North Star BASIC

Version 6
Version 6-FPB

INTRODUCTION

North Star BASIC was implemented by Dr. Charles A. Grant and Dr. Mark Greenberg of North Star Computers, Inc. This manual describes Version 6, an extended disk BASIC intended for use with the North Star MICRO DISK SYSTEM. Version 6 includes such features as multiple-dimensioned arrays, strings, multiple-lined functions, formatted output, and machine language subroutine capability. Programs can be loaded and saved using disk files, and data files may be accessed both sequentially and randomly. Version 6-FPB has been specially assembled for use with the North Star Floating Point Board, for increased speed and reduced memory requirements. [If you order an FPB for use in conjunction with the MICRO DISK SYSTEM, be sure to request that BASIC be delivered on diskette rather than paper tape.] Version 6 has been assembled for 8-digit precision. For accuracy of 2, 4, 6, 10, 12, or 14 digits, a special order may be made. Version 6 is assembled with origin at 2A00 hex. Special orders may also be made for different origins.

This manual assumes familiarity with some version of BASIC on the part of the reader. There are many tutorial and advanced publications on BASIC available in both stores and libraries.

The North Star Version 6 BASIC software is intended for use only with the North Star MICRO DISK SYSTEM, and no license is granted for any other use. Improved copies of Version 6, as they become available, may be obtained for a nominal copying charge.

INPUTTING A PROGRAM

Every program line begins with a line number. Any line of text typed to BASIC in command mode that begins with a digit is processed by the editor. There are four possible actions which may occur:

1) A new line is added to the program. This occurs if the line number is legal (range is Ø thru 65535) and at least one character follows the line number in the line.

2) An existing line is modified. This occurs if the line number matches the line number of an existing line in the program. That line is modified to have the text of the newly typed-in line.

3) An existing line is deleted. This occurs if the typed-in line contains only a line number which matches an existing line in the program.

4) An error is generated. If the line number is out of range, or the line is too long, or the memory would become full, then an error message is generated and no other action is taken by BASIC.

Blanks

Blanks preceding a line number are ignored. Blanks are permitted anywhere in a line for indentation purposes, except within reserved words, the line number, or constants.

Multiple statements per line

Multiple program statements may appear on a single line, if separated by a (\) back slash. A line number must appear only at the beginning of the first statement on the line.

Typing mistakes

If a typing mistake occurs during the entering of any line of text to BASIC, there are two possible corrective actions available:

When the user types an (@) at-sign character, BASIC completely ignores all input on the current line being typed in, and types a carriage return. The correct line may then be typed to BASIC.

When the user types a left-arrow (under-line on some keyboards), BASIC will ignore the previously typed character.

Also, the line editor may be used to correct typing errors (see

Appendix 2).

Compatibility

Certain characters, when they appear in programs, are automatically translated into other characters. This is done to minimize the effort of converting programs written for other BASIC systems. In particular, left bracket ([), right bracket (]), colon (:), and semi-colon (;) are converted to left paren, right paren, back slash (\), and comma (,) respectively. This conversion is not done within quoted strings in a program.

SAVING AND RELOADING PROGRAMS

BASIC programs may be loaded and saved using disk files existing on any mounted diskette. The LOAD and SAVE commands are described below. Familiarity with the DOS file naming conventions is required (see the accompanying Disk Operating System manual).

An error will be generated from a LOAD or SAVE command if the specified file is not a currently existing file with type 2. In addition, a SAVE command will give an error if the named file is not large enough to contain the program to be saved, or if the diskette is write-protected.

CONTROL-C

Typing the control-C character (ETX on some keyboards) has the effect of prematurely interrupting BASIC from whatever it is doing. If a LIST is in progress, the listing will be terminated at the completion of the output of the current line. If a RUN or CONT is in progress, then execution will stop after the completion of the currently executing statement.

DIRECT STATEMENTS

When BASIC is in command mode, certain statements may be typed for immediate execution. This is typically used for examining the values of certain variables to diagnose a programming error. Note that an exclamation point (!) may be used as a shorthand way of typing the PRINT reserved word. No direct statement is permitted which transfers control to the BASIC program. Also, DATA, DEF, FOR, NEXT, INPUT, and REM are forbidden.

COMMANDS

RUN <optional line number>
Begin program execution either at the first line of the program or else at the optionally supplied line number.

LIST <optional line number>,<optional second line number>
If no arguments are supplied, then print the entire existing program. If one line number is supplied, then print the program from the specified line number to the end. If two line numbers are supplied, then print the program in the region between the two line numbers.

SCR
Delete (scratch) the existing program and data, in preparation for entering a new program.

REN <optional beginning value>,<optional increment value>
Renumber the entire existing program. If the first argument is not supplied, then 10 is used as the initial statement renumber value. If the second argument is not supplied, then 10 is used as the increment value.

CONT
This command causes execution of a running BASIC program to continue after a STOP statement or after a control-C stop.

LINE <number of characters>
This command defines the line length of the user terminal. The maximum value is 132. The initial value is 72.

NULL <number of nulls>
This command specifies the number of ACSII null characters to output following the output of each carriage return character. The initial value is zero.

LOAD <file name>
This command will load a BASIC program from the specified disk file.

SAVE <file name>
This command will save the current BASIC program on the specified disk file.

EDIT <line number>
This command specifies a line to be editted by the line editor. See Appendix 2 for details.

BYE
This command will exit BASIC and cause control to be transferred to the DOS. The text of the existing BASIC program is not modified, and it is possible to continue BASIC from the DOS later, if desired (see below).

CONSTANTS

Magnitude range: .1E-63 thru .99999999E+63

Constants appearring in programs are rounded to 8 digits if necessary. Internal representation of numbers is binary-coded-decimal.

NAMES

All user defined names are one or two characters long: a letter of the alphabet optionally followed by any digit. For example: A, Z0, and Q9 are legal names. The same name may be used to identify different values, as long as the values they identify are of different types. For example, it is possible to have a scalar variable named A1, an array named A1, a string named A1$ and functions named FNA1 and FNA1$. There is no relationship between these entities.

OPERATORS

Numeric: +, -, /, *, ↑ (or ^ on some keyboards)

Relational: =, <, >, <>, >=, <=
A relational operation gives a 1 (true) or 0 (false) result.

Boolean: AND, OR, NOT
A Boolean operand is true if non-zero, and false if zero. The result of a boolean operation is 1 (true) or 0 (false).

PRECEDENCE OF OPERATORS
All operators can be used in any numeric expression. Higher precedence operators are evaluated first, and operators of equal precedence are evaluated left to right. The operators are listed below in order of increasing precedence.

OR
AND
=,<,>,=,<>,<=,>=
+,-
*,/
↑
NOT, unary minus (-)

File accessing statements are described in a later section. Consult the example programs in the Appendix for questions about the use of a particular type of statement.

LET

The LET is optional in assignment statements. Multiple assignments are not allowed. The statement A=B=0 assigns true or false to A depending on whether or not B equals 0.

IF

An IF statement may optionally have an ELSE clause. A THEN or ELSE clause may be a LET statement, a RETURN statement, another IF statement or a GOTO, for example. Additional statements appearing on a line following an IF will be executed regardless of success or failure (unless a GOTO is executed). If either the THEN clause or the ELSE clause is a simple GOTO, then the GOTO reserved word may be optionally omitted.

```
100 IF A=B THEN 150 ELSE A=A-1
```

FOR

FOR loops may be multiply nested. The optional STEP value may be positive or negative. It is possible to specify values such that the FOR loop will execute zero times. For example,

```
100 FOR J=5 TO 4 \ PRINT J \ NEXT
```

NEXT

A NEXT statement may optionally specify the control variable for the matching FOR statement, as a check for proper nesting.

GOTO

The GOTO statement will transfer program control to the statement with the specified line number.

ON

The ON statement provides a multi-branched GOTO capability. For example,

```
100 ON J GOTO 500,600,700
```

will branch to 500, 600 or 700 depending on the value of J

being 1, 2, or 3 respectively. Other values of J will cause an error.

EXIT

The EXIT statement is identical to a GOTO except that it has the effect of terminating one active FOR loop and reclaiming the associated internal stack memory. It should be used for branching out of a FOR loop.

STOP

The STOP statement will stop program execution and print a message.

END

The END statement will stop program execution without printing a message.

REM

The REM statement may be used for inserting comments into a BASIC program.

READ

The READ statement is used to sequentially access numeric or string values contained in DATA statements.

DATA

The DATA statement is used to specify lists of string and numeric values which can be accessed by the READ statement.

RESTORE

The RESTORE statement may optionally include a line number, specifying where the READ pointer is to be restored to. In the absence of the optional line number, the READ pointer is set to the first line of the program.

INPUT
INPUT1

The INPUT or INPUT1 statement may optionally specify a literal string which is typed on the terminal as a prompt for the input instead of a question mark. To inhibit the echoing of the carriage return at the end of user input, use the INPUT1 statement.

```
100 INPUT "TYPE VALUES: ",V1,V2
```

GOSUB

The GOSUB statement may be used to call a BASIC subroutine. Control is passed to the specified line number. When a RETURN is executed by the subroutine, then control returns to the statement following the GOSUB statement.

RETURN

The RETURN statement is used to return from a BASIC subroutine.

PRINT

The PRINT statement may include a list of expressions, variables, or constants separated by (,) commas. If the list of variables is terminated by a comma, then a final carriage return is not printed. The null PRINT statement will cause only a carriage return to be printed. A semi-colon is equivalent to a comma in the print list. All numeric values are printed in free format, separated by a blank, unless formatting is specified. If a value will not fit on the current output line, then it is printed on the next output line. Advancement of the printer to a specified output position may be accomplished with the TAB function. Formatting may be accomplished by including a "format string" in a PRINT statement (see below). An exclamation point (!) may be used as a shorthand way of typing the PRINT reserved word.

FILL

This statement permits filling a specified byte in the computer memory with a given expression value. For example, FILL 100,J+3 will fill memory byte 100 with the binary encoded value of J+3, truncated to 8 bits.

OUT

This statement permits doing an 8080 or Z80 OUT instruction. For example, OUT 5,3 will perform an OUT 5 instruction with 3 in the computer accumulator.

ARRAYS

Arrays may be dimensioned with any number of dimensions, limited only by available memory, e.g.,

```
100  DIM A(1), B7(5,2,3,4,5,6)
```

Array indexing starts at element 0. Array A in the above example actually has two elements, A(0) and A(1). Use of an undimensioned array causes automatic dimensioning to a one-dimension, 10 element array. Arrays may not be re-dimensioned within a program.

STRINGS

Strings of 8-bit characters may be dimensioned to any size, limited only by available memory, e.g.,

```
100 DIM A$(1),A1$(10000)
```

Note that a string name is a variable name followed by a ($) dollar sign. The value of a string variable after the DIM statement is a full string of blanks. Substrings of a string variable may be accessed as A$(N,M) which is the substring of characters N thru M. For example, if A$ is "ABCDEF" then A$(3,5) is "CDE". Alternatively, A$(N) identifies the substring including characters N thru the last character in the string. A substring reference must specify positions within the current length of a string. The program:

```
100 A$=""
110 A$(2,3)="AB"
```

will give an error in line 110 because A$ has length of less than 3 at the time of the assignment.

If an assigned value is longer than the destination string or substring, then it is truncated to fit. If an assigned value to a substring is shorter than the substring, then the extra characters of the substring are left unmodified. A string variable used before being DIMensioned is given the default dimension of 10. Strings may not be redimensioned within a program. Indexing string variables begins at 1, not 0.

Strings and substrings may be concatenated with the use of the plus (+) operator. (When large strings are concatenated, sufficient free storage must exist to contain the entire concatenated string.)

Strings, substrings and string expressions may be used in conjunction with: LET, READ, DATA, PRINT, IF, and INPUT statements. The string IF statement does alphabetic comparisons

when the relational operators are used, .e.g.

```
100  IF A$+B$<"SMITH" THEN 50
```

When string variables are INPUT, they must not be quoted. When strings appear in DATA statements, they must be quoted. The boolean operators (AND, OR, and NOT) may not be used in string expressions.

For those familiar with another popular method of accessing strings and substrings in BASIC, the following will be of interest:

```
LEFT$(A$,7)     is equivalent to  A$(1,7)
RIGHT$(A$,5)    is equivalent to  A$(5)
MID$(A$,5,7)    is equivalent to  A$(5,7)
```

Also, the North Star string methods may be used to construct a "table of strings". For example, A$(J*10,J*10+9) is the Jth 10-character substring within A$.

USER DEFINED FUNCTIONS

User-defined functions (either of type string or numeric) may be 1-line or multiple line functions. There may be any number of numeric arguments. Parameters are "local" to a particular call of a function. That is, the value of the variable is not affected outside of the execution of the function.

Functions are defined before execution begins (at RUN time), so definitions need not be executed, and functions may be defined only once.

Multiple line functions must end with a FNEND statement. A multiple-line function returns a value by executing a RETURN statement with the value to be returned, for example:

```
100 DEF FNA(X,Y,Z)
200 IF Z=1 THEN RETURN X
300 X=Y*Z+X*3
400 RETURN X
500 FNEND
600 PRINT FNA(1,2,X+Y)
```

BUILT IN FUNCTIONS

```
FREE(Ø)          returns number of bytes remaining in free storage.
ABS(expr)        returns the absolute value of the expression
SGN(expr)        returns 1, Ø, or -1 if the value is +, Ø, or -
INT(expr)        returns the integer portion of the expression value
LEN(string name) returns the length of the specified string
CHR$(expr)       returns a string with the specified character
ASC(string name) returns ASCII code of first character in string
VAL(string expr) returns the numeric value of the string
STR$(expr)       returns a string with the specified numeric value
SIN(expr)        returns SINE of the expression
COS(expr)        returns the COSINE of the expression
RND(expr)        returns a random number between Ø and 1 (see below)
LOG(expr)        returns the natural log of the expression
EXP(expr)        returns the value of e raised to the specified power
SQRT(expr)       returns the positive square root of the expression
CALL(expr,expr)  machine language subroutine call (see below)
EXAM(expr)       return contents of addressed memory byte
INP(expr)        return result of 8Ø8Ø IN to specified port
TYP(expr)        return file item type (see below)
TAB(expr)        Advance tab in PRINT statement
```

MACHINE LANGUAGE SUBROUTINE INTERFACING

The built-in function CALL takes a first argument which is the address of a machine language subroutine to call. The optional second argument is a value which is converted to an integer and passed to the machine language subroutine in DE. The CALL function returns as value the integer which is in HL when the machine language subroutine returns. For example:

```
10 A=CALL(5ØØ)
2Ø B=CALL(X,Y+3)/Z
```

RANDOM NUMBER GENERATOR

The RND function may be used to generate a sequence of pseudo-random numbers between Ø and 1 by the Rawson method. If the argument to the RND function is Ø, then the next pseudo-random number in the sequence is returned as value. Otherwise, if the argument expression is a value between Ø and 1, the sequence will be restarted with the supplied value as the "seed". For example, the program:

```
10 J=RND(.5)
2Ø FOR J=1 TO 1Ø
3Ø PRINT RND(Ø)
4Ø NEXT
```

will set the seed to .5 and then print 1Ø pseudo-random values.

FORMATTED OUTPUT

If no format string is present in a PRINT statement, then all numeric values will be printed in the "default format". (The default format is initially set to be free format.) A format string may appear anywhere in the print list and must begin with a per cent (%) character, e.g.

```
PRINT %$10F2,J
```

A format string consists of a per cent character (%) followed by any number of format characters followed by a format specification. The format characters are:

C place commas to the left of decimal point as needed
$ put a dollar sign to the left of value
Z suppress trailing zeroes
make this format string the default format

Format specifications (similar to FORTRAN) are:

nFm F-format. The value will be printed in a n-character field, right justified, with m digits to the right of decimal point.

nI I-format. The value will be printed in a n-character field, right justified, if it is an integer. (Otherwise an error message will occur.)

nEm E-format. The value will be printed in scientific notation in a n-character field, right justified, with m digits to the right of the decimal point.

All printed values are rounded if necessary. A null format string specifies free format.

ACCESSING DATA FILES

All data files accessed by BASIC must have been created prior to use. Files to be accessed as BASIC data files must be of type 3. Also, if a file is greater than 256 blocks (64K bytes) then the area of the file past 64K is not accessible.

Both numeric and string data may be written on a file. BASIC buffers the file data in RAM in order to cause the minimum amount of disk activity, but the size of the buffer has no effect on the way that data may be written in the file.

OPEN #<file number>,<file name>

Before a file can be accessed, it must be "opened". The OPEN statement assigns a "file number" to the specified file. When actual accessing of the file is begun, the file number is used to identify the file. There are 4 legal file numbers: 0, 1, 2 and 3. The file name may be any string expression which evaluates to a legal file name (and optional disk unit specification) as described in the DOS manual. The OPEN statement also sets the "file pointer" to the beginning of the file (see below). For example:

```
OPEN #0,"ABC"
OPEN #3,A$
OPEN #J+1,A$+",2"
```

CLOSE #<file number>

The CLOSE statement serves two purposes. First, it prevents any further access to the file through the assigned file number. Secondly, the CLOSE statement forces the contents of the file buffer to be written to the file if necessary. It is very important to CLOSE a file as soon as possible after completion of writing to a file.

The READ and WRITE statements may be used to access a file in either a sequential or random access manner. Sequential access will be discussed first.

WRITE #<file number>,<list of items>

The WRITE statement will write a list of numeric and/or string values to the specified file. Writing will begin at the current value of the "file pointer". (Note that the file pointer is always set to the beginning of a file after an OPEN.) After the last item in the list has been written, a special "end-mark" character will be written in the file, and the file pointer will be set to address the end-mark.

Thus, a subsequent WRITE statement will cause the new list of items to overwrite the end-mark and be written following the previously written data, with the file pointer once again updated accordingly. For example:

```
WRITE #Ø,A
WRITE #3,A$
WRITE #J,"ABC",SIN(X),A$+B$
```

READ #<file number>,<list of variables>

The READ statement will read data items from the file into the specified variables. The items are read from the file starting at the current value of the file pointer. At the end of the READ statement, the file pointer is set to point immediately after the last item read. An error will be generated if an end-mark is encountered before the list of variables is exhausted, or if the type of data in the file does not match the type of variable into wwich it is being read. For example,

```
READ #Ø,A
READ #J,A,A$,B$,B
```

The TYP function will return a value which indicates the type of the item in the file addressed by the file pointer. The argument expression must evaluate to the file number of a currently open file. The TYP function will return:

Ø	if an end-mark is next
1	if a string is next
2	if a numeric value is next.

For example,

```
1Ø IF TYP(2)=Ø THEN STOP
```

The following table specifies exactly how many bytes are required to internally represent a floating point value as a function of the precision of your version of BASIC.

PRECISION	BYTES
2	2
4	3
6	4
8	5
1Ø	6
12	7
14	8

Thus, if the precison of your version of BASIC is 8, each numeric value will occupy exactly 5 bytes of file space. Character strings of any length may be written to a file. Strings with length less than or equal to 255 bytes (including Ø) require a

number of bytes equal to 2 plus the actual length of the string written. Strings with length greater than 255 characters require 3 plus the actual length. Note that when computing the total amount of data that will fit into a file, one byte must be reserved for the end-mark.

READ #<file number> %<file pointer>,<list of variables>
WRITE #<file number> %<file pointer>,<list of items>

Random READ and WRITE statements differ from sequential READ and WRITE statements in that they specify the file pointer value before performing the read or write. The file pointer is a byte address within the file. Thus BASIC programs may organize files into logical records of any size, or even into variable sized records. For example, the following statement will read three values from the Xth 37-byte record in file number 1.

```
10 READ #1 %37*X,A,B,C$
```

Normally, as described above, at the end of any WRITE statement an end-mark is written following the last item written. For some applications such as modifying an item within a record, or to save bytes within a record, it may be desirable to prevent writing the end-mark at the end of the WRITE statement. When the reserved word NOENDMARK is the last item in the WRITE statement item list, then no end-mark will be written.

```
20 WRITE #J %X ,SIN(J),NOENDMARK
```

Random file accessing is an advanced technique which provides a great deal of flexibility but also allows the possibility of errors. Erroneous calculation of file pointer addresses can result in unpredictable and catastrophic errors.

The process of creating a version of BASIC which accommodates your own terminal I/O conventions and memory size is described in the DOS manual.

In the event that you leave BASIC for some reason (e.g. to execute a DOS command, or because of a system interruption) BASIC may be re-entered at one of the following entry points. It is assumed here that you have a standard assembly of BASIC with origin at 2A00 hex.

2A00 This is the initialization entry point for BASIC. Entry at this address will result in a null BASIC program.

2A04 This is a continue entry point. Any program that existed at the time of interruption will still exist, but the value of all temporary variables will be destroyed. For example,

```
SAVE TEST1
FILE ERROR
BYE
*CR TEST1 10
*TY TEST1 2
*JP 2A04
READY
SAVE TEST1
READY
```

```
100 REM  A NUMERIC SORT PROGRAM
110 REM
120 DIM A(15)
130 PRINT "INPUT FIFTEEN VALUES, ONE VALUE PER LINE"
140 FOR J=1 TO 15
150 INPUT A(J)
160 NEXT
170 REM  DO EXCHANGE SORT UNTIL ALL IN ORDER
175 F=0 \ REM THIS FLAG USED TO SIGNAL WHETHER ARRAY IN ORDER YET
180 FOR J=2 TO 15
190 IF A(J-1)<=A(J) THEN 220
200 T=A(J)\ A(J)=A(J-1)\ A(J-1)=T\ REM EXCHANGE A(J) AND A(J-1)
210 F=1\ REM SET FLAG
220 NEXT
230 IF F=1 THEN 175\ REM  LOOP IF EXCHANGES HAPPENED
240 PRINT "SORTED ARRAY: ",
250 FOR J=1 TO 15\ PRINT A(J),\ NEXT
```

```
100 REM   CHARACTER SORT
110 REM   EXAMPLE USING STRINGS AND FUNCTION
115 DIM A$(72)
120 INPUT "TYPE A STRING OF CHARACTERS: ",A$
130 IF LEN(A$)=0 THEN 120
140 IF FNA(LEN(A$))=1 THEN 140\ REM CALL FNA UNTIL IT RETURNS ZERO VALUE
150 PRINT "SORTED ARRAY: ",A$
155 END
160 DEF FNA(N)\ REM CHARACTER SORT
170 REM  RETURN 0 IF A$ SORTED, ELSE RETURN 1
175 F=0
180 FOR J=2 TO N
190 IF A$(J-1,J-1)<=A$(J,J) THEN 220
200 T$=A$(J,J)\ A$(J,J)=A$(J-1,J-1)\ A$(J-1,J-1)=T$
210 F=1
220 NEXT
230 RETURN F
240 FNEND
```

```
100 REM INPUT A STRING AND CHECK THAT IT IS A LEGAL INTEGER
105 REM
110 DIM A$(72)
115 INPUT "TYPE AN INTEGER: ",A$
120 IF LEN(A$)=0 OR LEN(A$)>8 THEN GOTO 500
130 FOR J=1 TO LEN(A$)
140 IF A$(J,J)<"0" THEN 500
145 IF A$(J,J)>"9" THEN 500
150 NEXT J
155 PRINT "THE INTEGER IS OK: ",VAL(A$)
160 GOTO 115
500 REM
510 PRINT "NOT AN INTEGER!"
520 GOTO 115
```

```
100 REM
110 REM  PRINT A SINE WAVE VERTICALLY ON THE PAGE
115 FOR J=1 TO 100 STEP .1
120 T=SIN(J)
130 S=INT(30*T)
140 PRINT TAB(30+S),"*"
150 NEXT
160 STOP
```

```
100 REM
110 REM  PRINT A TABLE OF FORMATTED VALUES
120 REM
130 FOR J=1 TO 100
140 PRINT %3I,J,
150 PRINT %6F3,SIN(J),%7F4,COS(J),
160 PRINT %10E3,EXP(J),
170 PRINT %12F10,RND(0)
180 NEXT
```

```
100 REM PRINT THE CONTENTS OF AN UNKNOWN SEQUENTIAL FILE
120 DIM A$(1000)
140 INPUT "FILE NAME: ",B$
160 OPEN #0,B$
180 IF TYP(0)=0 THEN END
200 IF TYP(0)=1 THEN 300
220 READ #0,N
240 PRINT N
260 GOTO 180
300 READ #0,A$
320 PRINT A$
340 GOTO 180
```

```
100 REM   CONSTRUCT A FILE CONTAINING NUMERIC SQUARES,
110 REM   AND THEN USE RANDOM ACCESS TO COMPUTE SQUARES
120 REM   OF TYPED INPUT VALUES
130 OPEN #0,"TESTFILE"
140 FOR J=0 TO 500
150 WRITE #0,J↑2
160 NEXT
170 INPUT "X=",X
180 IF X<0 OR X>500 THEN END
190 READ #0%5*X,X2\ REM  EACH FP VALUE USES 5 BYTES
200 PRINT "X SQUARED: ",X2
220 GOTO 170
```

Appendix 2: The Line Editor

This section describes an advanced method of editting BASIC program lines. For the purpose of this discussion, the term "old line" will refer to a line of the BASIC program which is to be editted. The EDIT command may be used to designate which program line is to become the old line, e.g.,

```
EDIT 100
```

Otherwise, the most recently typed-in program line is automatically designated the old line.

When typing in a "new line" to BASIC, the old line may be used as a "template" for creating the new line. Special editting characters may be typed to cause parts of the old line to be used in contructing the new line. Use of the line editor will result in dramatically more convenient program modification.

When a new line is entered with the assistance of the line editor, it is as if the new line was entered in the "normal" manner. Thus a new line can have either the same or a different line number as the old line. For example, if the new-line line number is the same as the old line, then the new line will replace the old line in the program.

A line edit is terminated as soon as a carriage return is typed. If an illegal editting command is attempted, the line editor will ring a bell and perform no action.

During the entering of a new line, there are two "invisible pointers" maintained, one which locates a character position in the old line, and one which locates a character position in the new line. When beginning to enter a new line to BASIC, both pointers locate the first character position. Typing a normal (non-editting) character will advance both pointers by one position. The editting character control-G will cause the characters in the old line to the right of the old line pointer to be printed as part of the new line. Thus, for example, if the line

```
100 PRINT "**************
```

was entered as part of a program, and the missing end quote mark was discovered, the error could be corrected by typing EDIT 100, then control-G, then a quote mark, and then a carriage return. If it was desired to make line 234 do the same PRINT statement, then the following could be typed to accomplish this: 234 followed by control-G followed by a carriage return. It is suggested that you try these and similar examples before reading on. Now follows descriptions of each of the editting control characters.

Control-G Copy rest of old line to end of new line.
This command will print the characters from the current pointer position in the old line as part of the new line.

Control-A Copy one character from old line.
This command will print one character from the current pointer position of the old line as part of the new line. Both pointers will be advanced one character position.

Control-Q Back up one character.
This command will erase the last character of the new line, and also back up the old line and new line pointers. A left-arrow (under-line on some terminals) will be typed to indicate that this command was typed.

Control-Z Erase one character from old line.
This command advances the old line pointer by one character position. This command would be used to remove undesired characters from the old line. A per cent character (%) is printed to indicate that this command was typed.

Control-D Copy up to specified character.
This command requires a second character to be typed before it is executed. The command will copy the contents of the old line up to, but not including the first occurence of the specified character to the new line.

Control-Y Toggle insert mode.
When entering a new line, insert mode is "off". When insert mode is off, then typing normal characters will advance the old line pointer. When insert mode is "on", then typing normal characters will not advance the old line pointer. Thus, insert mode may be used to add some omitted characters from the old line. A left angle bracket will be typed to indicate entering insert mode, and a right angle bracket (>) will be typed to indicate leaving insert mode.

Control-N Re-edit new line.
This command erases the current new line and permits re-entering the new line. The partially complete new line becomes the old line for subsequent editting. (Of course, the EDIT command could be typed to select a different old line.) An at-sign (@) is typed to indicate that this command was typed.