SM800

BASIC LANGUAGE REFERENCE

1.1 BASIC Programs / Displays

The BASIC compiler, as well as any BASIC programs, run in the background with the real time scan functions of the Sensor Modem.

BASIC can be accessed by linking directly from the command mode. ("LINK,BASIC", "ONLINE")

If you are currently in the Real-time Data Displays, or other linked functions other than BASIC, you must use the ESCAPE SEQUENCE (three ^X's) to return to the command level and then LINK to basic.

By going "ONLINE" linked to BASIC, any BASIC user program display output is presented on your terminal, and any keyboard inputs are available for the user program to read.

BASIC programs are very convenient to create custom summary displays or other unique operator interface requirements.

Details of accessing real-time registers from BASIC programs is discussed in section 9.6.33

Programs that are SAVED for permanent use and set for AutoStart, are suitable for continuous background operation without any further operator access or attention. Programs of this type allow complex control algorithms to be implemented in a "high level" language and the outputs passed directly to the real-time registers.

Programs may also be set up which are operator interactive and provide specialized operator displays. This type of program is useful in replacing dedicated hardware panel meters and other "panel mounted" indicators.

Specific operator interactions with BASIC display programs is determined by the system implementor and the particular BASIC program installed.

1.2 BASIC Compiler

1.2.1 Overview

The optional BASIC compiler (BASIC180) is a multitasking programming environment completely available to the user. It provides all programming capabilities. Editing or running programs can be done concurrently with on-line operation of the real-time system.

BASIC runs in a background mode to the real-time scanning, data processing and communications functions of the Sensor Modem. It can freely read (or write) real-time data from the register tables. The serial ports can be utilized by BASIC programs. (when not assigned for active use by the real-time system)

Programs can be interactively edited and run through either local or remote inter-active links.

Floating point and integer functions are supported, as well as, numerous advanced mathematical functions.

Windowing display support is provided for Heath 19 CRT terminals.

LOAD and SAVE functions will retrieve or store programs to the optional memory cartridge.

Note: You must use the LOAD and SAVE functions or the BASIC program will be lost upon loss of power. Unlike the run time part of the system the BASIC program is not automatically saved. See section 9.4.9 for automatic load & run.

1.2.2 Access to BASIC (Inter-Active)

Access to BASIC from the terminal display and keyboard is accomplished from the command level by entering the following:

LINK,BASIC

ONLINE

Return to Command Mode while linked to BASIC can be accomplished by entering three ^X's within one second.

Note: Exiting to command mode does not necessarily end execution of a BASIC program which can continue to run in background.

If a BASIC program is currently running, it can be stopped by entering ^C (Control C).

If a BASIC program is not running, the compiler is waiting for commands or input of statements. Entry of non-numbered statements or commands are referred to as Immediate Commands. Immediate commands include LIST, COMPILE, RUN, SAVE, LOAD, NEW, GO.

While a BASIC program is running, the program determines what keyboard inputs are allowed and what responses are generated.

1.2.3 BASIC Access to Real-Time Registers

Registers in TABLE 1, TABLE 2, Summary Registers and Historical Data can be accessed from a BASIC program.

Access to real-time data registers is accomplished with two BASIC statements: TBLRD and TBLWRT.

TBLRD is a function which returns the specified register or parameter. The syntax is I=TBLRD(J) , where J specifies the register or parameter, and I receives the value. The syntax of TBLWRT is TBLWRT J,K , where J specifies the destination register or parameter and K is the value to be written to the destination.

The J variable is an integer value. The usage of this function allows access to several different parameters. See section 7 for syntax and specific request information.

1.2.4 BASIC Access to Historical Data

Historical data (recorded samples) can easily be accessed by a BASIC program by using the HST function. The variable associated with this function request can be set so that all necessary information can easily be retrieved for use in the BASIC program. Each sample consists of time stamp, point #, station #, point type (digital or analog), timed or event flag, and the actual data value.

Additional information available include the current sample index, the most recent sample number (top), the oldest sample number (end), maximum sample number (size) and current word index. The parameter also permits selection of the same word in the previous sample. This facilitates searching by specific time, point or station criteria.

See section 9.6.33 for specific command syntax.

1.2.5 Access from BASIC program to Auxiliary Ports

A BASIC program can access the external world through two paths to the physical and logical ports in the Sensor Modem. The BASIC program can LINK to ports similarly to the way you would manually LINK to a desired port.

Inside BASIC, these redirections of keyboard and display data (via GET and PRINT statements) are accomplished by the FILE statement. FILE 5 and FILE 7 are the two paths for redirected I/O. (see 9.6.7)

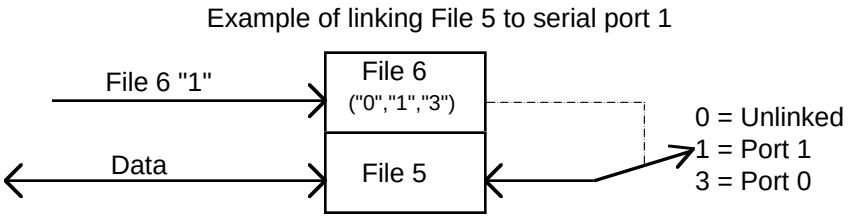

Figure .0-1
File Statements

1.2.6 Report Printing

Printer output can be routed to a serial port and then print output will be routed to the assigned port.

1.2.7 Non-Standard Instrument Interfaces/Secondary Port

Within a BASIC program, character I/O can be routed to either the "console" port, or an auxiliary port. The normal console port is the one which is typically used to manually enter BASIC commands, list programs, and other operator interactions. The connection from the user to this port is via the "LINK, BASIC" entry while at the command level.

The secondary BASIC port is the HOST port (LINK,HOST). These ports are the outside connections to a BASIC program. They are logical ports within the Sensor Modem. Like any of the Sensor Modem's logical ports, they can be "linked" to other logical ports or, physical serial ports.

Within a BASIC program (or task), Serial I/O can be directed to a selected output by using the FILE statement. (not to be confused with the File device files for program storage) The BASIC FILE statement parameter can be: 0- Console I/O, or 5 for the secondary (HOST) port I/O. By using BASIC's FILE 5 statement, data is diverted from the console device to the secondary port.

By logically linking the HOST port to a physical serial port, character I/O can be routed from a BASIC program to the serial port. Using this technique, a task can be set up within the program to exchange data of virtually any format.

1.2.8 Specialized Calculations

Basic programs may access real-time data and utilize log, trigonometric, and other floating point functions for specialized calculation requirements.

Common control algorithms used for pumping and other applications are available in the form of a library of example routines. These may be integrated into a customized program to facilitate the program development.

1.2.9 Multiple Tasks

The BASIC statements include statements to support multiple time-shared BASIC tasks. Up to 10 independent task programs may be concurrently active.

This feature allows the programming to be modular and to minimize problems of timing and undesirable task inter-actions.

As an example, one task may be performing a control calculation while another is displaying data or accepting keyboard commands. Each task can continue to execute independently of the other. Common variables can be accessed by either task. This allows easy communications and coordination between tasks.

However, caution should be exercised when programming multiple-task programs, since improper priority conflicts can prevent expected operation. Also, there are some restrictions on multiple tasks performing I/O to a single device.

1.3 BASIC Utility Functions

There are several utility type operations that are provided for use in BASIC programs.

These utility functions are accessed through the BASIC "SYSTEM UTIL array(0)" statement. The "SYSTEM" statement is essentially a call to assembly language subroutines built into the Sensor Modem firmware. (The UTIL variable must be defined and explicitly set to $7520) Parameters are passed to and from the SYSTEM utility functions via an integer array with at least 4 integer elements (0-3).

Functions available via the **SYSTEM UTIL,R(0)** statement are:

- CONVERT HI BYTE TO HEX ASCII.

(AND SWAP BYTES IN INTEGER VALUE)

- CONVERT LO BYTE TO ASCII DECIMAL.
- CONVERT INTERNAL PACKED TIME TO SS, MM, HH. (Integers)
- GET STATION NUMBER. (Integer)
- GET CURRENT TIME AS SS, MM, HH. (Integers)
- GET CURRENT DATE AS DD, MM, YY. (Integers)

(See "SYSTEM" statement in section 9.6.32 for details.

1.4 Direct Commands

Direct commands are those commands given to BASIC-180 to control its operation. Direct commands are not part of BASIC-180 programs, but are entered via a keyboard by the user. These are the BASIC-180 direct commands:

```
CLS      LOAD
COMPILE          NEW
CONSOLE          NOERR
END      RLOAD
ERROR RUN
FILEFORMAT    SAVE
GO       STATUS
LIST
```

Figure.0-2
Direct Commands

1.4.1 CLS

CLS clears the CRT screen to blanks and places the cursor in the upper-right corner of the screen. BASIC-180 must be properly installed for the terminal being used for CLS to work correctly.

1.4.2 COMPILE

COMPILE causes BASIC-180 to convert the program currently in memory to machine language. COMPILE is identical to the RUN direct command except that the compiled program is not executed. The compiled program may be executed using the GO command. Example:

```
>LOAD 11

>COMPILE

COMPILED

>
```

1.4.3 CONSOLE

CONSOLE causes output previously directed to the printer by the PRINTER command to be directed back to the system console. CONSOLE is the default setting.

10 PRINT"HI"

>RUN

COMPILED

HI

PRINT statement sent "HI" to the console

>PRINTER

RUN

>

Nothing happens on the screen because the program sent "HI" to the printer. To redirect output to the screen, type:

>CONSOLE

1.4.4 END

END is a direct command used only in files containing BASIC-180 source programs. During the LOAD command, "END" indicates to the compiler the end of the file containing BASIC-180 statements. Whenever the SAVE command is used to store a program on disk, BASIC-180 inserts an END as the last item in the file.

The only time the user has to use the END command is when a BASIC-180 program is created off-line with a text editor or word processor. In this case, an END with no line number should be the last line in the file. For example:

10 PRINT "This program was created using an editor."

END

1.4.5 ERROR

ERROR turns on the compiler's runtime error checking software. It is the converse of NOERR. When BASIC-180 is first started, all runtime error checking is on and stays on until explicitly disabled by NOERR.

Runtime error checking is essential in most programs to insure that mistakes don't "blow up" the compiler. For example, if a "SUBSCRIPT OUT OF RANGE" error was not detected, a program could write over itself, or even over the compiler, causing unpredictable and undesirable results.

Like NOERR, ERROR modifies how the code is generated during compilation of the program, so it must be specified before the program is RUN or compiled (if you have used NOERR in the BASIC-180 session before RUN or COMPILE). ERROR will then remain in effect unless disabled by NOERR.

CAUTION: Whatever debugging work is needed should be done with ERROR on (the default setting). When checkout is complete, compile using NOERR for faster code:

>NOERR

>COMPILE

1.4.6 FILEFORMAT x0

where x specifies the File Device to be initialized.

There are two BASIC file storage areas available for user programs. The first is in internal nonvolatile RAM and the second is in the removable RAM memory cartridge.

The first area, in internal RAM, is known as file device 1. The area in the removable memory cartridge is known as file device 2. Either of these "RAM disks" may be formatted using the FORMATFILE command. Formatting initializes the storage area and erases any existing files in the file device.

See LOAD (9.4.9) and SAVE (9.4.14) commands for details of storing and retrieving program files.

1.4.7 GO

The GO command starts execution of the most-recently COMPILED (or RUN) program. If the source code has been modified, a NEW command has been executed, or there was an error in the last compile, then the NO COMPILED CODE error results.

GO is typically used to run a program previously compiled using the RUN or COMPILE commands. If the program is very long, then GO is faster than using RUN repeatedly (since RUN must first recompile the program).

```
>LOAD 11

>COMPILE  !to check for errors

>GO                          !program executed
```

1.4.8 LIST <line1, line2>

LIST displays the program currently in memory on the system's console. If a PRINTER command has been issued, the program will also be listed on the printer.

If LIST is typed with no arguments, the entire program is listed. If only one line number is specified, just that line, if it exists, will be listed. If two arguments are given, then the set of program lines between and including the two line numbers will be listed.

Examples:

```
>LIST

>LIST 100

>LIST 100, 200
```

1.4.9 LOAD xy

LOAD xy where x is the file device and y is the file number

LOAD retrieves the specified BASIC program file from either of 2 File Devices. File Device 1 is internal non volatile memory. File Device 2 is in the removable memory cartridge. The first digit of the parameter specifies the file device and the second digit specifies the file number (1 through 9), the second digit is always zero.

An "autoload" can be set up by saving the program to FILE 11 and then setting dip switch #2 to the down position. The unit will automatically load and run the program in FILE 11 on startup. (A mini autoexec.bat function).

Also see the FILEFORMAT (9.5.6) command for discussion of initialization of the File Device.

1.4.10 NEW

NEW erases the program currently in memory.

```
>LOAD 11

>RUN
```

program executes

>NEW

>LOAD 12

NOTE: If you did not use the NEW command, FILE 11 and FILE 12 would be merged (in order by line number). This may not be the desired effect, but is sometimes useful.

1.4.11 NOERR

NOERR turns off much of the runtime error checking of BASIC-180, resulting in a program that runs much faster.

CAUTION: When using the automatic file load by setting dip switch #2 down the unit will start in NOERR mode.

To insure that a program doesn't run wild and destroy itself or the compiler, BASIC-180 inserts quite a lot of error-checking code into the compiled program. For example, tests are made for "SUBSCRIPT OUT OF RANGE," "NEXT WITHOUT FOR," and so on. NOERR also removes the test for Ctrl-C. (Ctrl-C aborts a running program, so a program compiled under NOERR can't be stopped.) Most of these tests are removed by specifying NOERR. Some error-checking routines remain since, in some cases, the error-checking code does not greatly affect execution speed.

An increase in execution speed of several-hundred percent can often be realized by using NOERR. The compiled program will also be significantly shorter. Be warned, though. NOERR permits the user's program to execute erroneously, possibly trashing BASIC-180 or the operating system. Only fully-debugged programs should be compiled under NOERR.

When the compiler starts, all error checking is enabled and remains on until NOERR is specified. NOERR affects the way a program is compiled, so it must be specified before the program is compiled or RUN.

CAUTION: Whatever debugging work is needed should be done with ERROR on (the default setting). When completed, compile using NOERR for faster code.

>NOERR

>COMPILE

1.4.12 RLOAD

The RLOAD is a direct command that is put at the beginning of a BASIC source program file. During the LOAD command it tells the compiler not to echo the lines back to the CRT, thus the loading of the file is significantly faster.

If there is an error, the line number does not show, so this is usually used after normal program development and minor changes have been made.

1.4.13 RUN

RUN compiles and executes a program. The program is not executed if any compilation errors are found.

>LOAD 11

>RUN

>COMPILED

program executes

>LOAD 12

>RUN

STRING LENGTH EXCEEDED 110

>Compilation error detected, program not executed.

1.4.14 SAVE xy

SAVE stores the current BASIC program on either of 2 File Devices. File Device 1 is internal non volatile memory. File Device 2 is in the removable memory cartridge.

The first digit of the parameter specifies the file device and the second digit specifies the file number (1 through 9).

Also see the FILEFORMAT command for discussion of initialization of the File Device.

1.4.15 STATUS

STATUS displays the start and end address of the compiled program, along with the amount of free memory and the starting address of the variables used in the program. The space between the end of the program and the start of the variable storage may be used for assembly language routines.

```
>LOAD 12

>COMPILE

COMPILED

>STATUS

Start: A56F

End: A619

Variable: FBFC

Starting address      6EOF

Ending address        794C

Variable start              F619
```

1.5 Components of Statements

A statement is composed of the name of the statement itself (for example, GOTO) along with optional parameters. These parameters may consist of variables, constants, functions, operators, or combinations of all of the above.

1.5.1 Variables

BASIC-180 variables are formed by a letter followed by up to six letters or digits. For example, A is a legal variable, as well as AO, A1, HI92, and JUMP. However, 9AB is not a legal variable, nor is A1234567899.

String variables are formed the same way, but are followed by a dollar sign. HIYA$, A1$. and A9$ are all legal string variables.

A maximum of 255 different variables may exist in any given program. These variables may be in any form within the rules given above. Note that integer or real variables may not have the same name as string variables. For example, the variable A may not be used in the same program that uses the variable A$. BASIC-180 will try to use A and A$ as the same variable and a string variable error will result. Similarly, a dimensioned variable must not have the same name as an undimensioned variable (for example, you cannot use B and B(n) in the same program).

All variables must be declared at the beginning of the program before any executable code is encountered. In practice, this means that the variable declaration statement (INTEGER, REAL, and STRING) should be the first statements in the program. If undeclared variables are found during the compilation, the compiler will display an error message.

For strings, the string length is specified in the STRING statement (for example, B$(80)). If no string length is specified, a string length of 20 will be assigned. The maximum allowable string length is 127.

String arrays are defined by specifying the length of each element, followed by the number of elements in the array. STRING A$ (10,20) specifies 20 strings, each of length 10. Any element can be accessed just like a single-dimension array. A$(3)="123" assigns a value to the third element of the string array.

Subscripted variables are specified within the INTEGER and REAL statements. Note that subscripted variables start with the zero dimension and extend to the maximum dimension specified. Therefore, the statement INTEGER A(10), defines a variable with eleven members, A(0) through A(10).

1.5.2 Numbers

Integer variables are stored using s sixteen-bit two's complement representation. An integer value can range from +32,767 to -32,768. Positive values which exceed +32,767 will appear as negative numbers.

Real values are four-byte (32-bit) IEEE-compatible single-precision real numbers. This means that approximately 6.5 digits of precision are maintained for real numbers. Many BASIC interpreters and compilers use BCD mathematics or 64-bit representations resulting in high-accuracy numbers that require lots of memory. BASIC-180 does not support either of these in the interest of maximizing speed.

The user must be aware that a real number may not be exactly the number anticipated. For example, since real numbers are constructed by using powers of two, the value 0.1 cannot be exactly represented. It can be represented very closely (within 2 -23), but it will not be exact. Therefore, it is very dangerous to perform a direct equality operation on a real number. The statement "IF A= 0.123" (assuming A is real) will only pass the test if the two values are exactly equal, a case which rarely occurs. This is true for all real relational operators including, for example, the statement "IF a>B" if values very close to the condition being measured are being used. Be aware that the number you expect may not be exactly represented by the compiler. If necessary, use a slight tolerance around variables with relational operators.

1.5.3 Constants

Constants are formed by combining decimal digits with an optional decimal point. Whenever a decimal point is included in a constant, the compiler assumes this constant is a real number. If the constant is expressed without a decimal point, the compiler assumes that the value is an integer. This has significance when combining real and integer values within an expression (see Chapter 7). Even though BASIC-180 is smart enough to convert constants between integer and real as required, programs will run more efficiently if modes are not mixed.

When expressing constants between -1 and 0, a zero should be used between the negative sign and the decimal point. For example, "-0.25" should be used and not "-.25".

BASIC-180 provides a facility for using hexadecimal constants. Hexadecimal constants are specified with a leading dollar sign..$1AB represents the hexadecimal constant 1AB.

1.5.4 Mode Conversions

BASIC-180 is unlike many BASICs in that it forces the user to declare the mode of each variable, thereby optimizing the compiler's speed. With all variables pre-declared, the compiler is not forced to evaluate all expressions in floating point at run time (which is a very slow procedure). and then convert to integer as the need arises. Instead, the algorithms used in BASIC-180 attempt to evaluate all expressions in the output mode (the mode of the variable to which the expression is being assigned).

To make it easier to write programs, BASIC-180 provides automatic mixed-mode expression evaluation. This means that an expression may consist of a combination of real and integer values. BASIC-180 will automatically convert the components of the expression to the proper mode of the variable to which the expression is being assigned. This is very convenient for programmers; however, there are some important implications arising from it.

Whenever an expression is to be assigned to a real variable, every component of that expression is evaluated in real mode. Components of the expression which are integer (for example, integer variables) are automatically converted to real before any arithmetic is performed. This conversion takes place entirely within temporary values in the compiler; the integer values themselves are not changed. Whenever a constant is specified with no decimal point, the compiler assumes that it is an integer value. Any constant designated with a decimal point will be assumed to be real. Since the process of converting an integer to a real is relatively slow, faster code will result with real operations when all real operands are specified.

Expressions are defined in terms of parenthesis. Whenever an expression in parenthesis is encountered, it is treated as a new expression although it may be part of a larger expression. This has significance when expressions are being evaluated which will be assigned to integer arguments. When the compiler encounters a new expression (one with parentheses). it attempts to evaluate that expression in the mode of the variable to which it will be assigned. In the case of a real operator this is not important since all values are converted to real before any operation takes place. With integer variables, however, if any component of an expression is real, the rest of that expression will be converted to real before the operation takes place. A few examples will make this clear.

```
10  INTEGER A

20  A=(1/2)*2
```

In this case, the expression will evaluate to the value zero. All operations specified are integer. Integer operations take place by truncating the result, so one divided by two evaluated to zero.

```
10  INTEGER A

20  A= (1.0/2)*2
```

This expression also evaluates to zero, but for a different reason. The inner 1.0/2 evaluates to 0.5, but after the value is calculated, the compiler attempts to convert this back to integer to be in the proper mode for variable A. The integer version of 0.5 is 0.

```
10  REAL A

20  A=(1.0/2)*2.0
```

In this case, the expression will evaluate to 1. Each of the operations is real, so all operations take place in real mode.

1.5.5 Operators

Operators are connections within expressions that perform logical or mathematical computations.

1.5.5.1 Integer and Real numbers

These operators work with both integer and real numbers:

+	addition
-	subtraction
*	multiplication
/	division

Figure.0-3
Mathematical Operators

Note: see 9.7.11 for exponential math functions.

1.5.5.2 Relational

Some operators are relational. They generate a nonzero result if their condition is met. These operators may be used in mathematical expressions, but they are more frequently used with IF/THEN statements. A variable may hold the result of a relational comparison. For example,
A = R > 0.

>	greater than
<	less than
<> **or** ><	not equal to
=	relational equality test
>=	greater than or equal (integers and reals)
<=	less than or equal (integers and reals)
AND	logical AND (integer, reals automatically converted)
OR	logical OR (integer, reals automatically converted).

Figure.0-4
Relational Operators

Note: Strings are not supported by >= or <= and for concatenation you must use CONCAT$ instead of + .

1.5.5.3 Precedence

All operators are in a hierarchy that defines what operators will be evaluated first. The following is a list, from the highest to lowest priority.

***, /**
+, _
unary -, >, <, <>, ><
AND, OR.

Figure.0-5
Precedence Operators

1.6 Statements

A statement is an instruction in a program which specifies what action the program will take. These are the BASIC-180 statements:

CANCEL		INPUT	STOP
CLS	INPUT$	STRING	
CURSOR		INTEGER	SYSTEM
DATA	NEXT	TASK	
DEF	ON GOSUB		TBLWRT
DEFMAP		ON GOTO	TRACE OFF
ERASE	POKE	TRACE ON	
EXIT	PRINT	WAIT	
FILE	PRIORITY		WCLEAR
FNEND	RANDOMIZE		WFRAME
FOR/NEXT		READ	WINDOW
FPRINT	REAL	WPOKE	
GOSUB	REM	WSAVE	
GOTO	RETURN		WSELECT
IF/THEN		RUN	WUPDATE

Figure.0-6
Statements

All statements must be preceded by a line number. This line number determines the position of the statement within the program. For example, if a statement is entered with a line number of 10, and later on a statement is entered with a line number of 1, the statement numbered 1 will appear in the program before the statement numbered 10. This is BASIC-180's primary editing facility. To delete a statement, simply type its line number followed by a return. To replace a statement, type the statement's line number followed by the new statement. These procedures are the same as those used on many interpreters.

Multiple statements may be on the same line. Each statement must be separated by a colon. For example:

```
10   PRINT "HELLO"; : PRINT "BOB" : ! OUTPUT IS HELLO, BOB
```

1.6.1 CANCEL <task>

CANCEL stops the specified task from being started again when the schedule interval given in the RUN statement elapses. CANCEL does not immediately abort the task--only EXIT or STOP can stop the execution of a task. When a RUN statement is entered, a schedule interval is specified. CANCEL causes the scheduling of the task to cease. When the CANCEL is executed, the task continues executing normally until it EXITs or its current time slice expires. The task will not run again until it is specifically invoked using another RUN statement.

CANCEL is useful when a task needs to only run once. The task CANCELs itself.

1.6.2 CLS

CLS is the same as ERASE. It erases the contents of the CRT. BASIC-180 must be configured before using CLS.

1.6.3 CURSOR

CURSOR <row coordinate>, <column coordinate>

CURSOR is used to position the cursor within a window on the screen. Like all window-related commands, it will function properly only if BASIC-180 has been configured.

CURSOR's arguments are the logical row and column coordinates of the desired window position. These are logical values, meaning that they are referenced to the current window's upper-left-hand corner. CURSOR 0,0, therefore, positions the cursor to the current window's upper-left-hand corner.

1.6.4 DATA <data list>

DATA statements are used to define a block of constants which will be read by READ statements. All DATA statements must be in your program before the first READ statement.

1.6.5 DEF <function name> {<arguments>}

The DEF statement marks the beginning of a user-defined function. The function name must follow the rules for variable names. Arguments for the function are optional. Also see FNEND for more information.

1.6.6 DEFMAP <address>

The DEFMAP statement is used to provide access to memory outside of the current logical address space. See Chapter 6 for more detail on logical/physical addresses.

DEFMAP causes the PEEK, POKE, WPEEK, and WPOKE statements and functions to operate on memory anywhere in the physical address space of the HD64180. When BASIC-180 is started, DEFMAP defaults to a -1, which has the effect of accessing only the current logical map.

If a DEFMAP is issued with an argument other than -1, all PEEK, POKE, WPEEK, and WPOKE statements form their physical addresses as follows:

Physical address = (DEFMAP address) * 4096+ (PEEK address) AND $OFFF

1.6.7 EXIT

This statement must be used whenever a TASK is defined. It must be the last statement in a task to stop the TASK from continuing until called again. If the EXIT is missing, no other TASK will start.

1.6.8 FILE <file number>

FILE causes all subsequent I/O from PRINT, FPRINT, INPUT, and INPUT$ statements to be sent to the device specified by <file number>. The file number definitions are:

0 = console
5 = alternate device 1 (Linked as HOST)
6 = (associated control for file 5)
7 = alternate device 2 (Linked as HOST)
8 = (associated control for file 7)
9 = read phone numbers, station name

Figure.0-7
FILE types & values

Device numbers to use with the File statement

0	Unlinked	
1	Aux Serial Port	LINK,P1
3	Aux Serial Port	LINK,P0
4	Internal Dial Modem Port	LINK,COMM
7	BASIC Compiler	LINK,BASIC
8	Aux BASIC	LINK,HOST
9	Real-Time Displays (user 1)	LINK,RTSYS
A	Real-Time Displays (user 2)	LINK,RT2

Figure.0-8
Device numbers

The FILE statement sets the file number for the task that issues the statement. Each task may have a different active file number, but only a total of four files may be active at one time.

Doing a 'GET' character to read from either File 6 or 8 will return the port number that is linked. This allows the program to verify if a link was made. The link request may not be successful if the requested port is already linked or busy.

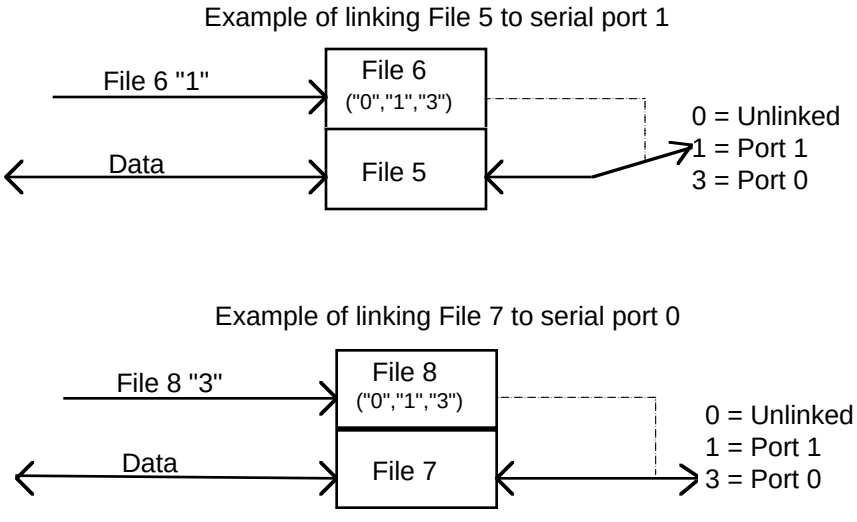

Figure.0-9
File Linking

```
10  INTEGER LEVEL,FLOW
20  STRING A$, B$
30  A$ = "FLOW = ": B$ = "LEVEL = "

100 FILE 6: PRINT "1": 'set up port 0 for serial output
110 FILE 8: PRINT "3": 'set up port 1 for serial output
130 LEVEL = TBLRD(1): 'read line 1, table 1
140 FLOW = TBLRD(2): 'read line 2, table1

200 FILE 5: ' next FPRINT will go out port 0
210 FPRINT A$,FLOW,B$,LEVEL

220 FILE 7: ' next FPRINT will go out port 1
230 FPRINT A$,FLOW,B$,LEVEL

240 FILE 6: PRINT "0": 'set file 5 back to console
250 FILE 8: PRINT "0": 'set file 7 back to console
```

1.6.9 FNEND

The FNEND statement marks the end of a user-defined function. Also see DEF.

1.6.10 FOR

FOR <variable> = <expr1> TO <expr2> {STEP <expr3>}

This statement will start execution of all statements between itself and the first NEXT it finds. Initially, <variable> is set to <expr1>. During each iteration of the loop, <variable> (which must be a non-subscripted variable) is incremented by 1 (the default value) or by <expr3>. The statements within the loop will be executed until <variable> is equal to or greater than <expr2>. *Loops may be nested.*

1.6.11 FPRINT <format>, <expr list>

FPRINT allows formatted printing on the screen. Using the <format> string, the list of expressions, <expr list>, is printed. FPRINT is a much more sophisticated version of the PRINT statement since FPRINT allows the programmer to precisely control the format of data output by the program.

If the width specified for a field is not wide enough to contain the item printed, then asterisks are printed. For example, if a format of H2 is specified, and the number output is $3344, then "**" will be printed. Whenever a field overflow of this sort occurs, the field width specified in the format statement sort occurs, the field width specified in the format statement (in this case 2) will be filled with asterisks.

In the <format> string the following symbols are valid:

Hn Prints n hexadecimal digits, truncating if needed. Leading zeros are printed.

In Prints n integer digits. Leading zeros are converted to blanks before being printed.

Sn Prints n characters of a string.

Xn Prints n spaces.

Fm.n Prints m digits before the decimal point and n digits following it. Leading zeros are converted to spaces and trailing zeros are left as zeros. No more than 6 positions after the decimal point are allowed.

Z Suppresses the carriage return at the end of the line. This is only legal at the end of the format string.

Figure.0-10
FPRINT formats

Examples:

```
10  FPRINT "H3, X1, F3.4",X,Y
```

If X=$2AB and Y=123.123456, then the output will be:
2AB 123.1234

```
20  FPRINT "S5, X1, S4",A$,B$
```

If A$="Hello" and B$="John Doe", then the output will be:
Hello John

```
30  FPRINT A$,B,C
```

If A$="I1,X1,I1", B=3, and C=9, then the output will be:
3 9

1.6.12 GOSUB <line number>

This is BASIC-180's subroutine call technique. GOSUB is a shortened form of GO to SUBroutine. When a GOSUB is encountered, program execution continues at <line number> until a RETURN is executed.

1.6.13 GOTO <line number>

GOTO transfers control to the line number given as its argument.

1.6.14 IF <rexpr> THEN

IF <rexpr> THEN <statement> or <line number>

This statement allows branches to <line number> if the relational expression <rexpr> evaluates to be true. If a statement follows the THEN, it will be executed if <rexpr> is true. If <rexpr> evaluates to be false, program execution continues at the next line in the program. For integers and real number, the relational operators are: **=, <>, ><, <, >, >=, <=, AND** and **OR**. For strings, only **=, >, <, <>,** and ><, may be used.

Note The IF statement actually tests the result of an expression. Transfer to the indicated line number or execution of the given statement takes place if the expression is nonzero. Therefore, statements of the form "IF A THEN 100" are legal.

1.6.15 INPUT <var> {,<var>...}

The INPUT statement stops the program until the user types a value for the variable <var> and ends it with a carriage return. If <var> is an integer or a real variable and a string is typed, an error message will be displayed and the request for input will be made again.

Multitasking operations are not affected by INPUT. An INPUT statement stops only the task that issues the INPUT, not any other task.

INPUT statements cannot read commas when string variables are given. The commas delimit input items to the program. By the same token, control characters cannot be read by INPUT. INPUT$ will, however, accept commas and place them in string variables.

1.6.16 INPUT$ <var> {,<var>}

INPUT$ is identical to INPUT with one exception. If a string variable is specified, INPUT$ will read whatever is entered, including commas and most control characters.

1.6.17 INTEGER <var> {,<var>,...}

INTEGER allows the user to specify that a variable or group of variables is to be handled as a 16-bit two's complement integer (useful range is -32768, to 32,767). BASIC-180 requires that all variables be declared as INTEGER, REAL, or STRING.

The INTEGER statement must occur before any executable statement in the program. Generally, all INTEGER, REAL and STRING statements are the first statements in a program.

Integer arrays are also specified using this statement. In the case of an array, the variable name must be followed by the maximum dimensions expected. Arrays may not include more than two dimensions. Array dimension 0 is always present. If an array is dimensioned as A(10), then A(0) through A(10) may be referenced.

1.6.18 NEXT <variable>

The NEXT statement terminates FOR loops. A NEXT must appear for every FOR.

1.6.19 ON <expression>, GOSUB line#, line#,...

ON GOSUB is a computed transfer statement. It causes the program to branch, via GOSUB, to a line number based on the value of the <expression>. The expression must evaluate to a number from 1 to the number of line numbers given in the command. If the expression evaluates to 1, then the program branches to the first line number; if it is 2, the program branches to the second line number; and so on. If the expression evaluates to a number larger or smaller than the number of line numbers given, then the error message "Line number does not exist" will appear.

1.6.20 ON <expression>, GOTO line#, line#,...

ON GOTO is a computed transfer statement. It causes the program to branch, via GOTO, to a line number based on the value of the <expression>. The expression must evaluate to a number from 1 to the number of line numbers given in the command. If the expression evaluates to 1, then the program branches to the first line number; if it is 2, the program branches to the second line number; and so on. If the expression evaluates to a number larger or smaller than the number of line numbers given, then the error message "Line number does not exist" will appear.

1.6.21 POKE <expr1>,<expr2>

POKE places the value of <expr2>, which must be from 0 to 255, into memory location <expr1>.

USE WITH CAUTION!

The following program places $C9 at $8700:

```
10  INTEGER K
20  POKE $8700,$C9
40  STOP
```

On HD64180 systems, POKE places the data in the segment specified by the DEFMAP statement. The physical address is formed as follows:

Physical address = (POKE address) AND 0FFF= (DEFMAP address) * 2 12

The address supplied to POKE must be less than $1000 (2**12) or BASIC-180 will AND it with $0FFF. Careful selection of a DEFMAP value will let you access anywhere in physical memory.

If DEFMAP is set to -1, no address translation takes place. The 16-bit address supplied to POKE will be used as a logical address in the HD64180's current map.

1.6.22 PRINT <expr>, {expr>...}

PRINT sends output to the currently-active device or file based on the setting of the last FILE statement. The console is the default device.

Commas or semicolons may separate print items. Commas cause each item to be separated into columns, while semicolons cause the items to be run together. If the last thing on the PRINT line is a semicolon, no carriage return or line feed will be printed.

A question mark (?) is equivalent to using PRINT.

1.6.23 PRIORITY <priority number>

The priority statement assigns a priority level to a task. It is used to alter the manner in which tasks are executed. It allows the user to assign a priority, or degree of importance, to a task. PRIORITY is useful only in multitasking programs. The PRIORITY statement must be issued from within the task whose priority level is to be altered.

PRIORITY takes one argument, which is the relative priority level for that task. The larger the number, the more important the task is. These numbers may range from 0 to 127. Therefore, 0 is least important and 127 is most important. Note that BASIC-180 assumes all tasks run at priority level 0 if no PRIORITY statement is issued.

In a multitasking program with no PRIORITY statements, each task is executed in a "round-robin" fashion. Upon receipt of a tic, BASIC-180 stops running the current task and starts running the next sequential task if it is ready to be run.

PRIORITY can be used to alter this sequence.

BASIC-180 re-evaluates the priorities on each tic, so tasks can dynamically change their level and the compiler will alter the scheduling as demanded.

Whenever a CANCEL statement is encountered, the priority level for that task is dropped to 0. This yields faster context switching.

If the task with highest priority level does not EXIT or WAIT, then it will use all of the computer time, effectively disabling multitasking, until it EXITs, goes into WAIT, or reduces its priority level.

1.6.24 RANDOMIZE

The RANDOMIZER statement reseeds the random number generator. It extracts this seed from the HD64180 refresh register. To make sure the numbers generated by RND are truly pseudo-random, it is best to use this statement before RND is used.

1.6.25 READ <variable list>

READ loads the variables in its argument list with values from a DATA statement. As successive READs are executed, data is taken from each DATA statement in the program.

1.6.26 REAL

REAL allows the user to specify that a variable or group of variables is to be handled as a 4-byte floating-point number (one with a decimal point). BASIC-180 requires that all variables be declared as INTEGER, REAL, or STRING.

The REAL statement must occur before any executable statement in the program. Generally, all INTEGER, REAL, and STRING statements are the first statements in a program.

Real arrays may also be specified using this statement. For arrays, the variable name must be followed by the maximum dimensions expected. Arrays may not include more than two dimensions. Array dimension 0 is always present. If an array is dimensioned as A(10), then A(0) through A(10) may be referenced.

9.6.27. REM <text> or ! <text> or ' <text>

REM or ! or ' precede comments. The ! or ' can be anywhere on a line as a statement separator but REM must be used at the beginning of a line or after a ":". Nothing is done with comments during compilation; they are for the user's benefit.

```
10  REM This is a remark.

20  ! This is another remark.

30  ' This is also a remark.

40  INTEGER X,Y :REM this is also a remark
```

1.6.27 RETURN

RETURN ends a subroutine started by a GOSUB. After the RETURN, execution of the program continues at the line number following the GOSUB statement.

1.6.28 RUN <task number>,<schedule interval>

RUN, not to be confused with the direct command of the same name, is used within a multitasking program to start execution of a task. The specified task in <task number> starts execution one tic after the RUN statement is executed. If the specified task EXITs, then it is automatically restarted after the number of tics in <schedule interval> has elapsed. This provides a convenient method of making something happen periodically. The schedule interval must be in the range of 1 to 32,767.

For example:

```
10  RUN 1,100

30  GOTO 30

40  TASK 1

50  PRINT "Task running"

60  EXIT
```

In the above example, task 1 is started by line 20. Since task 1 EXITs (line 60), the schedule interval on line 20 causes task 1 to restart every time 100 tics elapse after task 1 EXITs.

In the following example, the schedule interval has no effect since task 1 never EXITs.

```
20  RUN 1,20

30  GOTO 30

40  TASK 1

50  PRINT "TASK": GOTO 50
```

1.6.29 STOP

This statement stops execution of the program.

1.6.30 STRING <var>{,<var>...}

STRING allows the user to specify that a variable or group of variables is to be handled as a string. BASIC-180 requires that all variables be declared as INTEGER, REAL, or STRING.

The STRING statement must occur before any executable statement in the program. Generally, all INTEGER, REAL, and STRING statements are the first statements in a program.

Variables in the STRING statement may have a maximum string length specified by enclosing the maximum length in parentheses. If no maximum is given, a maximum length of 20 characters is assumed. No more than 127 may be specified.

A string array may be specified by giving two parameters after the string's name. The first is the length of each string element. The second argument is the number of elements in the array. For example:

```
10   STRING A$(10,20)
```

defines a string array A$ containing 20 strings, each of length 10.

1.6.31 TASK <task number>

This statement marks the beginning of a task. All tasks, other than the lead task (main program), must begin with a TASK statement. The TASK statement must be on a line by itself. The task number is a unique digit from 1 to 31 used to identify the task.

The task numbers in a program must include all numbers between 1 and the highest task used. For example, it is OK to specify tasks 1, 2, and 3, but specifying tasks 2, 4, and 5 will not work; tasks 1 and 3 must be used. If any gaps exist, tasks numbered after the gap will never be executed. In the previous example, tasks 4 and 5 would never be executed.

A task must end with an EXIT statement. A task may branch through the use of GOTO statements but there must be an EXIT or the task will never relinquish control back to the main program.

1.6.32 SYSTEM <mem address>,<var array>

A variable, usually named UTIL, should be defined and set to $7520 for use as the "mem address" for the following utility functions.

The "var array" is an array of integers with at least 4 members. Element 0 of this array is entered as the "var array". The first four elements of this array are used to pass values to and from the UTILITY functions.

Assume that the integer array is defined as: INTEGER R(4).

R(0) would be set to the utility function desired. R(1) through R(3) would contain the values input to and/or output from the UTIL function.

1.6.32.1 Utility function types

0 = Convert upper half of integer (R(1)) to HEX ASCII
and swap upper and lower HEX bytes of integer.

RETURNS:

R(2) = HI Nibble ASCII HEX

R(3) = LO Nibble ASCII HEX

256 = Convert byte (LO R(1)) to ASCII decimal (R(1)-R(3))

512 = Convert internal packed time (R(1)) to HH,MM,SS

RETURNS:

R(1). =secs

R(2) =minutes

R(3) =hours

768 = Get station number in R(1)

1024 = Get current time

RETURNS:

R(1) =secs (in hundredths)

R(2). =minutes

R(3) =hours

1280 = Get current date in DD, MM, YY

RETURNS:

R(1) = Day

R(2). = Month

R(3) = YR

Figure.0-11
Utility Function Types

Example: Get station number.

```
R(0)= 768: SYSTEM UTIL, R(0): STA=R(1)
```

Example: Get start time and stop time from line 1 and print results.

```
RLOAD
10 INTEGER HOUR, MIN, SEC, COL, TIMEON, TIMEOFF
20 INTEGER UTIL, TIME(4), PKTIME, ROW
50 UTIL=$7520: COL=$100
55 TIMEON=3: TIMEOFF=4: ROW=1
100 PKTIME = TBLRD(COL*TIMEON + ROW)
110 PRINT "Start Time"
120 GOSUB 200
130 PKTIME = TBLRD(COL*TIMEOFF + ROW)
140 PRINT "Stop Time"
150 GOSUB 200
190 STOP
200 TIME(0) = 512: TIME(1) = PKTIME
210 SYSTEM UTIL, TIME(0)
```

```
220 PRINT "Hour: ";TIME(3),"Minute: ";TIME(2),"Second ";TIME(1)
230 RETURN

END
>run

COMPILED
Start Time
Hour: 9       Minute: 40    Second 10
Stop Time
Hour: 10      Minute: 1     Second 38
>
```

1.6.33 TBLWRT <var1>,<var2>

Allows access to Real Time System Data Tables. (Also see function TBLRD in section 9.7.23)

The first variable (var1) is the destination register in Table 1 or Table 2. See following figures for information on determining the address. The second variable (var 2) is the value to be written to the destination.

Note: If the destination is in Table 1 and is designated as a digital point then only the lower byte will be written to the destination.

To facilitate the definition of the first parameter above, it is suggested that you incorporate the following variables into your program and calculate “var1” as follows:
(Note: naming is by convention only)

```
90  INTEGER V1                          'VAR1
100 INTEGER TB1,SMY,CLM,PT       'Table1, Summary, Column, Point
200 INTEGER TB2A,TB2L,LNE,RGR 'Table2Absol,Table2LineRel,Lin,Regst
.....
300 TB1=0:SMY=$6000:CLM=$100
400 TB2A=$E000:TB2L=$8000:LNE=$100
.....
1000 'FOR TABLE 1 WRITE TO POINT 50.------------------------------
1010 PT=50
1020 V1= TB1+(n*CLM)+PT  'WHERE n IS THE DESIRED PARAM ID
                                                '(SEE FIGURE 9-13)
1030 TBLWRT V1,X                       'WHERE X IS THE QUANTITY TO WRITE
.....
2000 'FOR TABLE 2 ABSOLUTE REGISTER WRITE TO REG 300-------------
2010 RGR=300
2020 V1=TB2A+RGR
```

```
2030 TBLWRT V1,X                    'WHERE X IS THE QUANTITY TO WRITE
.....
3000 'FOR TABLE 2 LINE RELATIVE REGISTER WRITE TO ----------
3001 '          LINE 15, REGISTER 10
3010 RGR=10:LNE=15
3020 V1=TB2L+(n*LNE)+RGR
3030 TBLWRT V1,X                    'WHERE X IS THE QUANTITY TO WRITE
```

Note: Since TB1=0, and the param ID for CurVal=0, writing to the CurVal column of Table 1 simplifies to V1=PT.

1.6.33.1 Calculating the address "WORD".

Choosing Table 1 or Table 2

BIT 15=1 TABLE 2 (Bit 15=$8000)

BIT 14 = 0 and 13 = 0 - Station/Register Data

BITS 8 - 12 = Station index (Line number) (Line * $100)

BITS 0 - 7 = Register (offset from first data column "L")

BIT 14 = 1 and 13 = 0 - Index parameters (Bits 15,14=$C000)

BITS 8-12 = offset into table for this index (line)

BITS 0 - 7 = Index (line) number

BIT 14 =1 and 13 = 1 - Absolute register 0 - 899 (Bits 15,14,13=$E000)

BITS 0 - 12= Register number to read

BIT 15=0 TABLE 1

BITS 13 = 0 and 14= 0 - Current Day

BITS 13 = 1 and 14= 1 - Yesterday (Bits 14,13 =$6000)

BITS 8 - 12 - Parameter ID to read (Column) (ID*$100)

(see next table for Parameter ID values)

BITS 0 - 7 Index (line) number

Figure.0-12
Register Table Addressing

Calculating Parameter ID values for Table 1

Parameter ID values (bits 8 - 12) to access the following columns/data:

ID value	Column	Description (digital/analog)	
0	C	CURSTATE/CURVALUE	
1	G	CYCLES/MAXVAL	
2	H	DURATION/MINVAL	
3	I/J	LAST ON/TimeOfMx	(use SYSTEM to convert)
4	J/K	LAST OFF/TimeOfMn	(use SYSTEM to convert)
5	Q/O	TYPE & ModulSTS	(same for digital & analog)

To decode results see tables on next page for

PT TYPE values & STATUS values

6	L	NU/HI LIMIT	
7	M	NU/LO LIMIT	
8	U	BIT POS/RANGE FS	(BIT POS not available)
9	V	NU/OFFSET	
10	X	Link Pt./N.U.	
11	W/Z	Resol/UNITS	(same for digital & analog)
	To decode results see tables on next page for		
	Resol/UNITS		
12	A	PT.NAME	(returns characters 1 & 2 of name)
13	A	PT.NAME	(returns characters 3 & 4 of name)
14	A	PT.NAME	(returns characters 5 & 6 of name)
15	A	PT.NAME	(returns characters 7 & 8 of name)
		(See 6.2 for full description of field names)	

Figure.0-13
Parameter ID values

Decoding Table 1 PT TYPE values

Bits 11 - 8 = PT TYPE

0 = DI

1 = DIG OUT

2 = ANL IN

3 = ANL OUT

4 = TOTALIZE

5 = AVERAGE

6 = PWI 3-15

7 = AI w/DBA

Figure.0-14
PT TYPE values

Decoding Table 1 STATUS values

Bits 7 - 0 = PT TYPE

BIT 7 = New Alarm Flag

BIT 6 = Alarm/Ack

BIT 5 = HI/LOW

BIT 4 = Alarm

BIT 3 = N.U.

BIT 2 = Comm Status

BIT 1 = N.U.

BIT 0 = N.U.

Figure.0-15
STATUS values

Decoding Table 1 Resol/UNITS values

BITS 7 - 6 = Decimal place

BITS 5 - 0= UNITS, with following values

0	= MV	12 = PSI	24 = InH2O
1	= MA	13 = GAL.	25 = MB
2	= VOLT	14 = KGal	26 = RPM
3	= AMP	15 = MGal	27 = %

4	= KV	16 = GPM	28 = x10
5	= KW	17 = MGD	29 = x100
6	= KVAR	18 = SEC	30 = x1k
7	= KWHR	19 = DEG	
8	= WATT	20 = DEG F.	
9	= N.U.	21 = DEG C.	
10	= FEET	22 = R HUM.	
11	= INCH	23 = WndSp	

Figure.0-16
Resol/UNITS

Example: Write ten (10) Table 1 Points (Points 20 to 29), the values will be 0 to 9.

```
100 INTEGER I,P
110 FOR I= 0 TO 9
120 P= I+20
130 TBLWRT P,I
140 NEXT I
```

Example: Write value 23 to Table 1, Point 20, High Limit

```
100 INTEGER X, HL, PT
110 HL= $100*6 '6 is high limit column
120 PT= 20 ' point 20
130 X=23
140 TBLWRT HL+PT,X
```

TRACE ON and TRACE OFF

TRACE ON causes the line number of each line to print as it is executed. TRACE OFF disables TRACE ON.

Note that the line numbers will be sent to the selected file (as specified by a FILE statement), whether it is console, or printer.

Example:

```
10 INTEGER K
20 FOR K=1 TO 20
30 IF K>15 THEN TRACE ON
40 NEXT K
50 TRACE OFF
```

1.6.34 WAIT <expr>

The WAIT statement delays the current task from executing until <expr> tics has elapsed. This is the best way to have a delay in a multitasking program because it requires no computer time. By contrast, a FOR/NEXT-loop delay uses computer time (best used by other tasks) and does nothing useful.

If all tasks in a program are in a WAIT simultaneously, then <Ctrl-C> will not kill the program until one of the tasks finishes its WAIT.

1.6.35 WCLEAR

WCLEAR erases the currently-selected window. The entire window is erased, so if a window is created, framed (with WFRAME), then cleared, the frame will also be removed unless the window is resized after being framed.

This example creates a window, frames it, and clears it.

```
10 WSELECT 1
20 WINDOW 0,0,10,10
30 WFRAME "_", "|"
40 WAIT 500
```

```
50 WCLEAR
```

1.6.36 WFRAME <horizontal character>,<vertical character>

WFRAME draws a frame around a window. A frame is simply an outline to give a clear depiction of the window's borders.

The two arguments specify the characters which will outline the window. BASIC-180 doesn't support true graphics since graphics controllers are different on all systems, so these two characters simulate a graphics box around the window. The first argument, <horizontal character>, is used to draw the upper and lower window borders. The vertical character is used to draw the left and right window borders. The preferred characters are an underscore ("_") for the horizontal character and the vertical line ("|") for the vertical character. Of course, any other character may be used.

WFRAME draws the frame inside of the window, so it actually occupies space in the window. It is often a good idea to frame a window, but have the frame outside of the borders of the window so that output directed to the window will not run into the border and so the frame won't be erased by a WCLEAR. This is easy to do. Define a window which is one character larger in all directions than the required window. Frame it, then redefine the window to the needed size. For example, the following program generates a window, frame, and a smaller window inside the frame. The usable window size is 10 x 10 characters.

```
10 WSELECT 1
20 WINDOW 10,10,20,20          ! Draw a window 1 character too large.
30 WFRAME "_", "|"             ! Frame it.
40 WINDOW 11,11,19,19          ! This defines the actual window.
50 PRINT "*"                   ! Put something in the window.
```

1.6.37 WINDOW

WINDOW <UL row>,<UL column>,<LR row>,<LR column>

The WINDOW statement defines the size of a window. A window may not be used until it is both selected (using WSELECT) and defined (using WINDOW). The minimum size for a window is 2 by 2 characters.

The first two arguments define the upper-left (UL) row and column of the window, while the second two arguments define the lower-right (LR) row and column. A window is completely specified by designating its upper-left and lower-right corners. BASIC-180 counts rows from 0 (top of the screen) to 23 (bottom of the screen) and columns from 0 (left-hand side of the screen) to 79 (right-hand side of the screen). For example, "WINDOW 0,0,23,79" defines a window which is the size of the entire screen. If the user attempts to define a nonsense window (for instance, LR row less than UL row). BASIC-180 will set a default window size.

Once a window is defined and selected, all console output will go to that window until another window is selected. BASIC-180 will prevent access to any part of the screen outside of the borders of the window.

Example:

```
10 WSELECT 0                   !Always select a window first
20 WINDOW 5,20,10,40
30 PRINT "HELLO"
```

1.6.38 WPOKE <expr1>,<expr2>

WPOKE acts like a 16-bit POKE. It places the value of <expr2>, which must be from 0 to 65535, into memory location <expr1>.

USE WITH CAUTION!

The following program places 1234 at $8700.

```
10 INTEGER K

20 WPOKE $8700,1234

30 STOP
```

On HD64180 systems, WPOKE places the data in the segment specified by the DEFMAP statement. The physical address is formed as follows:

Physical address = (WPOKE address) AND 0FFF+ (DEFMAP address) * 2 12

The address supplied to WPOKE must be less than $1000 (2 12) or BASIC-180 will AND it with $0FFF. Careful selection of a DEFMAP value will let you access any location in physical memory.

If DEFMAP is set to -1, no address translation takes place. The complete 16-bit address supplied to WPOKE will be used as a logical address in the HD64180's current map.

1.6.39 WSAVE <integer array>

WSAVE is used to save the contents of the currently-selected window to an integer variable array. This feature, coupled with WUPDATE, which restores a window from an array, allows the user to generate pop-up menus and save windows to disk.

WSAVE requires an integer array as an argument. Strings may not be used. In BASIC-180, arrays must be specified with an argument, so generally it is best to specify the array with a subscript of 1 (e.g. T(1)).

WSAVE saves all printable characters found in the specified window to the array. The first word (2 bytes) will contain the number of characters saved. The characters will then be packed two to a word into the array. A carriage return will be saved after the rightmost character in each line. Remember, spaces are characters, too. The array should, therefore, be dimensioned (using the INTEGER statement) to: (the number of expected characters divided by 2) + 1 + the number of rows.

CAUTION: BASIC-180, in the interest of speed, does not check for subscript overflow while filling the array during WSAVE, so it is a good idea to make the array a little on the large size. An array size of 1000 will permit saving an entire 80x24 screen.

Example:

```
10 INTEGER T(1000)

20 WSELECT 1

30 WINDOW 10,10,20,20

40 WFRAME "_", "|"

50 WAIT 500

60 WSAVE T(1)

70 ERASE

80 WUPDATE T(1)

90 WAIT 500
```

1.6.40 WSELECT <window number>

WSELECT is used to enable the windowing system and to specify which of up to 10 windows is to be used. The argument is an integer from 0 to 9 which specifies the window number. If this argument is more than 9 then windows will be turned off for that task. To disable windowing altogether, WSELECT a <window number> greater than 10.

WSELECT must be issued before any other windowing statements are executed. If a selection is not made, BASIC-180 will ignore the windowing statements.

```
10 WSELECT 0                ! Always select a window first.

20 WINDOW 5,20,10,40

30 PRINT "HELLO"
```

1.6.41 WUPDATE <integer array>

WUPDATE is the converse of WSAVE. It redraws the entire window and its contents from <integer array>, which must have been saved using a WSAVE statement.

As in WSAVE, the argument must be an integer array, and should be specified as, for example, T(1).

```
10 INTEGER T(1000)

20 WSELECT 1

30 WINDOW 10,10,20,20

40 WFRAME "_", "|"

50 WAIT 500

60 WSAVE T(1)

70 ERASE

80 WUPDATE T(1)

90 WAIT 500
```

1.7 Functions and User-Defined Functions

Functions are called by referencing them in an expression. In BASIC-180, each function returns an INTEGER, REAL, or STRING argument. These are the BASIC-180 functions.

ACOS	COS	RND
ASC	EXP	SIN
ASIN	GET	SQR
ATAN	HST	STR$
BAND	KEY	TAN
BOR	LEN	TBLRD
BXOR	LOG	VAL
CHR$	MID$	WPEEK
CONCAT$		PEEK

Figure.0-17
Functions

1.7.1 ACOS(<expr>)

REAL
Calculates the arccosine of <expr>, and returns this value in degrees.

1.7.2 ASC(<sexpr>)

INTEGER
Returns the ASCII equivalent of the first character in the string <sexpr>. ASC is the converse of CHR$.

1.7.3 ASIN(<expr>)

REAL
Calculates the arcsine of <expr>, which is returned in degrees.

1.7.4 ATAN(<expr>)

REAL
Calculates the arctangent of <expr> and returns this value in degrees.

1.7.5 BAND(<expr1>,<expr2>)

INTEGER
Logically ANDs <expr1> and <expr2> as 16-bit integers. A bitwise AND is performed. Each of the 16 bits is individually ANDed.

1.7.6 BOR(<expr1>,<expr2>)

INTEGER
Logically ORs <expr1> and <expr2> as 16-bit integers. A bitwise OR is done. Each bit is individually ORed.

1.7.7 BXOR(<expr1>,<expr2>)

INTEGER
Logical XORs <expr1> and <expr2> as 16-bit integers. A bitwise XOR is done.

1.7.8 CHR$(<expr>)

STRING
Returns a one-character equivalent to the integer <expr>. <Expr> must be between 1 and 255. CHR$ is the converse of ASC.

1.7.9 CONCAT$(<sexpr1>,<sexpr2>)

STRING
Concatenates two strings into one string; <sexpr2> is appended after <sexpr1>.

1.7.10 COS(<expr>)

REAL
Calculates the cosine of <expr>, which must be in degrees.

1.7.11 EXP(<expr>)

REAL
Computes e<expr>. A number can be raised to another power (i.e., Y^X) by using: EXP(LOG(X)*Y).

1.7.12 GET

INTEGER
Returns one character from the current file. INPUT and INPUT$ read an ASCII line terminated by a carriage return, so they can't read binary files. Since GET only reads single characters, it can read binary files. (See the FILE statement for details of ‘GET’s from File 6 or 8)

1.7.13 HST(<expr>)

UNSIGNED INTEGER
HST provides access to Historical data records. When <expr> is a number less than the maximum possible sample number (e.g., 61648), it specifies the sample record number to retrieve. Each sample record consists several values returned in 4 word sized values. Each request for a given sample record will return the next word of the sample. The formats of the four words is shown below. If <expr>

changes in value (i.e., indicates a new sample record is being accessed), the first word of the new sample record will be returned. Subsequent accesses to that same sample record will return the second, third and fourth words in order.

If <expr> equals -1, the function will return the index number

If <expr> equals -2, the function will return the index of the top sample (latest or most recently recorded sample).

If <expr> equals -3, the function will return the index of the end sample (oldest recorded sample or end of memory).

If <expr> equals -4, the function will return the word number (0-3) which was returned on the last access to the current sample.

If <expr> equals -5, the function will return the maximum sample index (indicates memory size).

If <expr> equals -6, the function will decrement the sample index and return the same word (e.g., contents of the current word number) from the previous sample. This allows easy searching of samples for matching criteria.

Example:

```
100 S= HST(-2)      ;GET INDEX OF LATEST SAMPLE
110 V(1)= HST(S)    ;GET 1ST WORD OF SAMPLE
120 V(2)= HST(S)    ;GET 2ND WORD OF SAMPLE
130 V(3)= HST(S)    ;GET 3RD WORD OF SAMPLE
140 V(4)= HST(S)    ;GET 4TH WORD OF SAMPLE
.
.
160 S=S-1
170 V(1)= HST(S)    ;GET 1ST WORD OF PREVIOUS SAMPLE
```

	HI BYTE (8 BITS)	/	LO BYTE (8 BITS)
1st word-	POINT	/	STATION
2nd word-	DP+DAY	/	MONTH+TYPE
3rd word-	TIME STAMP	(HR, MIN, SEC)	
4th word	DATA VALUE		

Figure0-18
HST WORD Formats

DP field	bits 7-6	=	DP
DAY field	bits 0-5	=	DAY
TIME STAMP	bits 15 - 11	=	HR
	bits 10 - 5	=	MIN
	bits 4 - 0	=	SEC/2
TYPE Field:	bit 7	=	TotalVal flag

bit 6 = Dig/Anl flag,

bit 5 = CurVal/MaxMin flag

bit 4 = Max/Min flag.

bits 3-0 = month.

Figure.0-19
HST WORD Field Formats

1.7.14 KEY

INTEGER
Returns the ASCII value of the character the keyboard currently has ready. KEY returns a 0 if no character is ready. Note that the console is always read regardless of which file number is active. KEY returns an integer representation of the character (i.e., its ASCII value). If a string is needed, convert it using CHR$. (see GET for character reads from file devices)

KEY reads the current character from the keyboard and resets the console's UART. Therefore, the following construct will not work since when the second KEY is executed, the character is already gone (having been read by the first KEY):

```
10 IF KEY=0 THEN 10

20 PRINT CHR$(KEY)          ;This program won't work correctly.

30 GOTO 10
```

Instead, set a variable equal to KEY and test and print the variable, as in the following example.

```
10 INTEGER L

20 L=KEY

30 IF L=0 THEN 20

40 PRINT CHR$(L)

50 GOTO 20
```

1.7.15 LEN(<sexpr>)

INTEGER
Returns the length of the string argument <sexpr>.

1.7.16 LOG(<expr>)

REAL
Calculates the natural logarithm (base e) of <expr>.

Note: to calculate LOG base 10 of a REAL number use the following function:

```
10 REAL LG10,N

20 REM This function returns the log base 10 of (N)

30 DEF LG10(N)

40 LG10=LOG(N)/LOG(10.)

50 FNEND
```

In general, the LOG base B of a number N can be calculated by the expression "LOG(N)/LOG(B)" where B and N are both real numbers.

1.7.17 (<sexpr>,<expr1>,<expr2>)

STRING
Returns <expr2> characters of <sexpr> starting at character <expr1>.

1.7.18 PEEK(<expr>)

INTEGER
Returns the 8-bit contents of memory address <expr>.

```
10 POKE $80,1

20 PRINT PEEK($80)
```

On HD64180 systems, PEEK places the data in the segment specified by the DEFMAP statement. The physical address is formed by:

Physical address = (PEEK address) + (DEFMAP address) * 2 12

The addresses supplied to PEEK must be less than $1000 (2 12) or BASIC-180 will AND it with $0FFF. Careful selection of a DEFMAP value will let you access anywhere in physical memory.

If DEFMAP is set to -1, no address translation takes place. The 16-bit address supplied to POKE will be used as a logical address in the HD64180's current map.

1.7.19 RND

INTEGER
Generates a pseudorandom number between -32,767 and 32, 767. It's a good idea to reseed the random number generator at the start of your program by executing the RANDOMIZE statement.

A sample program to generate a random number between a high and low limit (the low limit can be a negative value):

```
RLOAD


10   INTEGER KBD,HILIM,LOLIM,ANSWER,RANGE

15   REAL RNUM

100 KBD=KEY: 'look for keyboard input

110 IF  KBD = 3 THEN  STOP : 'if CTRL C then quit

120 HILIM=TBLRD(1): 'read in high limit, table 1, row 1

130 LOLIM=TBLRD(2): 'read in low limit, table 1, row 2

140 RANGE=HILIM - LOLIM

150 RANDOMIZE : 'seed random number generator

160 RNUM=BAND($7FFF,RND) / 32767: 'normalize, positive only

170 ANSWER=RNUM * RANGE + LOLIM

180 TBLWRT 3,ANSWER: 'write random number, table1, row 3

181 'PRINT ANSWER,RNUM: 'for testing

200 GOTO 100


END
```

1.7.20 SIN(<expr>)

REAL
Calculates the sine of <expr>, which must be in degrees.

1.7.21 SQR(<expr>)

REAL
Calculates the square root of <expr>.

1.7.22 STR$(<expr>)

STRING
Converts the number given as its argument, <expr>, to a string. STR$ is the converse of VAL.

1.7.23 TBLRD(<expr>)

INTEGER
Allows access to Real Time System Data Tables. (Also see procedure TBLWRT in section 9.6.33)

The variable <expr> is the source register (or address) in Table 1 or Table 2.

See 9.6.33.1 for Calculating the BASIC address "WORD"

Note: If the source address is in Table 1, and it is designated as a digital point, then only the lower byte will be returned.

Example: Read Current Value of seven (7) Table 1 Points from 20 to 26

```
10 INTEGER I, V(6)
20 FOR I= 0 TO 6
30 V(I)=TBLRD(I+20)
40 PRINT V(I) 'for debug
50 NEXT I
```

Example: Read Table 2, Line 3, Reg 2 into variable V

```
100 INTEGER T2,LX,ROW,REG,V
110 T2=$8000 : LX=$100
120 ROW=3 * LX
130 REG=2
140 V=TBLRD(T2 + ROW + REG)
150 PRINT V
```

Example: Read Absolute register 2, (Table 2) into variable X

```
100 INTEGER T2ABS,REG,X
110 T2ABS=$E000 : REG=2
```

```
120 X = TBLRD(T2ABS + REG)

130 PRINT X
```

1.7.24 TAN(<expr>)

REAL
Calculates the tangent of <expr>, which must be in degrees.

1.7.25 VAL(<sexpr>)

REAL
Returns the numeric value of the number at the beginning of the string expression given as VAL's argument. VAL is the converse of STR$.

1.7.26 WPEEK(<expr>)

INTEGER
Returns the 16-bit contents of memory address <expr>.

Chapter 9 – BASIC INDEX