FBASIC V1.1

Pegasus Software
P.O. Box 10014
Honolulu, Hawaii 96816

TABLE OF CONTENTS

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 software that would otherwise require assembly language.

A working knowledge of OSI/Microsoft BASIC is recommended in order to take full advantage of this manual.

HOW TO WRITE AND COMPILE AN FBASIC PROGRAM

FBASIC has been designed to be used with the OSI interpreter during development of programs in order to simplify the transition from interpreter to compiler, and to allow limited testing in the interpretive mode.

You first create your program source using the Microsoft BASIC editor supplied with your OSI system. If you do not use any of FBASIC's special features, and work carefully within the integer subset you can test-run your program with Microsoft BASIC before compiling it.

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

```
A*XQ FBASIC
```

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

```
FBASIC Compiler  Version x.x
Copyright 1980,81 by Pegasus Software
```

Following this the compiler will print a greater than symbol (>) and pause for input. The command line interpreter is similar in syntax to that used with many of the utilities available under the Bell Labs UNIX system.

Minimally a source file name must be specified. The input

```
>PHRED
```

will cause the system to compile the source file PHRED and list any errors to the console. No object code will be produced. To enable the various compile-time options they must be flagged on the same input line.

These options are signalled by a dash or minus sign (-) followed by a letter. To specify an object file which the compiler can use to store the machine code it produces; the -O option should be used. Therefore

```
>PHRED -O TEST
```

instructs the compiler to read source from PHRED and write the resulting object code to TEST. With the -O option, the compiler assumes the file name is to follow. The order in which input parameters are entered is otherwise unimportant.

```
>-O TEST PHRED
```

is equivalent to the previous example. The system will list the available options if a single dash (-) is entered.

A disk drive may be specified along with each file name if necessary. In this example

```
>B:PHRED -O A:TEST
```

the file named PHRED is to be found on drive B and TEST is to be found on drive A. If the compiler was invoked from the A drive then the A specifier for TEST is not necessary.

COMPILE-TIME OPTIONS

FBASIC supports a list of options which may be enabled as desired at compile-time. These options are specified on the same input line as the file names to be accessed for the compilation. Each option is designated by a dash or minus sign followed by a letter. Note that a source file is always required to affect a compilation. In the examples that follow SOURCE will be used as the source file name.

Also note that the various options may be entered at the command line in any order, blanks are simply ignored, and all input is converted to upper-case.

-O

Used to specify a file for the object code produced by the compiler. It instructs the compiler to use the name that follows as the output file name.

```
>SOURCE -O OBJECT
```

-U

This option causes the compiler to generate code compatible with the OS-65U operating system. It adjusts input and output calls to suite that systems requirements. It also causes the zero page and stack memory to be swapped in and out on entering and returning from the compiled code so that it may function as an adjunct to BASIC. And it causes the compiler to report DISK! statements as syntax errors since they are not applicable with the 65U system.

```
>SOURCE -O OBJECT -U
```

With this option enabled the compiler sets its default starting code address to $6000. This allows the code to be merged with a 65U BASIC file.

-L

This option causes the compiler to list the line numbers of the source program as they are encountered, along with the memory address of the object code produced for each line. This is useful for calculating the amount of code produced for each line and for calling compiled subroutines from other programs. A line address so reported should not be considered a valid entry point unless it is referenced by a GOTO or similar statement within the same program.

```
>SOURCE -L
```

-S

This option causes the compiler to sort its symbol table and list it to the console at the end of a compilation. It lists all the non-subscripted variables used along with their absolute memory addresses.

```
>SOURCE -S
```

-C

This option allows the starting code address to be set to something other than the default ($317E for 65D, $6000 for 65U). The compiler expects to find the address in hexidecimal following the -C.

```
>SOURCE -O OBJECT -C E000
```

-H

This option causes the OS-65D five byte header to be omitted from the output file. Since the header is only necessary for the LOAD, XQ, and RUN commands its omision can simplify other techniques for loading compiled modules. As an example, if the code is compiled for address $E000 (using the -C option) to call it into memory at that address the statement would be something like this:

```
DISK!"CA DFF8=25,1"
```

The address has been adjusted for the five byte header ($E000-5). By using the -H option at compile-time this adjustment is unnecessary. Therefore if the program were compiled like this:

```
>SOURCE -O OBJECT -C E000 -H
```

then to call the code in to memory at $E000 would require something like this:

```
DISK!"CA E000=25,1"
```

-W

This option causes the compiler to pause or Wait when an error is encountered. This is useful to keep from missing error messages as the compiler proceeds with a compilation.

After pausing at an error the compiler can be instructed to continue on by pressing RETURN, or the compilation may be aborted by pressing Control-C.

-E

Causes an exit to the operating system without compilation.

RUNNING A COMPILED PROGRAM

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

```
A*XQ filename
```

Object files produced by the compiler can be used along with the BASIC interpreter. This allows you 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 caused BASIC to adjust its work-space pointers above itself via the file header.

A standard BASIC program may now be typed in, or loaded from the indirect file. 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.

NOTE: A compiled program does not require the OSI BASIC interpreter to be resident at runtime.

VARIABLES

Variable names are of the same form as those accepted by the OSI interpreter. That is, an upper-case letter followed by zero or more upper-case letters or digits. Only the first 2 characters are significant. Therefore the variable names:

NA NAME NAK

are referenced as the same variable by the compiler. A percent sign following this alpha-numeric string (variable name) as used to flag integer variables for the interpreter, is acceptable. Other valid variable names are:

A A1 AB TRUE FALSE%

NOTE: Variable names must not contain reserved words; especially troublesome (easy to overlook) are TO, OR, AND, LET, LEN, and END.

EXPRESSIONS

An expression, 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
*   /   MOD
+   -
=   <>  <   >   <=  >=
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; Ø (false), or 65535 (true). As an example, the expression:

10<>5

will return the value true, because 10 is not equal to 5. In conditional statements such as IF - THEN any non-zero value is regarded as true.

Conditionals may be mixed with arithmetic operators. As an example the following are two ways of setting a "flag" variable according to a certain condition:

```
FLAG=0 : IF C>=5 AND B<>22 THEN FLAG=1

FLAG = C>=5 AND B<>22
```

The flag can then be tested elsewhere with a statement like this:

```
IF FLAG THEN PRINT "YES"
```

These examples are compatible with the Microsoft BASIC as well.

The MOD operator is used to return the remainder of the division of it's two arguments. In the example:

```
A = 5 MOD 3
```

A is assigned the value 2 because 5 divided by 3 leaves a remainder of 2. MOD is not supported by OSI BASIC.

Logical operations accept 16 bit arguments and produce a 16 bit result. OSI BASIC 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, but also places a greater burden on the programmer to insure that these conditions do not arise or cause erroneous results.

The ASC() function:

The ASC function returns the ASCII value of the first letter of the enclosed 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. Character constants may also be specified by placing them between single quotes. The next two examples are functionally equivalent:

```
IF C>=ASC("A") AND C<=ASC("Z") THEN 2000
IF C>='A' AND C<='Z' THEN 2000
```

The second example is not compatible with OSI BASIC.

INT

The INT function is supported to ease the transition from OSI BASIC to FBASIC. In BASIC the INT function returns the integer part of the enclosed expression. Since FBASIC is integer only, INT is actually ignored. Therefore programs which are to be tested with OSI BASIC can use the INT function where necessary to keep results the same.

Division is the most troublesome in this respect. By enclosing all divisions within INT functions the main point of incompatibility bewteen the two BASICs can be alleviated. This allows greater freedom to develope programs in the interactive environment of OSI BASIC before compiling them.

PEEK

PEEK allows the inspection of any byte of memory. It requires one argument, which is used to designate the address of the desired memory location. The value stored at that location is then returned.

RND

The RND() function returns a pseudo-random number within a range controlled by the enclosed expression. In the example:

```
A=RND(10)
```

A will be set to a number in the range from 0 to 9.

FLOW OF CONTROL

GOTO

The GOTO statement allows the simplest form of control of program flow. It causes control to transfer to a specified location allowing the normal sequential execution of lines to be altered as required. The target to transfer control to is specified by a decimal constant corresponding to a line in the program (GOTO 1ØØ) or by a hex or decimal constant preceded by an exclamation point (!); which denotes an absolute memory address (GOTO !$2A51).

The compiler generates a single JMP instruction for GOTO.

GOSUB

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. As with GOTO the target address may be specified by linenumber (GOSUB 2ØØ) or by absolute address (GOSUB !$2D92).

Since this statement uses the 65Ø2 processor stack to save the return address, over 1ØØ levels of subroutine calls may be used. 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 compiler generates a single JSR instruction for GOSUB. This makes the use of common subroutines advantageous for a reduction in code size.

RETURN

The RETURN statement causes control to return to the statement immediately following the most recently executed GOSUB. If a RETURN is encountered without a preceding GOSUB, control is returned to the calling program, usually the operating system or BASIC.

A single RTS instruction is generated for the RETURN statement.

IF

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 5ØØ
IF A=1 THEN 5ØØ
```

IF statements may be nested:

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

ON GOTO/GOSUB

The ON keyword is used with either GOTO or GOSUB to create a multiway branch in the logical flow of a program. That is, it allows you to transfer control to any one of a list of locations within a program, with the location chosen according to the value of the expression.

```
10 INPUT "Enter option number (1-3)"; N
20 ON N GOTO 100,200,300
30 PRINT "Bad selection" : GOTO 10

100 REM Option 1

200 REM Option 2

300 REM Option 3
```

As in this example the list of locations follows the GOTO or GOSUB with commas in between as separators. Each number in the list is associated with a value according to its position in the list. The first member being 1, the second 2 and so on. Therefore if the variable N in line 20 is equal to 1 then control is transfered to line 100.

If the expression is equal to 0 or is greater than the number of references in the list then the statement is ignored and control passes to the statement immediately following the ON statement. In the example, if N equals 0 or N is greater than three then the ON statement does nothing, and control passes to line 30. This is called the default.

Up to 128 references may be made within an ON statement. To do this the list of references may be spread over several lines by using a plus-sign (+) as a continuation character.

```
10 ON X-20 GOTO 100,200,300,400,500+
20 ,600,700,800,900,1000,1010,1020+
30 ,1030,1040,1050,1060,1070,1080
```

Each continuation line must begin with a comma. The rest of the line following each plus sign is ignored by the compiler.

Using GOSUB with ON is slightly different than with GOTO because of its built-in return feature. When a RETURN statement is encountered after an ON GOSUB, control is returned to the statement immediately following the ON GOSUB statement. *That is, it returns to the same statement that takes control on default.*

The ON statement helps to consolidate multiple decisions and to reduce code size. When three or more references are made with the ON statement instead of using multiple IF statements a very noticeable decrease in the amount of code generated is usually apparent.

WHILE

The WHILE statement is used to combine a group of statements into a single unit which is executed zero or more times until the value of its expression is zero (false). 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 and NEXT keywords. Each WEND is automatically associated with the closest preceding WEND-less WHILE. WHILE statements may be nested to any level.

FOR

The FOR statement is used to form a program loop.
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 assigns I to an initial value of 1. The keyword TO delimits the initialization and limit expressions. The second expression (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 closest preceding NEXT-less 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.

END

The END statement provides a means for completing or halting the run of a program. It 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. A single JMP instruction to the runtime package is generated for END.

EXIT

The EXIT statement is used to exit a program and transfer control directly to the operating system. This is especially useful for programs like the XREF utility which can be called from BASIC, but should not return to it because they write over BASIC. The compiler generates a JMP instruction when an EXIT is encountered.

INPUT

The INPUT statement provides program input via the operating system. It can take either of the two forms illustated by the following examples:

```
INPUT A
INPUT "Prompt string";A
```

In the first example, the program takes input from the currently active device, and stores the associated numeric value of that input into the variable A. In the second example, the string of characters within quotes in the INPUT statement are printed prior to taking input. The cursor is placed at the space immediately following the string. Unlike OSI BASIC no additional characters are displayed for prompting. You have *full* control over the input prompt.

Error handling on numeric input is *not* automatic. No message ("REDO FROM START") is produced. Instead this condition is flagged for the programmer to handle as he pleases. On non-numeric input the processor's Y register is returned with the high bit set to one.

```
10 INPUT A
20 T=.Y      : REM see section on register access
30 IF T>128 THEN PRINT "IMPROPER INPUT" : GOTO 10
```

The input is buffered at $2E79 ($26F2 for 65U) and may be accessed with the PEEK function. This buffer is used by the operating system to buffer the directory when searching for a file, it is not normally used for any other purpose.

The length of the line of input is also returned in the Y register and may be used as follows:

```
INPUT A
LN=.Y
LN=LN AND 127 : REM ignore high bit (error bit)
```

Null input (carriage return only) is flagged as valid numeric and the variable is set to zero.

At runtime the INPUT statement allows simple line editing. Both the Rubout and Underline (shift-O on video systems) can be used to delete characters. Control-U may be used to kill or delete the whole line, instead of OSI's use of the "at" symbol (@).

Input comes through the operating system's input/output distributor. The input device may be selected by changing the operating system's input distributor flag (see your operating system manual).

PRINT

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

```
POKE 8993,1   : REM input from serial
POKE 8993,2   : REM input from polled keyboard
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. Input may come from only one device at a time.

The PRINT statement accepts a list of zero or more arguments including:

String constants
numeric expressions
The CHR$() function
The SPC() function
The TAB() 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. As an example:

```
PRINT CHR$(65)
```

will print the character A to the current output device, as 65 is the ASCII code for that letter.

PRINT statements may contain any combination of these three types of arguments. 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.

Some additional examples:

```
PRINT "The answer is:"; A
PRINT "25 plus 88 equals: "; 25+88
PRINT CHR$(PEEK(I));
```

PRINT Functions:

To simplify formatting of output FBASIC provides several functions which are associated with the PRINT statement.

POS

The POS() function is used to obtain the current column position of the cursor. That is, the location at which the next character will appear on the current output device. This eases formatting of output.

```
A=POS(Ø)

IF POS(Ø)>32 THEN PRINT
```

The POS() function requires a dummy argument the value of which has no effect on the value it returns.

SPC

The SPC() function is used to print a number of spaces. The enclosed expression governs the number of spaces to be printed.

```
PRINT A; SPC(C*2); B
```

TAB

TAB() is used to move the cursor to a specified column before printing.

```
PRINT TAB(1Ø); B1; TAB(2Ø); B2
```

If the cursor position is greater than the TAB argument then the TAB is ignored.

DISK!

The DISK! statement provides access to 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"
```

One FBASIC program can be called from another (chained) by using something like this

```
DISK! "XQ TEST"
```

DIM

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 argument must be a numeric constant as no facilities are provided for array allocation at runtime.

The DIM statement may also be used to specify an address rather than to reserve space. This allows you to place data anywhere in memory. The syntax for this option is as follows:

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

The exclamation 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 DIM statement also allows the values of an array to be initialized at compile time.

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

The first place within this array, location A(Ø), will be set equal to 1. Similarly, A(1) will equal 2 and A(4) will equal 5. Additional locations not provided with initial values are set to Ø. 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 wrong
```

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 closing right parenthesis.

```
1ØØ DIM C(1ØØ)=(1,2,3,4,5,6,7
11Ø ,           8,9,1Ø,11,12,13
12Ø ,          14,15,16,17,18
13Ø ,          19,2Ø,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 or character constants as in the next example:

```
DIM CH(9)=('#','$','%','&')
```

Array initialization in FBASIC is provided as an alternative to DATA statements. DATA statements lend themselves poorly to efficient generation of code.

POKE

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

```
POKE 57088,2
```

A 2 will be stored at location 57088 in memory. 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) the high order bits are simply ignored. In the example:

```
POKE 57088,256
```

the value stored at 57088 would be 0.

Note: The statement POKE exp1,PEEK(exp2) will not work with OSI BASIC but will work properly with FBASIC.

#FILE

The #FILE statement allows source files to be linked together. This removes the 36K source file size limitation imposed by OSI BASIC. The only limitations to the size of an FBASIC source program are the amount of disk space available, and the number of line numbers' acceptable to BASICs editor (64,000). This statement is used as follows:

```
#FILE "NEXTFILE"
```

As with the command line interpreter file names may be preceded by a disk drive specifier:

```
#FILE "B:NEXTFILE"
```

Linked files should not contain coinciding line numbers for obvious reasons. During pass one the compiler reports each #FILE statement encountered to the console.

DIRECT REGISTER ACCESS

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 are guaranteed not to alter the registers are; GOSUB, GOTO, RETURN and simple assignment statements such as

```
.X=200
.Y='A'
.A=VAR
```

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 considered 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 system's input routines return inputted 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 !$234Ø     : 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. If the variable is used only for 8 bit values (Ø to 255) then zeroing each time is unnecessary.

Register assignment may also be used to advantage in large programs for passing parameters to often called subroutines. This can increase speed while reducing program size. To simplify passing 16 bit parameters a form of register addressing is supported which accesses the A and X registers symultaneously:

```
.AX=NUM*25
VAL=.AX
```

The low order byte is placed in the A register and the high order in the X register.

One last form of register addressing allows access to the 65Ø2 stack pointer. It is useful when several levels of GOSUBs need to be exited at once, as was necessary with the compiler itself for error recovery.

```
1ØØ STACK=.S
11Ø REM
12Ø .S=STACK
13Ø GOSUB 5ØØ
14Ø :
5ØØ GOSUB 6ØØ
51Ø :
6ØØ IF ERR THEN 12Ø : REM cleanup stack & restart
61Ø RETURN
```

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 $D0. Therefore locations from $D0 to $FF should not be used in your programs.

FBASIC does not initialize the processor's 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 not employing the absolute address feature are allocated space within the object module. This allows their values to be pre-initialized if desired.

RESERVED WORDS

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

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

A 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. 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 Ø to 65535.

Since the 65Ø2 processor does not directly support signed integers the unsigned approach 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 Ø boundary. In the example:

```
IF A<Ø THEN 2ØØ
```

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

With OSI BASIC when the integer range is exceeded in either direction an error is reported. With FBASIC the number simply wraps around. Therefore the expression:

```
65535+1
```

is equivalent to Ø 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:

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

Further, FBASIC zeros only the non-subscripted variables at runtime upon entering a program. OSI BASIC zeros all variables at that time.

UTILITIES

Included with the compiler are several utility programs written in FBASIC. The source code is included with most of these to provide examples of the various FBASIC constructs.

The FBASIC system diskette comes with the following files, (as listed with the DIR utility):

```
BEXEC* 09-09   DELETE 11-11   FBLIB  30-30   PASS2  13-23
CREAT$ 29-29   DIR    12-12   FDUMP  10-10   RENUM  31-31
CREATE 33-33   DIR$   24-24   FDUMP$ 37-37   RENUM$ 34-36
DEL$   32-32   FBASIC 25-28   OS65D3 00-08   XREF   38-41

 16 Files Defined
```

CREAT$

This file contains the source code for the CREATE utility.

CREATE

The CREATE utility is similar to the one supplied with the OS-65D operating system. A sample dialogue:

```
RUN"CREATE"
File name:FRED

42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58
61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76

First track:42
Number of tracks:3
```

The numbers listed after the file name is entered are the available tracks on the current disk.

DEL$

This file contains the source code for the DELETE utility.

DELETE

The DELETE utility functions much the same as the standard DELETE provided with 65D except for a great increase in speed.

DIR

The DIR utility is similar to the usual directory utility except that it sorts the file names and displays them in four columns. The sort used is a simple (usually slow) bubble-sort which gives a good demonstration of the speed of the compiled code.

DIR$

The DIR$ file contains the source for the DIR utility.

FBASIC

The FBASIC file contains the first pass of the compiler and is invoked to start a compilation.

FBLIB

The FBLIB file contains FBASICs library of functions which are added as necessary to a compiled program. Since FBLIB is composed of several sectors on a single track it cannot be copied with a simple LOAD and PUT sequence.

FDUMP

The FDUMP utility is used to examine the contents of a disk file. It prints sixteen bytes per line in hexadecimal and ASCII. This utility is composed of both an interpreter BASIC module which is used to setup and open a file, and a compiled module which does the actual dump.

FDUMP$

This file contains the source to the compiled portion of the FDUMP utility.

OS65D3

This file contains the operating system,OSI BASIC, Assembler, and Extended monitor.

PASS2

This file contains the second pass of the compiler. It is invoked by the first pass automatically.

RENUM

This utility is used to renumber BASIC source programs. As provided it is compiled to run at $Ø2ØØ (over OSI BASIC) in order to allow the source program to be resident. Example:

```
A*CA Ø2ØØ=31,1

A*GO Ø2ØØ
Start: 1Ø
Incr : 1Ø
Pau

A*PU TEMP

A*BA
```

The "Start:" prompt refers to the number to be used as the new starting line number. "Incr :" refers to the increment to be used between line numbers.

It is wise to save the newly renumbered file to a different file until you check it for correctness. RENUM will report references to nonexistant lines to the console.

If you have memory at $DØØØ (on serial systems) or at $EØØØ the process of using RENUM can be simplified by recomipiling it to that location and calling it directly from BASIC. First recompile:

```
>RENUM$ -O RENUM -H -C EØØØ
```

Then call into memory:

```
DISK!"CA EØØØ=31,1"
```

Then whenever you need to renumber the program you currently have in memory just type:

```
DISK!"GO EØØØ"
```

Control will be returned to BASIC when the renumbering is complete.

RENUM$

This file contains the source code for the RENUM utility.

XREF

The XREF utility is used to generate full sorted cross reference listings of FBASIC programs to ease the task of keeping track of all variables and lines referenced within the program.

RENUM$

```
100 REM BASIC program renumberer
110 REM
120 REM may be compiled to $E000, if you have RAM there
130 REM and called when desired with a:
140 REM
150 REM DISK!"GO E000"
160 REM
170 REM or
180 REM may be compiled to $0200, then to use:
190 REM
200 REM EXIT
210 REM LOAD filename
220 REM CALL 0200=xx,1
230 REM GO 0200
240 REM
250 REM the file should then be saved back to disk
260 REM
270 REM
280 REM
290 REM Prompt for starting number, and increment
300 :
310 INPUT "Start: "; ST
320 INPUT "Incr : "; IN
330 :
340 DIM LN(!$B000) : REM for table of old line numbers
350 :
360 DIM D(5) : REM for digits of line number
370 :
380 AD=PEEK($3179)+PEEK($317A)*256 : REM pointer to text
390 L=ST
400 D=AD
410 :
420 REM first pass, change line no.s and build table
430 REM of old numbers.
440 :
450 LNUM=PEEK(D+2)+PEEK(D+3)*256 : REM line number
460 IF D=0 THEN 520               : REM end of file?
470 LN(N)=LNUM : N=N+1            : REM put no. in table
480 POKE D+2,L : POKE D+3,L/256   : REM change to new no.
490 L=L+IN
500 IF N>2000 THEN PRINT"Too many lines":END
510 D=PEEK(D)+PEEK(D+1)*256 : GOTO 450 : REM new pointer
520 :
530 :
540 REM second pass, find GOSUBs, GOTOs, & THENs, &
550 REM adjust references for new line numbers.
560 :
570 D=AD
580 :
590 I=D+4
600 IF D=0 THEN PRINT "PAU" : END
610 C=PEEK(I) : I=I+1
620 IF C=0 THEN D=PEEK(D)+PEEK(D+1)*256 : GOTO 590
```

```
630 :
640 REM THEN           GOTO           GOSUB           RUN
650 IFC<>160 THEN IF C<>136 THEN IF C<>140 THENIFC<>137 THEN 610

660 :
670 REM build line number
680 GOSUB 1000 : REM get char
690 P=I-1 : REM pointer to start of number
700 NUM=0 : LN=0
710 GOTO 730
720 GOSUB 1000 : REM get char
730 IF C<ASC("0") OR C>ASC("9") THEN 770
740 NUM=NUM*10+C-ASC("0") : LN=LN+1
750 GOTO 720
760 :
770 IF LN=0 THEN 620
780 :
790 REM search for match in table
800 FOR J=0 TO N-1
810 IF LN(J)=NUM THEN 860  : REM go change number
820 NEXT
830 PRINT"Bad ref to "NUM" in line "PEEK(D+2)+PEEK(D+3)*256
840 GOTO 940
850 :
860 :
870 REM change line reference to agree with new line no.
880 GOSUB 1060 : REM convert num from binary to ASCII
890 LL=L-LN
900 IF L>LN THEN GOSUB 1190 : REM block move to make room
910 FOR K=5-L TO 4 : POKE P,D(K) : P=P+1 : NEXT
920 IF L<LN THEN POKE P,32 : P=P+1 : L=L+1 : GOTO 920
930 I=I+LL : REM adjust text pointer
940 IF C<>',' THEN 620  : REM ON GOTO/GOSUB x,x,x, ......
950 IF PEEK(I)=',' THEN I=I+1
960 GOTO 680
970 :
980 :
990 REM-----get character from program
1000 C=PEEK(I) : I=I+1
1010 IF C=32 THEN 1000
1020 RETURN
1030 :
1040 :
1050 REM binary to ASCII conversion
1060 X=J*IN+ST
1070 L=0
1080 FOR K=1 TO 5
1090 Y=X/10 : DIG=X-Y*10
1100 D(5-K)=DIG+ASC("0")
1110 X=Y
1120 L=L+1
1130 IF X=0 THEN RETURN
1140 NEXT
1150 RETURN
1160 :
1170 :
1180 REM block move, used to make room for larger number
1190 T=PEEK($317B)+PEEK($317C)*256 : REM eot pointer
```

```
1200 COUNT=T-P
1210 FOR K=0 TO COUNT
1220 Q=T-K
1230 POKE Q+LL,PEEK(Q)
1240 NEXT
1250 T=T+LL
1260 POKE $317B,T : POKE $317C,T/256 : REM update eot pointer
1270 :
1280 REM fixup line pointers
1290 :
1300 K1=D
1310 :
1320 K=PEEK(K1)+PEEK(K1+1)*256+LL
1330 POKE K1,K : POKE K1+1,K/256
1340 K1=K
1350 IF PEEK(K1)+PEEK(K1+1) THEN 1320
1360 RETURN
```

CREAT$

```
 100 REM Create utility for OS-65D
 110 REM must be compiled
 120 REM
 130 PN=$4000 : REM location of directory buffer
 140 DIM A(!$4200) : REM flags for tracks in use
 150 DIM BUFF(!$4300) : REM for storage of file name
 160 :
 170 FOR I=0 TO 76 : A(I)=0 : NEXT
 180 :
 190 REM Call both sectors of directory into memory
 200 :
 210 POKE 10082,0   : REM keep head loaded
 220 DISK!"CA 4000=08,1"
 230 POKE 10082,128 : REM allow head to unload
 240 DISK!"CA 4100=08,2"
 250 :
 260 REM find all tracks in use, and find first  open  space  for
dir entry
 270 :
 280 NE=0  :  REM zero next-entry pointer
 290 FOR I=PN TO PN+511
 300 IF PEEK(I)<>ASC("#") THEN 330
 310 IF NE=0 THEN NE=I  :  REM save pointer to first open space
 320 GOTO 360
 330 T=PEEK(I+6) : GOSUB 970  : A=T
 340 T=PEEK(I+7) : GOSUB 970  : B=T
 350 FOR J=A TO B : A(J)=65535 : NEXT : REM flag tracks as in-use
 360 I=I+7  : REM instead of STEP 8 in FOR statement
 370 NEXT
 380 :
 390 REM get file name from user
 400 :
 410 INPUT "File name:";A
 420 LN=.Y : LN=LN AND 127 : REM length of input in low 7 bits of
Y reg
 430 IF LN=0 THEN END       : REM give user a way out
 440 IF LN>6 THEN 410
 450 :
 460 FOR I=0 TO LN-1
 470 BUFF(I)=PEEK($2E79+I)
 480 NEXT
 490 FOR J=I TO 7 : BUFF(J)=32 : NEXT
 500 T=PEEK($2E79)
 510 IF T<ASC("A") OR T>ASC("Z") THEN 410
 520 :
 530 REM check if name already in use
 540 :
 550 FOR I=PN TO PN+512
 560 FOR J=0 TO 5
 570 IF PEEK(I+J)<>BUFF(J) THEN 620
 580 NEXT
 590 GOSUB 1010 : REM print file name to console
 600 PRINT " already in use"
```

```
610 GOTO 410
620 I=I+7
630 NEXT
640 :
650 REM print free tracks
660 PRINT
670 FOR I=9 TO 76
680 IF A(I)=0 THEN PRINT I; " ";
690 IF POS(0)>55 THEN PRINT
700 NEXT
710 PRINT : PRINT
720 :
730 INPUT "First track:"; FT
740 T=.Y : IF T>127 THEN 730  : REM not numeric if Y reg > 127
750 IF T=0 THEN END
760 IF A(FT) THEN PRINT FT; " in use" : GOTO 730
770 INPUT "Number of tracks:";NT
780 T=.Y : IF T>127 THEN 770
790 IF T=0 THEN END
800 :
810 FLAG=0
820 FOR I=FT TO FT+NT-1
830 IF A(I) THEN PRINT I; " in use" : FLAG=1
840 NEXT
850 IF FLAG THEN 660
860 :
870 T=FT : GOSUB 1080 : BUFF(6)=T
880 T=FT+NT-1
890 GOSUB 1080 : BUFF(7)=T
900 :
910 FOR I=0 TO 7 : POKE I+NE,BUFF(I) : NEXT
920 :
930 IF NE-PN <250 THEN DISK!"SA 08,1=4000/1" : END
940 DISK!"SA 08,2=4100/1"
950 END
960 :
970 T=INT(T/16)*10 + (15 AND T)
980 RETURN
990 :
1000 REM print out file name
1010 FOR K=0 TO 5
1020 TT=BUFF(K)
1030 IF TT<>32 THEN PRINT CHR$(TT);
1040 NEXT
1050 RETURN
1060 :
1070 REM subroutine to convert T to binary coded decimal (BCD)
1080 T=(T/10)*16 + T MOD 10
1090 RETURN
```

DIR$

```
100 REM Source for "DIR"
110 :
120 :
130 REM Directory utility for OS-65D
140 REM may be interpreted or compiled with FBASIC
150 REM
160 NF=-1
170 PN=16384 : REM location of buffer
180 DIM A(64)
190 FOR I=0 TO 64 : A(I)=0 : NEXT
200 :
210 REM Call both sectors of directory into memory
220 :
230 POKE 10082,0   : REM keep head loaded
240 DISK!"CA 4000=08,1"
250 POKE 10082,128 : REM allow head to unload
260 DISK!"CA 4100=08,2"
270 :
280 GOSUB 320
290 PRINT : PRINT NF+1 " Files Defined"
300 END
310 :
320 :
330 REM
340 PRINT
350 :
360 REM Build array of pointers to file names
370 :
380 FOR K=0 TO 63 : I=K*8+PN
390 IF PEEK(I)<>35 THEN NF=NF+1 : A(NF)=I
400 NEXT K
410 :
420 REM Sort array
430 :
440 FL=0
450 FOR I=0 TO NF-1
460 FOR J=0 TO 5
470 T1=A(I)+J : T2=A(I+1)+J
480 IF PEEK(T1)>PEEK(T2) THEN 510
490 IF PEEK(T1)<PEEK(T2) THEN 530
500 NEXT J
510 FL=1
520 T=A(I):A(I)=A(I+1):A(I+1)=T
530 NEXT I : IF FL THEN 440
540 :
550 REM Print out in 4 columns
560 :
570 T=NF+1-INT(NF/4)*4 : REM can also be expressed as:
580 :                      REM T=(NF+1) MOD 4
590 ROWS=INT(NF/4) : IF T THEN ROWS=ROWS+1
600 :
610 FOR K=0 TO ROWS-1
620 FOR L=0 TO 3
```

```
630 J=L*ROWS+K
640 IF A(J) THEN GOSUB 700
650 NEXT
660 PRINT
670 NEXT
680 RETURN
690 :
700 :
710 FOR I=A(J) TO A(J)+5:PRINTCHR$(PEEK(I));:NEXT
720 PRINT " ";
730 Z=PEEK(I):GOSUB790
740 PRINT"-"; : Z=PEEK(I+1):GOSUB790
750 PRINT"   ";
760 RETURN
770 :
780 REM-----Convert BCD to ASCII and print it
790 PRINT CHR$(Z/16+ASC("0")) CHR$((ZAND15)+ASC("0"));
800 RETURN
```

DEL$

```
 100 REM Delete utility for OS-65D
 110 REM must be compiled with FBASIC compiler
 120 REM
 130 NF=-1
 140 PN=$4000 : REM location of directory buffer
 150 DIM BUFF(7) : REM file name buffer
 160 :
 170 REM Call both sectors of directory into memory
 180 :
 190 POKE 10082,0   : REM keep head loaded
 200 DISK!"CA 4000=08,1"
 210 POKE 10082,128 : REM allow head to unload
 220 DISK!"CA 4100=08,2"
 230 :
 240 REM get file name from user
 250 :
 260 INPUT "File name:";A
 270 LN=.Y : LN=LN AND 127 : REM get length of input (low 7  bits
of Y
 280 IF LN=0 THEN END       : REM give the user a way out
 290 IF LN>6 THEN 260
 300 :
 310 FOR I=0 TO LN-1
 320  BUFF(I)=PEEK($2E79+I)  :  REM  input  is  buffered  at
$2E79,moveto BUFF
 330 NEXT
 340 FOR J=I TO 7 : BUFF(J)=32 : NEXT
 350 T=PEEK($2E79)
 360 IF T<ASC("A") OR T>ASC("Z") THEN 260 : REM first  char  must
be alpha
 370 :
 380 REM search for name in directory
 390 :
 400 FOR I=PN TO PN+512
 410 FOR J=0 TO 5
 420 IF PEEK(I+J)<>BUFF(J) THEN 480
 430 NEXT
 440 REM found name
 450 GOSUB 660  : REM print file name to console
 460 NE=I
 470 GOTO 570
 480 I=I+7 : REM instead of STEP 8 (not supported by FBASIC)
 490 NEXT
 500 :
 510 GOSUB 660  : REM print file name to console
 520 PRINT " not found"
 530 GOTO 230 : REM go ask for file name again
 540 :
 550 REM routine to delete entry from directory
 560 :
 570 FOR I=0 TO 7 : POKE I+NE,ASC("#") : NEXT
 580 :
 590 IF NE-PN < 250 THEN DISK!"SA 08,1=4000/1" : GOTO 610
```

```
600 DISK!"SA 08,2=4100/1"
610 PRINT " deleted"
620 END
630 :
640 REM subroutine to print file name to console
650 :
660 FOR K=0 TO 5
670 TT=BUFF(K)
680 IF TT<>32 THEN PRINT CHR$(TT);
690 NEXT
700 RETURN
```

FDUMP$

```
 100 REM  Compiler portion of FDUMP utility
 110 :
 120 REM  The file is opened by the OSI BASIC program that  calls
this
 130 :
 140 DIM R(15)
 150 :
 160 PRINT  :  LC=1
 165 BC=0
 170 :
 180 WHILE 1          :  REM loop  till  end  of  file  (aborts  on
error)
 185 .A=BC/256        :  REM high byte of byte counter
 186 GOSUB !$2D92     :  REM print in hex
 187 .A=BC            :  REM low byte of byte counter
 188 GOSUB !$2D92
 189 PRINT " ";
 190 FOR I=0 TO 15    :  REM display 16 bytes per line
 200 GOSUB !$23A1     :  REM direct call to input from disk  device
#6
 210 CH=.A            :  REM get character from Accumulator
 220 GOSUB !$2D92     :  REM call system hex output routine
 230 PRINT " ";       :  REM print a space to separate the bytes
 240 R(I)=CH          :  REM save character to print later
 250 GOSUB 410        :    REM  check  for  control-C  &  abort  if
necessary
 255 BC=BC+1          :  REM increment byte counter
 260 NEXT
 270 :
 280 REM now print in ASCII
 290 FOR I=0 TO 15
 300 CH=R(I)
 310 IF CH>=32 AND CH<128 THEN PRINT CHR$(CH); : GOTO 330
 320 PRINT ".";          :  REM just print a dot if not a  printing
char
 330 NEXT
 340 PRINT
 350 REM  do formfeed every 60 lines for printer
 360 LC=LC+1  :  IF LC=60 THEN PRINT CHR$(12); : LC=0
 370 WEND             :  REM bottom of main loop
 380 :
 390 :
 400 REM control-C check
 410 IF PEEK(10950)=1 THEN 430   :  REM master system I/O id
 420 :                              REM 1=serial, 2=video
 422 :
 425 POKE 57088,1 : REM strobe keyboard (video)
 426 IF (PEEK(57088) AND 64)=0 THEN RETURN
 427 POKE 57088,4
 428 IF PEEK(57088) AND 64 THEN END
 429 :
 430 IF (PEEK($FC00) AND 1)=0 THEN RETURN : REM check port  input
status
```

```
 440 IF (PEEK($FC01) AND 127)=3 THEN END  : REM abort if control-
C (3)
 450 RETURN
```

FDUMP

```
 100 REM   FDUMP
 120 REM
 140 REM Set disk buffer to the top  of  memory  (48K)  where  it
belongs!
 145 :
 150 REM (up at $D000 or above is even better  if  you  have  RAM
there)
 160 :
 180 POKE 8998,0 : POKE 8999,180 : REM START OF BUFFER
 200 POKE 9000,0 : POKE 9001,192 : REM END OF BUFFER PLUS 1
 220 :
 240 REM ADJUST MEMORY SIZE
 260 :
 280 POKE 132,255 : POKE 133,179
 300 POKE 8960,179: REM Master sys-mem size in pages (less 1)
 320 :
 340 PRINT "File dump utility"
 360 PRINT
 380 PRINT "Abort with control-C"
 390 PRINT
 400 :
 420 INPUT "File name";F$
 440 :
 460 DISK 0,6,F$
 480 :
 500 DISK!"GO 317E"
```

LINKAGE TO 65U

A program compiled with the -U option may be combined with a 65U program and SAVEd and RUN as a single package via the LOAD48 utility included with your 65U operating system.

The compiled code may be transfered as per the instructions on page 45 of the OS-65U Operators Manual.

To call the compiled module from OSI BASIC you first set up the USR vector with the following two POKEs:

```
POKE 8778,0 : POKE 8779,96 : REM $6000
```

Then at any point within the OSI BASIC program you desire to call the compiled module simply insert:

```
X=USR(X)
```