though no channel number had been specified. As an example:

```
200 print #0, "This is a test"
```

This statement will print "This is a test" on the terminal, just as though the "#0," were not present.

It is also possible to open a file on channel 0. The syntax for this is the same as for opening a file on one of the other 12 channels. When this is done, all writing to channel 0 goes to the file. It is the same as writing to a data file from one of the other channels. As soon as channel 0 is closed with a "close" statement, any writing to it reverts back to the user's terminal. Note that when a file is opened on channel 0, it essentially becomes a "write-only" file since it is not possible to read a file through channel 0. Once the file has been written, however, it is possible to open it through another channel and read its

contents. The primary use of channel 0 is for writing files that are to be spooled to a line printer.

As an example of writing to channel 0, the following program prints a table of square roots to either the user's terminal or to a data file. If the table goes to a data file, it is spooled to a line printer before the program terminates.

```
10 print "Type file name, if any:";
20 input line #0, r$
30 print
40 if len(r$)<>0 then open new r$ as 0
50 for i=1 to 10
60 print #0, i, sqr(i)
70 next i
80 if len(r$)=0 then end
90 close 0
100 exec, "spr "+r$
```

Line 10 asks the user to type the name of the file that is to receive the data. If the data is to be printed at the user's terminal, only a carriage return should be typed. The use of channel 0 in line 20 suppresses the question mark prompt. Line 40 checks for a file name having been typed. If only a carriage return were typed then length of the string read in line 20 would be zero. If a file name was typed, line 40 opens it for writing. Line 60 prints the data to either the user's terminal (if no file name as typed), or to the specified data file. Line 80 terminates the program if no file name was specified. Line 90 closes the data file, and line 100 invokes the spooler program "spr", passing to it the name of the data file.

## 8.6 The "position" Statement

The "position" statement is a direct interface to the UniFLEX "seek" system call (described in the UniFLEX Operating System manual). This statement may only be used on Basic sequential files. An analogous statement, "seek", is available for use with Basic record I/O files. The "position" statement allows a Basic program to position a sequential file to any character within the file. The most frequent uses of this statement are to 1) rewind a file (start from the beginning again), 2) find the end of the file so that additional data may be appended to the file, and 3) to remember the current position in a file and return to that position later on.

The general form of the "position" statement is:

```
<line number> position #<channel> ,<character number>
                [,mode <mode number>]   [,response <variable>]
```

The "channel" is the channel number associated with the file being positioned. The file must already have been opened with an "open" statement.

The "character number" refers to that character in front of which the file will be positioned. The first character in the file is character number 0.

The "mode number" is the same as the UniFLEX "seek mode" as described in the UniFLEX Operating System manual. This number may take on the values 0, 1, or 2. Mode 0 indicates that the positioning is to take place from the beginning of the file; mode 1, from the current position; and mode 2, from the end of the file. If no mode is specified, mode 0 (from the beginning of the file) is assumed.

The "variable" specified as an argument to the "response" portion of the "position" statement must be a floating point variable. If a "response" is requested, by specifying "response" and a variable name, the number of the character in front of which the file is positioned is stored in the specified variable. If no response is requested, none is returned.

The following examples illustrate the most common uses of the "position" statement as outlined above.

### 8.6.1 Rewinding a File

Rewinding a file consists of positioning it in front of all of the data that is contained in the file. The easiest way to do this is to use the "position" statement to perform a position to character 0 from the beginning of the file. For example:

```
10 position #1,0
```

Since mode number defaults to 0, this statement will rewind the file associated with channel 1. If a "response" were requested, it would always be zero since the position operation left the file positioned in front of the first character in the file.

### 8.6.2 Positioning to the End of a File

In some instances, it is desired to append information to the end of an existing sequential file. In order to do this, it is necessary to skip all of the data on the file before starting to write the new information. The "position" statement provides a quick method for

skipping all of the information in a file. The following example demonstrates how this is accomplished.

```
10 position #1, 0, mode 2
```

This statement causes a position to character number 0 after the end of the file (specified by mode 2). The file is now positioned after all of the data in the file and any writing will not destroy existing data.

By requesting a response it is possible to determine the number of characters in the file, as follows:

```
10 open old "file" as 1
20 position #1, 0, mode 2, response r
30 close 1
40 print "The number of characters is",r
```

### 8.6.3 Remembering a Position in a File

By using the "response" request feature of the "position" statement, it is possible to remember the current position of a file, and return to that position later on in the program. For example:

```
100 position #1, 0, mode 1, response r
...
500 position #1, r
```

Statement 100 causes a position to zero characters after the current position (which does nothing), and requests that the position be stored in the variable "r". Statement 500 causes the file to be positioned to the location that was remembered in variable "r".

## 8.7 Error Handling

When reading and writing disk files, a Basic program should be prepared to handle a variety of errors that may occur. The "on error goto" mechanism, described earlier, should be used for such error handling.

The most frequent error that will be encountered when reading sequential files is the "end of file" error. This error is generated when an attempt is made to read more data than is contained in the file. The following program is a modification of an earlier example, and illustrates the use of "on error goto" processing to handle the "end of file" condition.

```
10 on error goto 100
20 open old "squares-x" as 2
30 input #2, a,b
40 print a,b
50 goto 30
60 rem error handling routine follows
100 if err<>8 then on error goto 0
110 close 2
```

This is similar to the earlier program that read the "squares-x" file and printed its contents. The main difference between the two programs is that this new version does not use a "for-next" loop to determine when all of the data has been read. Line 10 declares that an error handling routine exists at line 100. The "for-next" loop has been replaced by an unconditional loop that always looks for another line of data. When all of the data in the file has been processed, the next attempt to execute the "input" statement at line 30 will cause an error 8 to occur. Since an "on error goto" statment was executed, Basic starts executing the error handling routine. Here, the error number is checked for the "end of file" error. If it is not the "end of file" error, the statement "on error goto 0" is executed which causes the error number to be printed and Basic to return to the "Ready" prompt. If the error is the "end of file" error, the statement at line 110 is executed which closes the file and the program terminates.

Another use for the "on error goto" processing is to prevent the accidental deletion of a file by opening it as a "new" file. The following program segment illustrates how this is done.

```
10 on error goto 1000
20 open old "file" as 1
30 close 1
40 input "Delete existing file",r$
50 if r$<>"yes" then end
60 open new "file" as 1
70 rem body of program follows
...
1000 if err=4 then resume 60
1010 on error goto 0
```

Line 10 declares the error handling routine to be at line 1000. Line 20 attempts to open the file for read-only. If the file already exists, no error is generated and execution continues with line 30. Here, the file is closed and the user is asked if the file should be deleted. If the user types "yes" in response to the prompt, the file is opened as a "new" file which deletes the old file. If the file did not already exist, an error 3 is generated. The error handling routine at line 1000 checks for this and if this error occurs, resumes execution at line 60, bypassing the prompt, and creating a new file.

There are other errors associated with disk files, such as read and write errors and permission violations. An annotated list of the possible errors that may be generated by Basic is provided in an appendix to this manual.

# Chapter 9.

# Additional File Handling Statements

There are some statements used in handling files that refer to the file by its name rather than a channel number. These statements do not require that the file be opened by an "open" statement. All may be used as statements in a program or typed in response to the "Ready" prompt for immediate execution.

## 9.1 The "kill" Statement

The "kill" statement is used to delete an existing disk file. The general form of the "kill" statement is:

<line number> kill <string expression>

The "string expression" is the name of the file to be deleted. For example:

```
100 kill "ledger"
200 kill "/usr/otto/data"
```

Line 100 will delete the file "ledger" from the current working directory. Line 200 deletes the file "data" from the directory "/usr/otto". No error is generated if the specified file does not exist.

## 9.2 The "rename" Statement

The "rename" statement is used to rename a disk file. The general form of the "rename" statement is:

<line number> rename <string expression>, <string expression>

The first string expression specifies the name of the file to be renamed and the second string expression is the new name it will have. For example:

```
225 rename "test", "old-test"
```

This statement will change the name of the file "test" to "old-test". An error is generated if the file does not exist or if there is already

a file with the new name in the directory.

## 9.3 The "chain" Statement

When a program is too large to be loaded into memory and run by Basic it must be divided into several smaller programs. The "chain" statement allows a program to load another program over itself and execute it. The individual programs may be either Basic source programs or "compiled" programs. "Compiled" programs load much faster than source programs. The general form of the "chain" statement is:

<line number> chain <string expression> [<expression>]

The "string expression" is the name of the file that contains the program to be executed. The second, optional, expression designates the line number at which the program should be started. Without the line number specification, the program will begin execution at the lowest numbered line in the new program. An example will demonstrate:

```
1300 chain "/test/program/balance-2" 250
```

This statement will cause the file named "balance-2" in the directory "/test/program" to be loaded and run starting at line 250.

Note that when a "chain" occurs, the calling program is overwritten in memory and must be reloaded if it is to be run again. Also, when a "chain" statement is executed, all open files are closed, and all variables are destroyed. The only way that the calling program can communicate with the called program is through data files.

# Chapter 10.

# Record I/O

The basic idea behind record I/O is that data is stored on the disk in fixed length records. The length of each record is specified by the user when the file is opened. A record may be as small as one byte or as large as 16,383 bytes. Any record in a file may be read or written upon request and the data in each record is easily defined to be ASCII characters or binary numeric data. Strings may be stored on disk as characters and numbers may be stored in binary form as 8 byte floating point or 2 byte binary integers, eliminating the need for I/O conversions.

## 10.1 Opening a Record I/O File

Before any record I/O file may be referenced, it first must be opened. There are three forms of the "open" statement for use with record I/O files. These are:

```
<line number> open old <string> as <expression> [,size <expression>]
<line number> open new <string> as <expression> [,size <expression>]
<line number> open <string> as <expression> [,size <expression>]
```

The "string" specifies the name of the file to be opened. Path information may be included as part of the file name if necessary to accurately locate the file. The first "expression" specifies the channel number to be associated with the file. Subsequent references to the file from the Basic program use the channel number, not the file name. The channel number must be an integer between 0 and 12 inclusive. The "size" modifier specifies the size of each record, in bytes. If omitted, the default record size is 252 bytes. The record size may range from 1 through 16383.

The "open old" form of the "open" statement tells Basic to search for a file which already exists (an "old" file). If the file is not found, an error number 4 will be issued. The "open old" statement opens the file as a "read-only" file. Such files may be read, but not written. For example:

```
10 open old "test" as 8 size 50
```

This will cause Basic to search the current working directory for the file named "test". If found, it will be opened for reading only. The size of each record in "test" is declared to be 50 bytes.

The "open new" form of the "open" statement tells Basic to create a "new" file. If the file specified already exists, it will automatically be deleted and a new file of the same name will be created. If the file does not exist, it will be created. A file opened with the "open new" statement is opened for both reading and writing. For example:

```
10 open new "inventory" as 4 size 500
```

This statement will cause a new file named "inventory" to be created in the current working directory. Any existing copy of the file will automatically by deleted. The size of each record in "inventory" is declared to be 500 bytes.

The "open" statement, without the "new" or "old" modifier, will first attempt to open an existing file. If no file can be found, one will be created with the specified name. Files opened with this form of the "open" statement are opened for both reading and writing. For example:

```
10 open "names" as 1 size 25
```

This statement causes Basic to first search for the file "names" in the current working directory. If the file is found, it is opened for both reading and writing. If the file is not found, one is created with the specified name. In either case, the size of each record in "names" is declared to be 25 bytes.

## 10.2 Closing Record I/O Files.

Record I/O files are closed with the "close" statement in the same manner as are sequential files. This is described in the chapter that introduced disk files. All record I/O files should be closed by the program before it terminates.

## 10.3 Accessing Records

Data is read from and written to record I/O files one record at a time. Records may be accessed randomly or sequentially. Once a file has been opened on a particular channel, the "get" and "put" statements are used to transfer data to and from the file.

### 10.3.1 Reading/Writing Records - The "get" and "put" Statements.

Data is read from a record I/O file by using the "get" statement. The "put" statement is used to write record to a record I/O file. The general forms of these statements are:

```
<line number> get #<expression> [,record <expression>]
<line number> put #<expression> [,record <expression>]
```

Some examples:

```
10 get #1, record 56
20 put #5, record 18
30 get #2
```

The first expression specifies the channel number associated with the file being read or written. This number must be the same as that used in the "open" statement for the file. The "record" portion of each line is optional, and if used, the expression following it specifies the number of the record that is to be read or written. If the "record" option is not specified, the next sequential record in the file will be read or written. The "get" statement reads the selected record from the disk file into a buffer, internal to Basic, associated with the channel number used. The information in the record is accessed through variables that have been associated with the buffer through the "field" statement, described later on. The "put" statement writes the contents of the internal buffer to the specified record of the file. Information is stored in the buffer by using the "lset" and "rset" statements, as described later on.

Records in a record I/O file are numbered starting with 1. The maximum possible record number is 2,147,483,647. This is larger than the maximum possible size of a UniFLEX file.

When a file is initially created, it is empty; none of the records exist. Trying to "get" a record which does not exist will result in an error number 24. This error may be handled with the "on error goto" mechanism. When reading records sequentially, this error indicates that the end of the file has been reached.

Records being written need not be "put" in any particular order. Using "put" to write a record which already exists will cause the new information to replace the old information. If the record being written does not already exist, it is automatically created and the information is stored in it. Using "put" to write a record that does not yet exist causes all records below that one to "exist". For example, assume that a file is empty, that is, it contains no records. Now assume that the first "put" operation on the file is a "put" of record number 10. This would cause records 1 through 9 to become defined automatically, and they may be accessed with a "get" command. Such "automatically defined" records contain only zeroes.

When a record I/O file is initially opened, it is positioned in front of record number 1. Thus, it is possible to read a record I/O file sequentially by using "get" statements without specifying any record number. Each such "get" statement executed will then read the next record in the file.

As an example of "get" and "put" usage, the following program copies record 25 into record 26.

```
10 open "test5" as 2 size 78
20 get #2, record 25
30 put #2
40 close 2
```

The "get" statement in line 20 reads record 25. Since the "put" statement in line 30 does not specify a record number, it will write the next sequential record, which is record 26.

### 10.3.2 Record Locking and Unlocking

Because UniFLEX is a multi-user and multi-tasking system, it is possible for more than one user or task to want to access the same data at the same time. It is even possible for a single Basic program to have the same file open on more than one channel, simultaneously. If the data is only being read, there is no problem. If, however, two or more tasks want to read a record, change it, then write it back to the file, a means of interlocking the record while it is being changed must be available. This data interlock is performed automatically by Basic.

Whenever a file is opened for both reading and writing, each "get" from that file causes the record to be interlocked from other Basic programs. If a file is opened "old" (that is, as a read-only file), no interlocking of records is performed for that file since it is not possible for the program to change the data in a record. The interlock is automatically cleared by a "put", "seek", or another "get" on the same file. The program may clear the interlock manually using the "unlock" statement, described below. Closing a file also clears any outstanding interlock. At any one time, each task may have only one record in each file interlocked. NOTE: If one Basic program has the same file open on more than one channel simultaneously, still only one record in that file may be interlocked.

If a program has a file open for both reading and writing and attempts to access a record that has been locked by another task, a "locked record" error (number 49) is generated. This error may be handled with the "on error goto" mechanism. Programs written to write record I/O files should be prepared to handle such an error. If a program has a file opened only for reading, it cannot get a "locked record" error.

If a program that has a file opened for both reading and writing gets a record and then decides that it does not want to change the contents of the record, it would be wise to release the interlock on that record. If the program is going to read another record immediately, the interlock will be released automatically. If some time will elapse before another record will be read, the program should manually release the interlock with the "unlock" statement. The general form of the "unlock" statement is:

<line number> unlock <channel> [,<channel> ...]

Any outstanding interlock is cleared for each specified channel. No error is generated if there is no outstanding interlock. For example:

300 unlock 1,2,6

This statement would clear any interlock outstanding on a record in the files associated with channels 1, 2, and 6.

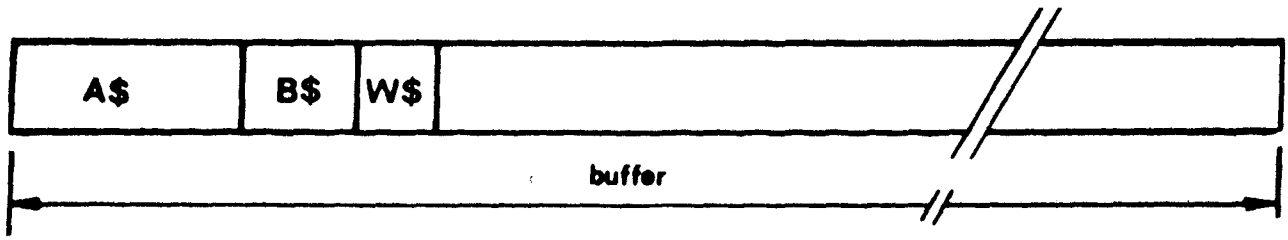
## 10.4 Defining Record I/O Variables - The "field" Statement.

For each record I/O file that is open, there is a buffer associated with it internal to Basic. Each buffer is as long as the declared record size and is used for temporary storage of each disk file record as it is being manipulated. Records are read into and written from the buffer by the "get" and "put" statements described above. The "field" statement is used to associate string variable names with various parts of the buffer. The general form of the "field" statement is:

<line number> field #<expression>, <expr1> as <string var1>
[, <expr2> as <string var2> ...]

The "expression" designates the channel number associated with the file being read or written. "Expr1" is an expression that declares the length, in characters, of the associated string variable and "string var1" is a string variable name for this part of the buffer. As many expressions and names as desired may be listed. Space in the buffer is assigned to the variables in the order that the variables are declared in the "field" statement. Consider the following example and its associated diagram.

100 field #1 20 as A$, 10 as B$, 6 as W$




As shown in the diagram, line 100 would associate the string A$ with the first 20 character positions in the buffer, B$ with the next 10 character positions, and W$ with the next 6 positions. Note that it is not necessary to account for all of the space in the buffer; however, the total number of character positions associated with a buffer must be less than or equal to the declared record size.

Each time a "field" statement is executed, it will start the string association with the first character position in the buffer, regardless of how the buffer has been previously defined with prior "field" statements. Definitions of variables from a previous "field" statement are still intact, however. This allows one to define "synonyms", that is more than one variable pointing to the same part of the buffer. For example:

```
10 open "test" as 1 size 30
20 field #1, 10 as a$, 20 as b$
30 field #1, 10 as x$
```

In this example, both a$ and x$ point to the first ten characters in the buffer.

The "field" statement may be executed either before or after the data record has been read with a "get" statement. It is even possible to "re-field" a buffer with data in it. For example:

```
10 open "test" as 1 size 30
20 field #1, 2 as a$, 5 as b$
30 if a$="aj" then field #1, 2 as a$, 10 as b$
```

In this example, the definition of b$ is changed if a$ contains the string "aj".

Large records may contain more variables than can be defined with a single "field" statement. In this case, multiple "field" statements may be used provided that dummy variables are used to "skip" the portions of the buffer defined by the previous "field" statements. For example:

```
10 field #1, 10 as a$, 5 as b$, 7 as c$, 8 as d$
20 field #1, 30 as x$, 9 as e$, 5 as f$, 20 as g$
30 field #1, 64 as x$, 10 as h$, 12 as i$
```

Line 10 accounts for the first 30 characters of the buffer. In line 20,

the definition of x$ "skips" those characters that are associated with the variables defined in line 10 and continues by defining 3 more variables, e$, f$, and g$. These two statements account for 64 characters. Line 30 skips these 64 characters, again by using x$ as a dummy variable, and defines 2 more variables, h$ and i$.

Once a variable name has been fielded by using the "field" statement, its value will be whatever is currently in the buffer with which it is associated. If the contents of the buffer change (by executing a "get" statement) the contents of the string will also change. The content of the variables before any data is read, or after the file is closed, is undefined. The length of each string variable will always be the length declared for that variable in the "field" statement.

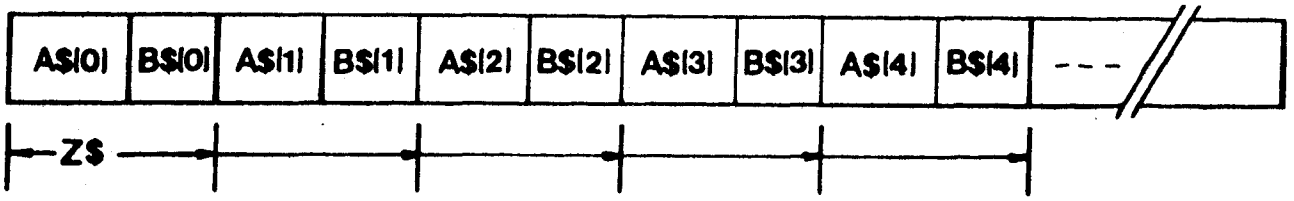
The "field" statement itself does not move any data between variables and the buffer but merely sets up a definition for later use by the "lset" and "rset" statements which perform the actual data transfer. Using a "field" statement to associate a string variable with a buffer is temporary and the definition is nullified by any attempt to assign a value to the string variable by using "let" or the implied "let" statements. As an example:

```
10 open "test" as 1 size 50
20 field #1, 50 as B$
30 B$="Test string"
```

Line 30 does not put the string "Test string" into the buffer but instead removes B$'s association with the buffer and makes it a "normal" string variable. The result is that line 30 nullifies the "field" definition setup in line 20.

It is also possible to use subscripted string variables in a "field" statement (except for virtual arrays, discussed in a later chapter). Consider the following example and its associated diagram.

```
100 dim A$(13), B$(13)
110 for i%=0 to 13
120 field #1, i%*18 as Z$, 10 as A$(i%), 8 as B$(i%)
130 next i%
```




This set of statements will associate each element of the string arrays A$ and B$ with a "sub-record" in the buffer. The variables A$(0) and B$(0) will be associated with the first sub-record, A$(1) and B$(1) with

the second, and so on. Note the use of the dummy variable Z$ to skip elements defined in previous iterations of the loop.

## 10.5 Assigning Values - The "lset" and "rset" Statements.

The "field" statement is used to associate a string with a portion of a record I/O buffer. Once fielded, the content of these strings may be altered by using the "rset" and "lset" statements. These statements are similar in use to the "let" statement. The general forms of these two statements are:

```
<line number> lset <string variable> = <string expression>
<line number> rset <string variable> = <string expression>
```

Where "string variable" represents any legal string variable name, including subscripted variables. These statements store the result of the string expression into the previously defined string's space, overwriting the old string. The length of the string is not changed. If the new string is longer than the old one, the new one will be truncated to the same length as the old. If the new string is shorter, "lset" will left justify the string, padding to the right with spaces until the lengths are equal, while "rset" will right justify the string, padding with spaces on the left. The normal use of "lset" and "rset" is with fielded string variables but they may also be used with regular strings. These statements can be used to assign a value to any legal string variable in Basic, following the above rules for padding and truncation. Some examples:

```
100 lset a$="This is a test"
200 rset b$=a$+"."
```

## 10.6 Storing Integers and Floating Point Numbers

The "lset" and "rset" statements allow string data to be stored and retrieved from the buffer associated with a record I/O file. However, it is often desirable to store numeric data in a record I/O file without having to convert the numbers to strings. There are two functions which allow the passing of the binary representation of integers and floating point numbers to the "lset" and "rset" statements. Two additional functions are provided for retrieving integers and floating point numbers from fielded strings. These four functions do not perform any actual conversion of data. There are merely a means for making binary

numbers "look like" strings. The syntax of each is demonstrated through the following examples.

```
lset a$ = cvtf$(x)
lset b$ = cvt%$(i%)

x = cvt$f(a$)
i% = cvt$%(b$)
```

The first function maps a floating point number into an eight character string; the second, an integer into a two character string. Each character in the resulting string is actually one of the bytes of the number. The second pair of functions reverse the process. The first one maps the first eight characters of a string into a floating point number, and the second maps the first two characters of a string into an integer. If the string has fewer than the required number of characters, null characters (hexadecimal 00) are appended.

The names of the functions are somewhat indicative of their function. Note the the first three letters are all "cvt" and the last two are combinations of "%", "$", and "f". The "%" stands for "integer"; the "$", for "string"; and the "f", for "floating point". The first such character specifies the type of variable being mapped, and the second, the type into which it is to be mapped. Thus, "cvtf$" means "map from floating point (f) to string ($)".

Again, it should be noted that these functions, although sometimes called "convert" functions, do not change the data. These functions merely provide a compact and fast way of storing numeric data in a record I/O file.

As an example of the correct use of one of these functions, assume it is necessary to store the items of a floating point array in a record I/O file. The following program fragment demonstrates a way of accomplishing this.

```
10 dim a(30), a$(30)
20 open "fpdata" as 1 size 248
30 for i%=0 to 30
40 field #1, 8*i% as d$, 8 as a$(i%)
50 next i%
    .
    .
    .
200 for i%=0 to 30
210 lset a$(i%) = cvtf$( a(i%) )
220 next i%
230 put #1
240 close 1
230 end
```

After the variables are dimensioned and the file is opened, lines 30

through 50 specify the "field" definition for the array in the record I/O buffer. Each element of the string array a$ is associated with an eight byte section of the buffer. It is assumed that the array "a" is assigned values between lines 50 and 200. Lines 200 through 220 copy the numeric array into the fielded string array. Notice that the "cvtf$" function is used to store the floating point elements of the array "a" into the buffer. Line 230 writes the record to the disk file and line 240 closes the file.

Another short example will demonstrate the record I/O tools described thus far.

```
10 open "data5" as 1 size 120
20 field #1, 10 as a$, 40 as b$, 70 as c$
30 get #1, record 15
40 print a$
50 print b$
60 print c$
70 lset a$="New a$"
80 lset b$="New value for b$"
90 rset c$="New right justified c$"
100 put #1, record 15
110 close 1
```

Line 10 opens the file "data5" in the current working directory. Line 20 defines a$, b$, and c$ to be in the record I/O buffer. Line 30 reads record number 15 into the buffer. The fielded string variables now contain the values from that record, so the "print" statements in lines 40 through 60 will print their values. The strings printed reflect what is contained in those character positions of record 15. Lines 70 through 90 assign new values to these strings using the "lset" and "rset" statements. Remember that the new strings are actually being stored in the record I/O buffer. Line 100 writes the buffer back into record number 15 of the data file. If line 100 were omitted from this program, the file would remain unchanged. The file is finally closed in line 110.

## 10.7 The "seek" Statement

The "seek" statement is an interface to the UniFLEX "seek" system call (described in the UniFLEX Operating System manual). This statement may only be used on Basic record I/O files. An analogous statement, "position", is available for use with Basic sequential files. The "seek" statement allows a Basic program to position a record I/O file in front of any record in the file. The most frequent uses of this statement are to 1) determine the number of records in a file, 2) skip records, forwards or backwards without having to know the current record number, and 3) determine the current record number.

The general form of the "seek" statement is:

```
<line number> seek #<channel> ,<record number/offset>
                    [,mode <mode number>] [,response <variable>]
```

The "channel" is the channel number associated with the file being positioned. The file must already have been opened with an "open" statement.

The "record number/offset" is the record number in front of which the file will be positioned for mode 0 seeks, and an offset (record count) for mode 1 and mode 2 seeks ("mode" is described below).

The "mode number" is the same as the UniFLEX "seek mode" as described in the UniFLEX Operating System manual. This number may take on the values 0, 1, or 2. Mode 0 indicates that the positioning is to take place from the beginning of the file; mode 1, from the current position; and mode 2, from the end of the file. If no mode is specified, mode 0 (from the beginning of the file) is assumed.

The "variable" specified as an argument to the "response" portion of the "position" statement must be a floating point variable. If a "response" is requested, by specifying "response" and a variable name, the number of the record in front of which the file is positioned is stored in the specified variable. If no response is requested, none is returned.

The following examples illustrate the most common uses of the "seek" statement as outlined above.

### 10.7.1 Determining the Number of Records in a File

The following statement will return the number of the last record in the file on channel 1. Since records are numbered starting at 1, this is the same as the number of records in the file.

```
100 seek #1, -1, mode 2, response r
```

This statement requests that a position to 1 record in front of the end of the file be performed. The response variable "r" wil contain the number of that record.

If it is desired to add records to the end of a record I/O file, the following statement will position the file after all of the records.

```
100 seek #1, 0, mode 2
```

Records may now be added by specifying a "put" with no record number.

### 10.7.2 Skipping Records

Records may be skipped on a record I/O file, without having to know the current record number, by using a mode 1 "seek" statement. For example:

```
100 seek #1, 5, mode 1
200 seek #1, -10, mode 1
```

Line 100 will skip 5 records forward, and line 200 will backspace over ten records. An error is generated if a seek backwards will go beyond the beginning of the file.

### 10.7.3 Determining the Current Position

By using the "response" request feature of the "seek" statement, it is possible to determine the current position of a file, and return to that position later on in the program. For example:

```
100 seek #1, 0, mode 1, response r
...
500 seek #1, r
```

Statement 100 causes a seek to zero records after the current record (which does nothing), and requests that the record number be stored in the variable "r". Statement 500 causes the file to be positioned to the location that was remembered in variable "r".

## 10.8 Record I/O Example

The following short program demonstrates some of the record I/O statements. It works with an existing employee file which contains information about each employee. Each record of the file contains information about one employee. Each employee has a number which corresponds to the record number in the file. This program is used to print a selected employee's phone number and change it if desired. The employee's name is stored in the first 20 characters of each record and the phone number is in character positions 90 through 104.

```
10 on error goto 150
20 open "employee-data" as 1 size 200
30 field #1, 20 as n$, 69 as d$, 15 as p$
40 input "Enter employee number",e%
50 if e%<=0 then 40
60 get #1, record e%
70 print n$,p$
80 input "Change number",r$
```

```
90 if left$(r$,1)<>"y" then unlock 1:goto 40
100 input "New number",a$
110 lset p$=a$
120 put #1, record e%
130 print "Number changed"
140 goto 40
150 if err=8 and erl=40 then close 1:end
160 if err<>24 then on error goto 0
170 print "No record for that employee"
180 resume 40
```

Line 20 opens the employee file on channel 1. Line 30 fields the variables such that n$ points to the employee name and p$ points to the phone number. The variable d$ is used as a dummy variable so p$ is positioned correctly in the buffer (we have skipped data in the buffer). Line 40 prompts the user to enter the employee number. If the number is negative, the user will be prompted for the number again. Line 60 reads in the selected employee record and line 70 prints the name and phone number. If the number does not need to be changed, line 90 unlocks it and goes back to request another employee number. If it does need to be changed, line 100 inputs the new number, line 110 puts it in the record I/O buffer using "lset", and line 120 writes the updated record back to the disk file (which automatically unlocks it). An error handling routine starts at line 150. Here, if a "control-d" was typed in response to the prompt in line 40, the programs closes the file and terminates. Line 160 checks the error for being the "record not found" error. If it is not, the program lets Basic process the error. If it is the "record not found" error, a message is issued and control is return to the prompt for a new employee number.

# Chapter 11.

# Virtual Arrays

The virtual array mechanism allows the program to declare that a data array be stored in a disk file rather than in memory. The two advantages of this feature are 1) the array may be much larger than what would fit in the available memory, and 2) the data in the array remains after the program terminates and may be used at a later time by another program. Virtual array data is referenced exactly like standard memory array data.

## 11.1 Opening a Virtual Array File

A virtual array file is opened as a record I/O file. While not required, a "size" specification should be used to define the buffer within Basic to be the size of one element of the virtual array. Thus, "size 8" is used for floating point arrays; "size 2" for integer arrays; and the size of a single string for string arrays. Larger buffer sizes do not speed up virtual array processing; any excess buffer space is not used. Here are some examples of "open" statements for virtual array files:

```
10 open "floating-data" as 1 size 8
20 open "integer-data" as 2 size 2
30 open "string-data" as 3 size 48
```

Line 10 opens a file that contains (or will contain) a virtual array of floating point numbers. Line 20 opens a file that will contain integers. Line 30 opens a string array. The size of 48 should be the size of an individual string that will be stored in the array.

## 11.2 Declaring a Virtual Array

Before being used, a virtual array must be declared by using a modified form of the "dim" statement. The general form is:

```
<line number> dim #<expression>, <variable>( <dimension> )
```

The expression designates the channel number of the file on which the virtual array resides. The file must already be opened by an "open" statement when the "dim" statement is reached. Only one virtual array may be associated with each channel number. The array may have either

one or two dimensions. As an example:

```
20 dim #3, a(100,50)
30 dim #4, b%(100)
```

Line 20 would define the matrix "a" to be 101 by 51 in size and be associated with channel 3. Line 30 defines b% to be a vector with 101 elements associated with channel 4.

Virtual arrays may be floating point, integer, or string. All data items are stored in the file in "internal format", 2 byte binary for integers, 8 byte binary for floating point numbers, and ASCII characters for strings.

For the most part, virtual array and standard memory array manipulations are identical. One difference is in the way string storage is performed. In a memory string array, the data items may be any length and change in size as the program executes. A virtual string array requires each string in the array to have the same length. The maximum length allowed is 16383 characters but may be defined to be anywhere from 1 to 16383. Each string element stored into a virtual array will either have enough spaces attached to the right side of the string to make it equal in length to the defined string size (if it is shorter than the defined length), or it will be truncated from the right if it is longer than the defined length. To define the string length for a particular virtual string array, the following form of the "dim" statement should be used.

```
<line number> dim #<expr>, <string var and dimension>=<expr>
```

The equals sign and expression define the string length. For example:

```
100 dim #7, a$(100)=63
```

This example defines the virtual string array a$, which has 101 elements, each of which are 63 characters in length. If a string length is not specified in the "dim" statement, a default length of 18 characters is used.

## 11.3 Using Virtual Arrays

Before a virtual array may be used, the corresponding disk file must be opened using the "open" statement as described above, then the array must be defined using the modified "dim" statement. Once this has been done, the virtual array may be used in assignments and expressions exactly like any standard array. A short example will demonstrate virtual array use.

```
10 open "datafile" as 1 size 8
20 dim #1, a(100)
30 input "Type array element number and new value",e,v
40 print "The old value is";a(e)
50 a(e)=v
60 print "The new value is";a(e)
70 close 1
80 end
```

Line 10 opens the file named "datafile" in the current directory. If the file does not already exist, one will be created (because of the type of "open" statement used). Line 20 defines the virtual array "a" which is a floating point array containing 101 elements (counting element 0). From this point on, the program treats the array "a" just as if it were a memory array. Line 30 requests which element in the array is to receive a new value and what that value should be. Line 40 prints the current value of that data item, line 50 gives it the new value, and line 60 prints the new value of the selected array item. Notice that the array reference is purely random, and that it was not necessary to access the file sequentially. Line 70 closes the file (as is necessary after completing the use of any disk file). The file "datafile" now exists on the disk with the selected data item altered to reflect its new value.

## 11.4 Notes on Virtual Arrays

A file created or referenced as a virtual array has no information contained in it which specifies its dimension or data type. If the file was created as a floating point virtual array, it consists of 8 byte binary numbers, one right after the other. If it was created as a string array, it consists of one string after the other, each of the length specified in the "dim" statement. If the array is integer, it consists of two byte binary numbers, one right after the other. A double dimensioned virtual array is stored in row form. This means that row 0 is closest to the beginning of the file, followed by row 1, etc.

Since a virtual array data file is just a collection of data items in the file, one right after the other, the file may be dimensioned in any way desired. It usually only makes sense to dimension the array in the same way it was dimensioned when it was created, but this is not necessary. As an example, suppose a virtual array was created and dimensioned as (50,50). The array would potentially contain (50+1) times (50+1) data items or a total of 2601 elements. Normally, when referencing this array file at a later date, the same dimensioning would be used. We could however, just as easily dimension this existing array file as a single dimension array of 2060 (2061 elements). It could also be dimensioned as a one dimension array containing only 100 items, which would result in the remaining elements being inaccessible.

When a virtual array file is closed in a program, the array becomes "undimensioned". If the virtual array file is reopened, the array must be dimensioned again.

Virtual array elements may not be used as arguments to the "swap" statement.

When a virtual array file is initially opened for the purpose of creating the array, all of the elements in it are zero. In the case of a string virtual array, this means that each individual array element is a string of length as declared in the "dim" statement whose content is null characters (ASCII 00). This is not that same as the null string or a string of spaces.

If a file is opened, used as either a sequential or record I/O file, then declared to contain a virtual array, an error 44 will result. It is necessary to close the file then re-open it before declaring it as a virtual array file if it has been previously used as a sequential or record I/O file in the same program.

# Appendix A

## Writing UniFLEX Utilities in Basic

Basic has enough features to allow the writing of UniFLEX untility programs. The UniFLEX "shell" program can recognize a Basic "compiled" program on a disk file and will automatically load Basic from the "/bin" directory, passing it the name of the file containing the "compiled" program. The statements and functions described in the chapter "System Interface Features" may be used to decode arguments passed to the program from the command line. The "exit" statement is used to terminate both the program and Basic. Of course, Basic is an interpreter, and as such is slower than programs written in assembly language. Thus, Basic should not be used to write utilities that will be executed frequently, or that require high execution speed.

Since utility programs written in Basic may contain "exit" statements, it is important to remember to "save" the program after making any changes and before testing it since when the "exit" statement is encountered, Basic itself will terminate. An alternative to this is to use "end" or "stop" statements instead of "exit" statements while the program is being tested. Once the program works correctly, they may be replaced by "exit" statements.

When testing programs that require arguments, it is necessary that they be passed to the program when Basic itself is called. This may be done by saving the program in a disk file, exiting Basic, and then calling Basic, specifying the name of the file containing the program, followed by any arguments. For example:

```
basic file-name argument-1 argument-2 ...
```

Once the program is working correctly, it should be "compiled" with the "compile" command. The user should then set the "execute" permission bits on the file containing the "compiled" program so that it may be executed. Once this is done, the program is called by merely typing the name of the file containing the program followed by any arguments. For example:

```
file-name argument-1 argument-2 ...
```

The following program is an example of a utility program written in Basic. This program reads the "history" file maintained by the operating system, extracts information relating to the logon and logoff of individual users and prints a report of each such session within the past few days. The report lists the logon time, logoff time, and number of seconds of connect time used for each session. If no arguments are specified, the report is issued for all users. User names may be

specified as arguments. In this case, the report will be only for those users whose names are so specified. Additional information on the "history" file is found in the documentation of the "history" command in the "UniFLEX Utility Commands" manual, and the "System Manager's Guide".

```
10 n%=-1
20 if argc%<=1 then 80
30 n%=argc%-2
40 dim n$(n%)
50 for i%=0 to n%
60 n$(i%)=arg$(i%+1)+string$(chr$(0),10-len(arg$(i%+1)))
70 next i%
80 s=second-96.*60.*60.
90 f=1
100 nu$=string$(chr$(0),10)
110 on error goto 390
120 open old "/act/history" as 1 size 16
130 field #1,2 as c$,10 as n1$, 4 as t$
140 on error goto 520
150 get #1
160 if c$<>"ad" then 150
170 gosub 410
180 seek #1, 0, mode 1, response r
190 if t<s then f=r:goto 150
200 seek #1, f
210 dim ta(11),tu$(11)
220 on error goto 560
230 get #1
240 on error goto 590
250 j%=val(c$)
260 gosub 410
270 if n1$<>nu$ then 310
280 if ta(j%)=0 then 220
290 gosub 460
300 goto 220
310 if n%<0 then 360
320 for i%=0 to n%
330 if n1$=n$(i%) then 360
340 next i%
350 goto 220
360 tu$(j%)=n1$
370 ta(j%)=t
380 goto 220
390 print "Cannot open history file."
400 exit 255
410 t=0
420 for i%=1 to 4
430 t=256.*t+asc(mid$(t$,i%,1))
440 next i%
450 return
460 print tu$(j%),"from ";time$(ta(j%))
470 print spc(1),"to   ";time$(t)
```

```
480 print using "                    Time used: #######",t-ta(j%);
490 print " seconds"
500 ta(j%)=0
510 return
520 if err<>24 then 570
530 print "Cannot find a recent date in the history file."
540 close 1
550 exit 255
560 if err=24 then close 1:exit
570 print "Error";err;"reading the history file."
580 goto 540
590 resume 220
```

The "history" file contains 16 byte binary records. The first two bytes contain a two character code indicating the type of record. The last four bytes contain the date expressed as a 32-bit count of the number of seconds that have elapsed since January 1, 1980. Those records that we will be concerned with are those that start with "ad" which indicate a date change, and those that consist of two digits, which indicate a terminal logon or logoff. The logon record contains the name of the user in bytes 3 through 12.

The following is a line-by-line description of the program.

10 The variable n% is used as a flag to indicate if arguments have been specified. If negative, no arguments were on the command line. If positive, it is one less than the number of arguments, and is used as the dimension of an array that will be used to hold them. Here, it is preset to indicate "no arguments specified".

20 If the system variable "argc%" is less than 2, then no arguments were specified, and the argument processing is bypassed.

30-40 If arguments were specified, n% is set to one less than the number of arguments specified and an array n$ is dimensioned to hold the arguments.

50-70 When the user names are extracted from the history file record, they will have trailing null characters. In order to make the comparison with the arguments easier, these statements copy the arguments into the array n$, appending sufficient null characters to make the name length equal to ten (which is the size of the name field in the history file record).

80 The variable "s" is initialized to contain the date and time as it was 96 hours ago. Since the history file may be quite long, we do not want to print information about sessions held in the distant past. We will collect data from the first occurrence of an "ad" record preceding the date and time specified by "s".

90 The variable "f" will contain the number of the record with which we will start collecting data. Here, it is preset to record number 1, the start of the file.

100 The string nu$ is defined to contain 10 null characters. This will be used to distinguish a logon record (which has a user name) from a logoff record (which has no user name).

110-130 These lines open the history file and define the record format. The variable c$ contains the two character code; n1$ the user name (if any); and t$, the date and time.

140-160 These lines search for an "ad" record. An "ad" record is produced when the system manager sets the date by using the UniFLEX "date" command. The history file must contain at least one "ad" record specifying a date and time more recent than 96 hours ago for this program to work.

170 An "ad" record has been found, so a subroutine is called to convert the 32-bit integer in t$ to a floating point number.

180 The "seek" statement is used to determine which record is next. The statement does not reposition the file, but merely requests that the number of the next record be returned in the variable "r".

190 If the "ad" record refers to a date and time preceding that in "s", we remember the next record number in "f" and continue looking for "ad" records.

200-210 Here we have found an "ad" record specifying a time more recent than that contained in "s". We use the "seek" command to reposition to that record immediately following the previous "ad" record (or record 1 if there was no previous "ad" record). We also define two arrays to keep track of up to 12 terminals. Array "ta" will contain the logon time of that terminal and "tu$" will contain the user's name.

220-230 These statements begin the main processing loop. Here, individual records are accessed. The "on error statement" is used to detect the end of the file.

240-250 These statements check the record code for a number. This is done by attempting to convert the code from ASCII to a number by using the "val" function. If the function does not produce an error, then the code was indeed a number and this number is then stored in "j%". If the function produces an error because the code did not contain digits, the error handling routine causes the record to be skipped.

260-270 A logon or logoff record has been found. The time is converted to a floating point number and the user name in the record is checked. If it is not null, it is a logon record; if it is null, a logoff record.

280 A logoff record has been found. If the array entry for this terminal is zero, no associated logon was found and the record is ignored.

290-300 A subroutine is called to print the information, and control returns to the main loop to look at the next record.

310 Here we have found a logon record. If the flag, n%, is less than zero, we skip the part of the program that scans the name array, n$, since no arguments were passed to the program.

320-350 This loop searches the arrays of names extracted from the arguments to the program. If the name is not found, the record is skipped.

360-380 The logon record is processed here. The user name is stored in tu$, and the logon date and time in the array "ta". Then control return to the main loop to process the next record.

390-400 This is an error handling routine that is reached only if an error is detected when trying to open the history file. It prints a message and the terminates with an error code of 255. This error code is not a valid UniFLEX error code, but will suffice to indicate abnormal termination of the program.

410-450 This is the subroutine that converts the 32-bit time into a floating point number.

460-510 This is the subroutine that prints the report. When finished, the logon time is zeroed to indicate that the terminal is not in use.

520-550 This is an error handling routine that is entered if the search for "ad" records encounters an error. If the error is not a "record not found" error, the error number is printed, and the program terminates abnormally. If it is the "record not found error", then no date and time more recent than 96 hours ago was found in the file. An appropriate message is printed and the program terminates abnormally.

560-580 This is the error handling routine that is used when searching for logon and logoff records. If the error is "record not found", the program is finished and exits normally. If the error is not "record not found", then the error number is printed and the program terminates abnormally.

590 This is the error handling routine that is used in trying to determine if the record code is composed of digits or not. It is reached if the record code is not numeric and merely resumes execution at the point where the next record is read.

Assume that the above program has been typed in and saved on a file called "usage.source". Then to create the utility, which we will call "usage", invoke Basic and type the following:

```
load "usage.source"
compile "usage"
+perms o+x u+x usage
exit
```

The program in the file "usage" may now be run merely by typing:

```
usage
```

If it is desired, for example, to check on the usage of the system made by the users named "otto" and "derek", then type:

```
usage otto derek
```

Again, it should be stressed that Basic should not be used to write heavily-used utility programs. However, it is possible to write useful utilities in Basic in a short time.

# Appendix B

## Error Number Summary

This appendix contains a summary of the error numbers that are issued by Basic. Error numbers greater that 200 are UniFLEX-related errors. The appropriate UniFLEX error number is determined by subtracting 200 from the number issued by Basic.

Error Numbers and Their Interpretation

1-2 Not used.

3 File already exists.
Basic referenced a file, expecting it to not exist; however, it already existed. An example would be using the "rename" statement where the new name was already in the user's directory.

4 File does not exist.
Basic referenced a file, expecting it to exist, but it could not be located. An example would be opening a file as "old" and it does not exist. Renaming a non-existant file would also result in this error.

5-6 Not used.

7 Disk full.
All of the free space on the disk being written is used up. Files will have to be deleted to make more space available.

8 End of file.
A read was performed on a disk file and there was no more information to be read. This error may also be caused by typing "control-d" in response to a request for input from the terminal.

9 Read error.
A UniFLEX "i/o error" was detected when trying to read from a file. This is most likely due to a checksum error on a disk file.

10 Write error.
A UniFLEX "i/o error" was detected when trying to write to a disk file. This is most likely caused by a damaged sector on the disk.

11 Write protected.
The program attempted to open a file for writing, but the permission flags for the file do not allow writing on the file. This error also occurs when trying to write on a file that was opened for "read-only".

12 File protected.
An attempt was made to perform an action on a file that the permission flags for the file did not permit. An example is the opening for reading of a file whose permission flags do not allow reading of the file by the user.

13-20 Not used.

21 Illegal file specification.
A null string, or a numeric variable or constant was specified as the file name in an "open" statement.

22-23 Not used.

24 Nonexistent record.
A "get" statement was performed for a record that was beyond the end of the file, or a negative record number was specified in a "get" or "put" statement. This error may also be caused by a "seek" statement that would cause a position to a point in front of the beginning of the file.

25-27 Not used.

28 File must be a sequential file.
An "input", "print", or "position" statement, or an "inch$" function was invoked on a file that had previously been used as a record I/O file.

29 Empty file.
A "load" or "run" command specified an empty file.

30 Data type mismatch.
An "input" or "read" statement was given data that did not correspond to the variable to which the data was to be assigned. For example, a string was typed when a number was expected.

31 End of data list for "read".
A "read" statement was executed, but all of the "data" statements had already been used.

32 Bad argument to "on".
The value specified in an "on-goto" or "on-gosub" statement was either zero or larger than the length of the list of line numbers.

33 Not used.

34 "Control-c" typed.
A "control-c" interrupt was typed from the terminal.

35 Wait time exhausted.
A "timed" input was being performed (by having specified a non-zero value to the "wait" statement), but no input was received before the time limit elapsed. Input is considered to have been received when the carriage return is typed. In the case of the "inch$" function, input is considered received as soon as any character is typed.

36-39 Not used.

40 Bad channel number.
A channel number was specified in a file-related statement that was larger than the maximum allowed channel number (12).

41 File already open.
An attempt was made to open a file on a channel that is already associated with an open file.

42 Line too long.
A line in a file being read with the "load" command is longer than 255 characters.

43 File has not been opened.
A file-related statement referred to a file that has not been opened by an "open" statement.

44 Virtual array file previously used.
A "dim" statement declaring a virtual array refers to a file that has been previously used by the program.

45 Field size too large.
The sum of the sizes declared in a "field" statement exceeds the record size specified on the "open" statement.

46 Buffer size error.
A buffer size, specified with the "size" option on a "open" statement, was either negative, zero, or greater than 16383.

47 Record zero not allowed.
A "get", "put", or "seek" statement was encountered that referenced record number zero. Record I/O files start with record number one.

48 File must be a record I/O file.
A "put", "get", or "seek" statement was issued on a file that was previously used as a sequential file (with "print", "input", or "inch$").

49 Record is locked.
An attempt was made to "get" a record that is interlocked by another program. The program getting this error may wait, if desired, for the interlock to clear.

50 Unrecognizable statement.
The first item on a line cannot be recognized as a valid Basic keyword or variable name.

51 Illegal character in line.
A program being loaded from a disk file with the "load" command contains a line that does not start with a line number. This error also occurs if a line contains a punctuation character that is not part of the Basic character set.

52 Syntax error.
A general error message issued when a statement does not have the correct syntax.

53 Illegal line termination.
The end of a statement has been reached, but the next item on the line is not a carriage return or statement separator.

54 Line number zero is not allowed.
A statement was typed with a line number of zero.

55 Unbalanced parentheses.
An expression does not have balanced parentheses.

56 Illegal function reference.
A user-defined function starts with a number; for example "fn4".

57 Missing quotation mark in string constant.
A string constant does not have an ending quotation mark.

58 Missing "then" in an "if" statement.
An "if" statement is not followed by "goto" or "then".

59 Not used.

60 Line not found.
A statement contained a line number that could not be located in the Basic program. If there is an "on error goto" statement referring to a line that does not exist in the program, the "Error 60" is generated whenever some other error occurs and Basic tries to find the error routine. In this case, the line number printed in the error message is that of the line that contained the original error, not that of the "on error goto" statement.

61 "Return" without a "gosub".
A "return" statement was encountered without a previous "gosub" or "on-gosub" having been executed.

62 "For-next" nesting error.
A "next" statement was encountered with a variable name that does not match the variable name of the most recent "for" statement executed.

63 Cannot "continue".
A "cont" command was issued when Basic was in a state where continuation is not possible. If a "control-c" is typed when Basic is not expecting input, the program cannot be "continued". Even if a "control-c" was typed when Basic was expecting input, the program cannot be "continued" if new lines are added, a variable name other than any in the program is referenced, or the "renumber" or "+" commands are used.

64 Source not present.
A command was typed that requires that the Basic program in memory not be a "compiled" program. "Compiled" programs do not have the original source associated with them. The commands that may result in this error are: save, list, compile, and trying to change a line.

65 Tried to run a bad "compiled" file.
An attempt was made to run a "compiled" program from a file, but the file did not contain a "compiled" program.

66 "Resume" not in an error handling routine.
A "resume" statement was encountered without there having been an error causing a jump to an error handling routine.

67 Cannot change scale factor.
The "scale" command was entered when there was a program in memory. The scale factor can be changed only when there is no program loaded.

68-69 Not used.

70 Data type mismatch in "print using" statement.
The type of the item being printed (string or number) does not correspond to that specified in the "print using" format. For example, trying to print a string variable when the format expects a number will generate this error.

71 Illegal format in "print using".
A string of special format characters, encountered in a "print using" format, was contradictory. For example, specifying more than one decimal point in a numeric field, specifying floating dollar sign and scientific notation, etc.

72 Mixed mode error.
A number was expected in an expression, but a string was found.

73 Illegal expression.
An error was detected in an arithmetic or string expression. This may be due to an inappropriate operator (eg. using exponentiation between two strings) or concatenated or missing operators (eg. a+*b).

74 Argument <0 or >255.
A statement required a value between 0 and 255, the value supplied was outside of this range.

75 Not used.

76 Illegal variable type.
A statement's syntax required a specific type of variable, but none was provided. For example, "input line" requires a string variable. If a floating point or integer variable is specified, this error will result.

77 Array reference out of bounds.
A subscript in an array variable was larger than the dimension of the array.

78 Undimensioned array reference.
A subscripted variable was encountered without its having been seen in a "dim" statement.

79 Bad variable specified in "swap" statement.
A virtual array element was specified or the two variables are not of the same type.

80 Memory overflow.
There is not enough memory available to load or run the program. This may be due to having too many or too large arrays in memory. Large arrays should be made into virtual arrays.

81 Not used.

82 Stack overflow.
There is not enough stack space available to run the program. This may be caused by too many nested subroutines and "for-next" loops. Additional space for the stack may be obtained by shortening the program. (If the program is to be shortened, at least 4096 bytes must be made available before the stack space will increase.)

83 String too long.
The maximum size of a string is 32767 characters.

84 Subscript too large for virtual array.
The maximum subscript on a virtual array is determined by the maximum size of a file under UniFLEX. The subscript may not be so large that the data will not fit into a single UniFLEX file. This maximum subscript value depends on the size of the individual array elements (8 byte for floating point, 2 bytes for integer, some constant for strings).

85 Mode specification error.
The "mode" portion of the "position" and "seek" statements must be a number between 0 and 2 inclusive.

86-89 Not used.

90 User-defined function error.
A user-defined function was called without having been defined, or an error was detected in evaluating its argument.

91-93 Not used.

94 Bad string length specified.
A negative string length was defined in the declaration of a string virtual array.

95 Virtual array element larger than buffer size.
The buffer size declared on the "open" statement is not large enough to hold a single virtual array element. Integer elements require two bytes; and floating point elements, eight bytes. String elements require that the buffer size be declared to be at least as large as the string length declared in the "dim" statement.

96-99 Not used.

100 Expression too complex.
A table internal to Basic has overflowed during the processing of an expression. The expression should be broken down into several statements.

101 Overflow or underflow in floating point operation.
A floating point operation or function call has resulted in a non-zero number (maybe an intermediate result in a function calculation) that is larger or smaller than is allowed in Basic.

102 Exponent overflow in "exp".
The argument passed to the "exp" function is too large. The largest value that may be passed to the "exp" function is just over 88.

103 Division by zero.
An attempt was made to divide by zero.

104 Number too large to convert to integer.
An attempt was made to convert a number from floating point to integer, but the number was larger than 32767 or smaller than -32768. Note that this error may occur due to internal calculations within some of the scientific functions. If this should occur, it is because the argument to the function is too large.

105 Attempt to take the log of zero or a negative number.
The "log" function was passed an argument that was less than or equal to zero.

106 Line number too large.
A statement that refers to a line number that is greater than 32767.

107 Square root of a negative number.
An attempt was made to take the square root of a negative number.

108 Too many digits specified.
A number was typed with too many digits. Numbers should not contain more than 16 digits.

109 Overflow or underflow in an integer operation.
An integer arithmetic operation was performed whose answer could not be represented as an integer (value between -32768 and 32767, inclusive).

110-200 Not used.

All error number greater that 200 are the result of errors detected by UniFLEX. To determine the UniFLEX error number, subtract 200 from the error number printed by Basic.

Index