FBASIC USERS MANUAL

TABLE OF CONTENTS

Overview 2
How to write and compile a program 3
Running a compiled program 5
Expressions 6
Functions
    ASC 6
    INT 7
    PEEK 7
Statements
    DIM 8
    DISK! 9
    END 9
    FOR - NEXT 10
    GOSUB 11
    IF - THEN 12
    POKE 12
    PRINT 13
    RETURN 14
    WHILE 14
Direct register usage 16
Reserved words 17
FBASIC versus OSI BASIC 18
Memory usage 19

OVERVIEW

The Pegasus Software FBASIC compiler is a complete language system designed to produce fast efficient machine code.

FBASIC accepts a special subset of the Microsoft version of BASIC, especially well suited to systems level programming. Many extensions and slight variations have been included to take better advantage of the machine facilities available, and to afford the user greater flexibility.

Pegasus Software will offer continuing support for FBASIC. Updates with additional features will be made available to registered FBASIC License holders as they are developed.

FBASIC provides a tool that programmers well versed in BASIC can use to produce the software they need without having to resort to assembly language.

HOW TO WRITE AND COMPILE AN FBASIC PROGRAM

First create your program using the Microsoft BASIC editor supplied with the OSI system. If you do not use any of FBASIC's special features, you can test run your program with Microsoft BASIC.

After creating your program save it to a disk file. Exit BASIC and invoke the compiler with the command:

```
XQ FBASIC
```

The disk operating system will then load and run FBASIC. The following message should be displayed at the console:

```
FBASIC Compiler  Version 1.0

Copyright 1980 by Pegasus Software
```

Following this message the compiler will prompt for a starting address:

```
Start Address:
```

This refers to the memory address which you want the object code to begin at. A four digit hex address can be specified, or just press the RETURN key, and the compiler will default to $317E as the start address, which makes the object program compatible with the operating systems XQ command.

The compiler will then prompt for the name of the source file to be compiled:

```
Source file:
```

Enter the file name of your FBASIC program. The compiler will search the disk directory for the requested file and obtain the necessary track limits. If the file is not found, an error will be reported and control will be returned to the operating system.

Finally the compiler will prompt for an object file:

```
Object file:
```

Any valid file name will be accepted. If no object file is desired simply press the RETURN key.

The system will now begin translating the source program into machine code. Any errors encountered will be reported to the console.

After compilation the system will report the size of the object file produced and the number of variables used. Control will then be returned to the operating system. The compiler may be re-entered for additional compilations with the command:

```
RE B
```

A standard BASIC program may now be entered. At any point where the object module is to be called the USR function may be used, or just include the following BASIC statement:

```
DISK!"GO 317E"
```

This will cause the compiled program to be executed. Control will then be returned to the calling program when an END statement is encountered. A hybrid program created in this way can be stored on disk with the PUT command, and then executed as necessary with the usual:

```
RUN "filename"
```

As an example of this facility see the DIR program included with the FBASIC system. It may be run from BASIC in the normal fashion, or from the operating system with the XQ command.

RUNNING A COMPILED PROGRAM

After successfully compiling a program with a start address of $317E, it may be invoked from the operating system command loop by typing:

```
XQ filename
```

For programs starting at a different address, first read the file into $317E with the operating systems LOAD command, then move it into position with the OS-65D extended monitor.

All object files produced by FBASIC contain the standard five byte header necessary for their use with the operating systems LOAD command. Therefore as an alternative to using the extended monitor, an object file may be CALLed into memory if allowance is made for the file header. As an example if a program was compiled for location $8000, it should then be CALLed into $7FFB ($8000-5).

```
CA 7FFB=50,1
GO 8000
```

If the program is more than one track long then the address for CALLing subsequent tracks must be computed by adding $B00 (2816) to the previous address. If the object file in the previous example were 2 tracks long then the following sequence would be appropriate:

```
CA 7FFB=50,1
CA 8AFB=51,1
GO 8000
```

This sequence can be executed from virtually any program which allows these operating system calls to be made, including XPL0, BASIC, the Extended Monitor, and the Assembler.

The length of a file in tracks is reported at the end of a compilation, and is also stored in the object file as the fifth byte in the header.

Object files produced by the compiler are specifically designed to be used along with the BASIC interpreter. This allows the user to take better advantage of each systems capabilities.

To utilize this feature, a program is first compiled in the normal fashion. The BASIC interpreter is then invoked, and the compiled program loaded in the same manner as a standard BASIC source file.

At this point it will appear that no program is present, the LIST command will show nothing. The compiled program is resident at $317E, and has adjusted BASICs work-space pointers above itself.

Expressions

An expression, in both BASIC and FBASIC, is a list of one or more arguments coupled with arithmetic and/or logical operators which evaluate to a numeric value.

FBASIC accepts the following operators, listed in order of precedence, from highest to lowest:

```
(   )
Unary minus
*   /
+   -
=   <>  <   >   <=  >=
NOT
AND
OR
```

This precedence pertains to the order in which the various operations are evaluated in an expression containing more than one operator. If an expression is composed of operations of equal precedence then evaluation occurs from left to right. The relational operations evaluate to one of two values; 0 (false), or 65535 (true). As an example, the expression:

```
10<>5
```

will return the value true, because 10 is not equal to 5.

Logical operations accept 16 bit arguments and produce a 16 bit result. This provides a greater degree of control than is available with OSI BASIC which is limited to 15 bit arguments.

No run-time error checking is done for overflow, underflow, or division by zero. This contributes greatly to the speed of programs produced by the compiler.

FUNCTIONS:

A function is a special operator which requires an argument, and returns a numeric value. FBASIC functions may be used freely within numeric expressions, as operands.

The ASC function:

The ASC function returns the ASCII value of the first letter of a string constant. Therefore in the following example:

```
PRINT ASC("A")
```

the number 65 will be printed as that is the ASCII equivalent for the letter A. The ASC function can be used in this way to specify any printing character.

The INT function:

The INT function is provided to facilitate greater compatibility between BASIC and FBASIC. In BASIC the INT function returns the integer part of the enclosed expression. Since FBASIC is integer only, to insure that a program will run with both BASICs the INT function is used to force the same results.

The division operation is the most troublesome to compatibility for both BASICs since integer division discards any remainer that may result. Therefore by forcing all divisions to integer results, a major incompatibility is alleviated. This allows greater freedom to develope programs in the interactive environment of BASIC before compiling them.

Since all numeric operations in FBASIC are integer operations, the INT function does not actually cause any code to be produced.

The PEEK function:

The PEEK function allows inspection of any byte of memory. It requires one argument. The value of the argument is used to designate a specific memory location. The value stored at that location is then returned.

PEEK may be used within an expression, or in place of an expression. Some examples are:

```
PRINT PEEK(200)
B=PEEK(AD)+PEEK(AD+1)*256
```

The second example shows a convenient way of retrieving the value of a 16 bit address as stored by the 6502 processor, with the low byte first.

The PEEK function always returns a value within the range 0 to 255.

The DIM statement:

The DIM statement is a declaration statement that reserves space for integer arrays or vectors.

The statement:

```
DIM A(500)
```

allocates memory for 501 integers, (one more than is specified because the index starts with zero). The DIM statement requires that the argument be a numeric constant as no facilities are provided for array allocation at runtime.

To allow more efficient memory usage, and greater flexibility, the FBASIC DIM statement may be used to specify an address rather than to reserve space. This allows the programmer to place data anywhere he may have additional memory. The syntax for this option is as follows:

```
DIM B(!57344)
```

The exclaimation point preceding the number in this example instructs the compiler to use the constants value as the address for all subsequent accesses of the declared array. In this example the first element of the array will be stored in memory at location 57344. The same location could be specified in hexidecimal:

```
DIM B(!$E000)
```

No boundary checking is done at runtime, so it is possible to access a value outside the limits of a dimensioned array. This should be avoided as other parts of the user program or the operating system may be overwritten.

The FBASIC DIM statement also allows the values of an array to be initialized at compile time. Here is an example:

```
DIM A(10)=(1,2,3,4,5)
```

The first place within this array, location A(0), will be set equal to 1. Similarly, A(1) would equal 2 and A(4) would equal 5. Additional locations not provided with initial values are set to 0. Since this initialization is accomplished at compile time, use of this feature with an absolute memory location is invalid.

```
DIM B(!32768)=(5,5,5,5) : REM invalid
```

For larger tables of data this initialization may be continued on as many additional lines as necessary. The compiler will continue looking for parameters until it encounters a right parenthesis.

```
100 DIM C(100)=(1,2,3,4,5,6,7
110 ,           8,9,10,11,12,13
120 ,          14,15,16,17,18
130 ,          19,20,21,22,23)
```

Notice that the separating comma between the last parameter of one line and the first parameter of the following line is placed at the beginning of the line. This is necessary because BASIC's editor removes blanks from the beginning of each line, and would therefore concatenate the line number with the first parameter. The blank space in the second, third, and fourth lines of this example is for clarity, and is not required.

The initializing data must be numeric constants only.

The DISK! statement:

The DISK! statement provides access to the operating system commands from an FBASIC program. It accepts a string constant as the command specifier. Any command acceptable to the OS-65D operating system may be used.

```
DISK! "CALL 4000=08,1"
```

The END statement:

The END statement provides a means for completing or halting the run of a program.

The END statement may be used at any point to abort a programs flow and return control to the calling program, usually the operating system.

An END is automatically appended to the end of each FBASIC program.

The FOR - NEXT loop:

The FBASIC FOR-NEXT loop is very similar to the OSI FOR-NEXT loop. The FOR and NEXT statements are used together to provide a convenient means of loop control. The FOR statement is the loop initiator and therefore contains the limit parameters of the loop. In the example:

```
10 FOR I=1 TO 100
20 PRINT I
30 NEXT I
```

the variable I is used as the loop counter. Therefore the sub-statement "I=1" is actually an assignment which sets up the initial value of the loop counter I. The keyword "TO" delimits the initialization and limit expressions. The second expression, (in this case "100"), specifies the termination condition of the loop. In other words, the loop will be exited when the value of the counter variable I is greater than 100.

The counter variable is incremented by one each time through the loop until it is greater than the exit parameter.

The NEXT statement provides the delimiter for the "bottom" of the loop. Therefore, the loop is a group of statements with FOR at the beginning and NEXT at the end.

The I following the NEXT in this example is optional. Since each NEXT statement is always associated with the last FOR, the compiler makes the association automatically. Each FOR statement must have exactly one NEXT associated with it.

If two or more NEXT statements occur together, as when loops are nested, the following shorthand may be used in place of separate NEXT statements back to back:

```
NEXT J,I
```

The exit test is effectively at the bottom of the loop, therefore a FOR - NEXT loop is always executed at least once.

FOR-NEXT loops may be exited prematurely if necessary, with a GOTO or similar statement.

The initialization and limit expressions of the FOR statement are evaluated only upon entering the loop. Therefore altering the values of the variables used within these expressions (other than the loop counter) from within the FOR-NEXT loop will have no effect on the number of times the loop is repeated.

FBASIC does not support the step specifier.

The GOSUB statement:

The GOSUB statement is used to transfer control to a specified line much like the GOTO statement. The difference is that it saves the address of the statement immediately following the GOSUB, and upon encountering a RETURN statement control is then returned to that saved address.

The GOSUB statement requires one argument, the line number to be called or transfered to.

```
GOSUB 2000
```

To allow convenient interface to other machine language routines, the compiler also accepts an absolute address in place of the line number. This is signalled by placing an exclamation point in front of the address.

```
GOSUB !$FD00
```

or

```
GOSUB !64768
```

The compiler will generate the necessary code to transfer control to the machine code at location $FD00. When the processor encounters a Return from Subroutine (RTS), instruction control will be returned to the statement immediately following the GOSUB.

Since this statement uses the 6502 processor stack to save the return address, over 100 levels of subroutine calls may be used. This refers to the ability of a subroutine to call another subroutine. In contrast OSI BASIC allows a maximum of 26 levels. Although this increased capability allows a limited use of recursion, care should be execised so that the stacks limits are not exceeded as no runtime error checking is made on this condition.

The GOTO statement:

The GOTO statement allows the simplest form of control of program flow. It requires one argument, a line number.

The GOTO statement causes control to transfer to the specified line, and therefore allows the normal sequential execution of lines to be altered as required.

The IF statement:

The purpose of the IF statement is to provide control of program flow. The IF statement provides a means for conditional execution of a group of statements. This statement is comprised of a conditional expression and a statement or group of statements to be conditionally executed. The THEN keyword separates these two parts, and the end of line is used to mark the end of the group of statements.

Execution of an IF statement begins with the evaluation of the conditional expression. The associated statement or group of statements is then executed or skipped depending on the value of the expression.

The conditional expression is identical to a standard expression except that it is interpreted as having only two possible values, "true" (any non-zero value) or "false" (zero).

In the example:

```
IF A=5 THEN PRINT "YES"
```

the conditional expression "A=5" is evaluated first. If the variable A is equal to 5, the condition is true and control passes to the statement immediately following the THEN keyword. Therefore if A equals 5 then YES will be printed. If A is not equal to 5, the PRINT statement is skipped and control passes to the next line in the program.

The use of the GOTO statement is quite common with IF - THEN. For this reason a shortened version of this construct is allowed. The following two statements are functionally equivalent.

```
IF A=1 THEN GOTO 500
IF A=1 THEN 500
```

IF statements may be nested:

```
IF A=5 THEN IF J=12 THEN PRINT "You Bet!"
```

The POKE statement:

The POKE statement provides a means for altering any memory location. It requires two arguments. In the example:

```
POKE 57088,2
```

the value specified by the second argument, (in this case 2) is stored at the memory location specified by the first argument, (57088). Any valid expression may be used for either argument.

The address is a 16 bit value, allowing access to all 64K addressable memory locations. The second value is stored into an 8 bit memory location. If this parameter is greater than 255 (8 bits), then the high order bits are simply ignored. In the example:

```
POKE 57088,256
```

the value stored at 57088 would be 0. Therefore the second parameter is effectively ANDed with 255.

The PRINT statement:

The PRINT statement provides program ouput via the OS-65D operating system. Output goes through the OS-65D input/output distributor, and may be redirected to the various system devices as outlined in the OS-65D manual. FBASIC does not support the PRINT# statement. Therefore redirection of output is accomplished through direct access to the input/output distributor flags. Examples:

```
POKE 8994,2   : REM output to video
POKE 8994,1   : REM output to serial
POKE 8994,3   : REM output to both
```

This form of I/O redirection is more versatile than the PRINT#, because it allows output to be directed to several devices simultaneously.

The PRINT statement accepts a list of zero or more arguments, it supports:

String constants
numeric expressions
The CHR$() function

A string constant is a group of characters enclosed in quotes:

```
"I AM A STRING"
```

Numeric expressions are covered in detail within the section devoted to that topic.

The character string function "CHR$(exp)" is used to output a single character corresponding to the ASCII equivalent of the enclosed expression.

PRINT statements may contain any combination of these three types of arguments. FBASIC arguments should be separated by semicolons, not commas.

A carriage return/line feed is normally printed automatically at the end of a print statement. This may be suppressed by placing a semicolon after the last argument in the statement.

When FBASIC prints the value of a numeric expression no spaces are printed either before or after the number itself. This simplifies formatting of numeric output.

The RETURN statement:

The RETURN statement causes control to return to the statement immediately following the most recently executed GOSUB.

If a RETURN is encountered without a preceeding GOSUB, control is returned to the calling program, usually the operating system or BASIC.

The WHILE statement:

The WHILE statement is used to control program flow. It provides a mean to combine a group of statements into a single unit, which is conditionally executed, depending on the value of a conditional expression.

The WHILE statement is comprised of two parts, the conditional expression, and a group of statements. In the example:

```
WHILE A<>5
PRINT A
A=A+1
WEND
```

the expression immediately following the WHILE keyword is evaluated first. If the condition is true, control passes to the statements immediately following. When the WEND statement is encountered it passes control back to the WHILE statement thus forming a loop. If the WHILE condition is false, control passes to the statement immediately following the WEND keyword.

If the conditional expression is false initially then the enclosed group of statements will be skipped entirely.

Each WHILE statement must have exactly one WEND associated with it. This association is similar to that of the FOR-NEXT statement. Each WEND is automatically associated with the closest preceding WHILE. WHILE statements may be nested.

DIRECT REGISTER USAGE

FBASIC allows direct access to the A, X, and Y registers of the 6502 processor. This enables programs to be directly interfaced to existing machine language routines.

Register access is specified by a dot or period, followed by the letter corresponding to the desired register. In the assignment statement:

```
.A=H
```

the accumulator will be loaded with the low-order byte of the variable H.

As a further example, the OS-65D operating system contains a useful subroutine which when called prints the value of the accumulator to the console in hexadecimal. Therefore the following program will print the hexadecimal value of H to the console:

```
.A=H/256      : REM the high byte first
GOSUB !$2D92 : REM call hex print
.A=H          : REM the low byte
GOSUB !$2D92 : REM and print it
```

Since the 6502 registers are 8 bits wide the high order byte of the expression is ignored.

In this form of assignment statement any valid numeric expression is acceptable. However, be careful not to place any statements between the assignment and the GOSUB that might corrupt the register value. The only statements that do not alter the registers are; GOSUB, GOTO, and RETURN.

Assignment of the other registers may be made without affecting previous register assignments if the assigned expression is limited to a single variable or constant. For instance to call a subroutine with the A and Y registers holding the low and high bytes of a variable use the following format:

```
.Y=H/256          : REM get high byte of H
.A=H              : REM and low byte
GOSUB 1000        : REM call subroutine
```

In this case the two assignments MUST be made in the order shown, otherwise the computation involved in the first statement may corrupt at least one of the register values.

In any register assignment involving an expression with one or more arithmetic or logical operators, only the register involved is concidered of known value at the completion of the assignment. The other registers are often used for the evaluation of the expression.

The compiler also provides a means for placing a register value into a variable. This is necessary for the utilization of some existing subroutines. As an example, the operating systems input routines return inputed characters in the A register. In the following example the main OS-65D input routine is called and the returned character placed in the variable CH.

```
GOSUB !$2340     : REM input with echo
CH=.A            : REM save accumulator
CH=CH AND 255    : REM zero high byte
```

In order to preserve the three processor registers this type of assignment statement does not affect the high byte of the variable. Therefore the variable should then be subsequently ANDed with 255 in order to insure that the high byte is zero, or the variable may be zeroed prior to calling the subroutine.

RESERVED WORDS

AND	GOSUB	OR	THEN
ASC	GOTO	PEEK	TO
CHR$	IF	POKE	WEND
DIM	INT	PRINT	WHILE
END	NEXT	REM	
FOR	NOT	RETURN	
Input	SPC( )		
RND( )	POS( )		
MOD	TAB( )		

FBASIC versus OSI BASIC

The purpose of this section is to point out some of the differences between FBASIC and the OSI interpreter BASIC.

NUMBER SYSTEMS

The most fundamental difference between these two BASICs is their number systems. While OSI BASIC supports both integer and floating point numbers, FBASIC is limited to integers at this time. Also the representation of integers differs between the two systems.

OSI BASIC integers are 15 bits wide, with an additional bit for the sign. They cover the range from -32768 to 32767.

In contrast FBASICs integer representation is unsigned with a range of from 0 to 65535.

Since the 6502 processor does not directly support signed integers the unsigned approach supported by FBASIC affords a much greater level of efficiency and speed to compiled programs.

With an understanding of the differences between these two types of integer representation the transition should be fairly simple. The most important aspect to remember is the 0 boundary. In the example:

```
IF A<0 THEN 200
```

the condition A<0 while valid and useful in OSI BASIC, can never be true in FBASIC because negative numbers are not included in it's number system.

With OSI BASIC when the integer range is exceeded in either direction an error is reported. Conversely, with FBASIC the nmply wraps around. Therefore the expression:

```
65535+1
```

is equivalent to 0 in FBASIC. When fully understood this feature can be used to great advantage.

OTHER DIFFERENCES

The following statements, commands and functions are not supported by FBASIC at this time:

ABS	FRE	POS	STEP
ATN	~~INPUT~~	READ	STOP
CLEAR	LEFT$	RESTORE	STR$
CONT	LEN	RIGHT$	~~TAB~~
COS	LIST	~~RND~~	TAN
DATA	LOG	RUN	~~TO~~
DEF	MID$	SGN	USR
EXIT	NEW	SIN	VAL
EXP	NULL	~~SPC~~	WAIT
FN	ON	SQR	

MEMORY USAGE

This section deals with the allocation and use of memory by a compiled program.

All non-subscripted variables are stored within the base or zero page of memory. The maximum number of these variables is limited to 100. Allocation starts at location one (not zero), and goes upward.

Thus the amount of the base-page used by a program may be ascertained from the number of integer variables used. As an example, if the compiler reports that your program contains 10 variables (not including array variables), then the first free space in the base-page is at location 21 ($15). That is:

No. variables * 2 + 1

The number of variables is multiplied by 2 because variables require 2 bytes of storage each.

The operating system and the FBASIC runtime package use various locations in the base-page above $E0. Therefore locations from $E0 to $FF should not be used in your programs.

FBASIC does not initialize the processors stack pointer. This allows a compiled program to be called from virtually any other program as a subroutine, and to return to that calling program when its function is complete.

Dimensioned arrays are allocated space within the object module. This allows their values to be pre-initialized if desired.