# User's Guide
# Contents



### Other Manuals



{button www.htbasic.com,Inet(`www.htbasic.com')}


Distributed with release 9.3

# Chapter 1
# Language Elements

# Chapter 2
# Program Flow Control

# Chapter 3
# Mathematics

# Chapter 4
# Graphics

# Chapter 5
# General Input and Output

# Chapter 6
# CRT, Keyboard, and Printer

# Chapter 7
# Files

# Chapter 8
# IEEE-488 Interface Bus

Summary

# Chapter 9
# Serial (RS-232) I/O

# Chapter 10
# Other I/O Destinations/Sources

# Chapter 11
# DLL Toolkit

# Chapter 12
# International Language Support

# Chapter 13
# Condensed Keyword Reference

# Language Elements

**High Tech Basic** (HTBasic) is a highly configurable, technical programming language. Before HTBasic is used, the necessary drivers must be loaded and some customization may be required. Understanding the basic elements of the language like program lines, labels, keywords, contexts, I/O paths, constants, variables, arrays and data types is necessary to fully exploit the functionality of the product.

# This Manual

This manual, the *User's Guide*, contains in depth information about using the HTBasic language. It is arranged topically. The following list describes each chapter of the manual.

**Chapter 1**, "Language Elements," explains the basic elements of the language like keywords, operators, and variables.

**Chapter 2**, "Program Flow Control," explains how the path of program execution can be controlled to perform calls, loops, and decisions.

**Chapter 3**, "Mathematics," describes numeric expressions, string expressions, and array expressions. The various operators and functions are presented.

**Chapter 4**, "Graphics," presents the graphic drawing and presentation capabilities of HTBasic.

**Chapter 5**, "General Input and Output," explains Input and Output (I/O) in general. This chapter also explains the TRANSFER statement. The information in this chapter is necessary to understand chapters 8 to 12 and should be read by all users.

**Chapter 6**, "CRT, Keyboard, and Printer," describes I/O to the screen (CRT) and keyboard (KBD). Several statements are also presented that are designed for I/O with either the screen or a printer.

**Chapter 7**, "Files," presents I/O to files and the special file management statements available. These special statements often make it unnecessary to use operating system commands to move, copy, delete, or otherwise manipulate files.

**Chapter 8,** "IEEE-488 Interface Bus," explains use of the IEEE-488 (GPIB or HP-IB) interface.

**Chapter 9**, "Serial (RS-232) I/O," describes use of the Serial (RS-232) interface.

**Chapter 10**, "Other I/O Destinations," talks about I/O with a variety of I/O targets: Parallel (Centronix Printer) Interfaces, buffers, Strings and a special interface called the "Processor Interface". This chapter also describes methods for using interfaces and devices for which there is no HTBasic device driver.

**Chapter 11**, "DLL Toolkit," explains Dynamic Link Library use with HTBasic.

**Chapter 12**, "International Language Support," explains the support included in HTBasic for different languages and character sets.

**Glossary**, defines terms which may be unfamiliar to the user.

# Program Lines

An HTBasic program is a series of instructions. These instructions are in the form of program lines. The general form of a program line is:

```
[line_number] [ label: ] statement [ !comment ]
```

# Line Numbers

Each HTBasic program line requires a unique line number at the beginning of the line. Line numbers must range from 1 to 4,194,304. Optionally, line numbers may be toggled to display or not. Line numbers are used to:

- Indicate the order of statement execution
- Provide control points for branching
- Help in debugging and updating programs
- Indicate the location of run-time errors

The maximum number of characters allowed in a program line is 255. It may be necessary to scroll to see the end of a line.

# Line Labels

Line labels may optionally follow any line number. The use of line labels results in more structured programming. If all line references use line labels, the line numbers can effectively be ignored. Line references to labels are unaffected by line numbering. A line label cannot be the only thing on a line; you must include a statement, even if it is just a comment tail "!".

# Statements

A program statement is the smallest complete program unit. It is made up of HTBasic keywords, operators and operands. The on-line *Reference Manual* details the syntax for each program statement.

# Comments

An optional comment may be added to any program statement by starting the comment with an exclamation mark "!". The characters following the exclamation mark will be retained in the program as a comment and are ignored during the execution of the program. Comments are not moved by the INDENT statement.

# Keywords

Certain words have a special meaning in the HTBasic language and are called keywords. For example, PRINT is a BASIC keyword. Keywords can be used as variable names if they are entered partly in uppercase and partly in lowercase. Every statement except an assignment (if LET is omitted) or a subprogram CALL (if CALL is omitted) must begin with a keyword. A keyword cannot have embedded spaces. Keywords are printed in uppercase in program listings.

Keywords must be delimited by a space or some other legal delimiter and cannot be abbreviated. For example, PRINT USING cannot be entered as PRINTUSING. Neither can PRINT be entered as PRI.

# INTEGER Numbers

INTEGER is a numeric data type. INTEGERs are whole numbers (-1, 2, 35) ranging from -32,768 to +32,767. INTEGERs are stored in two bytes of memory and INTEGER operations are faster than REAL operations.

# LONG Numbers

LONG is a second numeric data type. Longs are actually "long integers" and are essentially identical to the integer data type, except that they have a range of -2,147,483,648 to 2,147,483,647. LONGs are stored in four bytes of memory.

# REAL Numbers

REAL is another numeric data type. REAL numbers, also called floating point numbers, are a subset of all rational numbers. The particular subset depends on your computer. Most computers, including the IBM PC, use IEEE Std 754-1985 for Binary Floating point numbers. This gives the REAL data type an approximate range of 1E-308 to 1E+308 and 15 decimal digits of precision. Eight bytes are used to store a REAL number. Both positive and negative numbers are represented. MINREAL and MAXREAL are functions that return the smallest and largest REAL numbers.

Integral numbers too large to be represented by the INTEGER or LONG type, numbers expressed with a decimal point, and numbers expressed in scientific notation are stored as REAL numbers. Scientific notation on computers is usually expressed as x.xxxEyyy, meaning x.xxx is multiplied by 10 raised to the yyy power. For example: 3.141E0, 4E-7, -101.1E+2. REAL operations are slower than INTEGER operations and REAL numbers take more memory space to store.

# Cautions

Some cautions are in order when using REAL numbers. It is possible to have two different REAL numbers whose 15 digit decimal representations are the same. However, when comparing or subtracting these two numbers that look equal, you will find they are NOT equal. Also, since REAL numbers are only a subset of the rational numbers, some operations produce a result that is not in the set of REAL numbers. If the result is too large in magnitude, it produces an overflow error. If the result is too small, it produces an underflow error. The result may be between two numbers in the REAL set. In this case, an approximation must be used for the actual value. Most of us have witnessed this happening by dividing one by three on a hand-held calculator. Digits that the calculator can not store are just discarded.

Each of these pitfalls is demonstrated in the following program:

```
10 RAD
20 One = COS(3)*COS(3)+SIN(3)*SIN(3)
30 PRINT One,One-1.0
```

From trigonometry we know that line 20 should assign the value one to the variable One. And indeed, when One is printed, we see it is "1". However, when the value of One-1.0 is printed, we do not get zero. We get a very small value.

# REAL Comparisons

Rather than compare two REAL values for equality, it is often best to compare them for an acceptably small difference. For example, the rest of the program started above:

```
40 IF ABS(One-1.0) < 1E-15 THEN PRINT "EQUAL ENOUGH"
```

Alternately, DROUND or PROUND can be used to round the binary representations to match each other:

```
50 PRINT One,DROUND(One,1)-DROUND(1.0,1)
60 END
```

# FOR Loops

It is not a good idea to use a REAL variable as the loop counter in a FOR loop. Rounding errors tend to accumulate when a REAL variable is used as the loop counter in a FOR loop. For example:

```
10     FOR X=1 TO -1 STEP -.05
20       PRINT X;TAB(20);ACS(X)
30     NEXT X
40     END
```

It is expected that this loop would include 0 and -1 among the values printed for X. However, when this example is executed, a small non-zero value is printed in place of 0 and the loop terminates before getting to -1.

INTEGER or LONG variables should be used for the loop counter. The number of iterations can then be exactly specified. The REAL values needed in the loop can be generated each iteration from the INTEGER or LONG variable. This approach does not allow rounding errors to accumulate. The previous example should be replaced with:

```
10     INTEGER I
20     FOR I=0 TO 40
30       X=1-I*.05
40       PRINT X;TAB(20);ACS(X)
50     NEXT I
60     END
```

# Definition

The exact subset of rational numbers that can be represented by REAL numbers (on computers using IEEE-754) is the set of all numbers expressible in the form:

(-1)s * 2e * mantissa

where:
s = 0 or 1.
e = any integer between -1022 and +1023.
mantissa = b0 * 2-0 + b1 * 2-1 + ... + b52 * 2-52
b0 = 1
bi = 0 or 1 (for i<>0)

# COMPLEX Numbers

COMPLEX is yet another numeric data type. The COMPLEX data type defines a number having two components, like a two-dimensional vector. In a complex number, however, there is a well-defined relationship between the two components of the number. The first component in a complex number is called the real part and the second component is called the imaginary part. A complex number is often written in the form

x + iy

where *x* represents the real part of the number and *y* the imaginary part. *X* and *y* are ordinary REAL numbers and have the same range as REAL numbers. The *i* is defined as the square root of -1.

# STATIC

STATIC is a data scope, rather than a data type. A static variable is persistent during a single run of an HTBasic program. Typically, static variables will only be used in SUB programs and/or FN functions because the MAIN context is usually called only once.

Static variables can effectively take the place of COM variables as they are presently used in many cases. If access to a COM variable is required in multiple SUBs and/or Functions (DEF FN) and/or the Main context, then a static variable is not appropriate. The scope of a static variable is limited to the context in which it is declared. In other words, a static variable declared in a SUB program cannot be accessed anywhere other than within that particular SUB program.

# Strings

A string is another data type. A string is a combination of ASCII characters. These are the letters, numbers, and symbols that you can type on the keyboard. ASCII characters also include control characters such as carriage return, etc. A string also has a current length. The length can be zero, meaning that there are no characters stored in the string, or any size up to a maximum of 32,767.

# I/O Paths

An I/O path is yet another data type. An I/O path consists of all of the routing information necessary for the computer to exchange data between your HTBasic program and another entity (such as a printer, data acquisition device, string, file, etc.). Data is assigned to an I/O path variable with the ASSIGN statement. Unlike the other data types, PRINT can not be used to examine the contents of an I/O path variable. However, the contents are used when an OUTPUT, ENTER, or other statement specifies the I/O path variable.

# Constants

A constant is a quantity with a fixed value. There are five types of constants: REAL, INTEGER, COMPLEX, LONG and STRING. COMPLEX, REAL, INTEGER and LONG constants are collectively called numeric constants. The following are examples of numeric constants:

| Example | Type of Constant |
|---|---|
| 1 | integer constant |
| 1.0 | real constant |
| 2.718281828 | real constant |
| -20000 | integer constant |
| -2121503777 | long constant |
| +1E+0 | real constant |
| 40000 | long constant (32,767 is max integer size) |
| CMPLX(0,1) | complex constant |
| 6000000000 | real constant (2,147,483,647 is the max long size) |

# Numeric Constants

Numeric integer constants can also be expressed in octal (base 8) or hexadecimal (base 16). A hexadecimal constant must begin with the characters "&H". An octal constant must begin with the characters "&O" or simply "&". Hexadecimal and octal constants are an extension in HTBasic and are not supported in HP BASIC. The following are examples of hexadecimal and octal constants:

| Example | Type of Constant |
|---|---|
| &H10 | hexadecimal constant with decimal value of 16 |
| &O10 | octal constant with decimal value of 8 |
| &20 | octal constant with decimal value of 16 |

# String Constants

A string constant is a sequence of ASCII characters enclosed in quotation marks. They are also called string literals. A quotation mark may be included in a literal by entering two adjacent quotation mark characters. For example:

```
100 PRINT "This is a quotation mark >""<"
```

This will print:

```
This is a quotation mark >"<
```

# Variables

A variable is an entity with memory and a changeable value. Each variable has a name, a type, a scope, and a value. Array variables remember multiple values. The rules for naming variables, the variable type, the scope of a variable, and array variables are explained in the following paragraphs.

A string or numeric variable can be assigned a value that comes from a constant, another variable, an expression, or a DATA, INPUT, ENTER, or READ statement. An I/O path variable can be assigned using the ASSIGN statement. The type of data must match the variable type.

# Variable Names

A variable name can have up to 15 characters. The characters can be alphabetic, numerals, underlines, and characters ranging from CHR$(128) to CHR$(254). The first character may not be a numeral or an underline. The last character of a string variable name must be followed by the dollar sign character, "$". The first character of an I/O path variable name must be the at-sign character, "@".

A variable name can be the same as a keyword if it is entered partly in uppercase and partly in lowercase characters. Variable names are listed with the first character in uppercase and the remaining characters in lowercase. Here are some examples of legal and illegal variable names (and why):

| Variable | Explanation |
|---|---|
| Smile | legal |
| FOR | illegal, FOR is a keyword |
| For | legal with mixed case |
| I | legal |
| X1 | legal |
| X-7 | illegal, minus sign is not allowed |
| Supercalifragalisticexpealadoeshous | illegal, too long |

# Variable Types

There are six types of variables:

- Real
- Integer
- Long
- Complex
- String
- I/O Path

You must indicate the type of each variable in some way. Use the REAL statement to declare real numeric variables. Use the INTEGER statement to declare integer variables, the LONG statement to declare LONG integer variables and the COMPLEX statement to declare complex variables. If a numeric variable is not declared, it is declared REAL automatically. To turn off automatic declaration of variables, execute CONFIGURE DIM OFF.

HTBasic recognizes a string variable from the dollar sign, "$", following the last character of the string name. The DIM statement is used to set the maximum string length. The length of a string stored in a string variable cannot exceed its DIMensioned length. To dimension a string named S$ with a maximum length of 20, use the following syntax:

```
DIM S$[20]
```

If a string is not declared in a DIM statement, it is normally dimensioned automatically to a maximum string length of 18 characters. To turn off automatic declaration of variables, execute CONFIGURE DIM OFF.

An I/O path variable does not have to be declared. HTBasic recognizes it from the at-sign character, "@", preceding the variable name. The ASSIGN statement is used to set up the I/O path variable before a data transfer.

# Variable Scope

HTBasic supports variables with two different scopes, local and global.

All variables not defined in a COM statement are local in scope. The values of local variables are only accessible within the context in which they are defined. All local variables are assigned the value zero when the context begins execution. When the context finishes execution, the values of the local variables are discarded. When a context is called recursively, each invocation of the context is given its own set of local variables. The COMPLEX, REAL, INTEGER, LONG, STATIC, DIM, and ALLOCATE statements are used to declare local variables.

No memory is reserved for ALLOCATEd variables until the ALLOCATE statement is executed. The DEALLOCATE statement releases the memory before the context finishes execution. ALLOCATE is only used for array and string variables.

COM variables are global in scope. The values of COM variables are stored in COM blocks that are global in lifetime. A COM block is a set of one or more variables that is held in "common" (i.e. may be shared) among one or more contexts. Each COM block is uniquely identified with a name (although one block is allowed to be nameless). As many COM statements as necessary may be used in a context to fully describe the COM block variables.

To access COM variables, a context must include a COM statement that identifies the COM block and gives the names by which the variables will be known in that context. Thus, each context can give a different name to the same COM variable. COM variables are hidden from all contexts that do not include a COM statement accessing that COM block.

When a new program is brought into memory, the existing COM blocks are compared to the COM blocks defined in the new program. Any COM blocks that exactly match are retained and their data values are available for use by the new program. Any COM blocks that do not match are deleted and the memory used by their data values is released and may be reused by the new program.

Note that STATIC variables are similar to COM variables multiple invocations of a given context will all use the same instance of the variable.

# Array Variables

A simple variable has one data value. An array is a multi-dimensional ordered set of data values. Each member of the set is called an array element. All the members of the set have the same data type which can be COMPLEX, INTEGER, LONG, REAL, or string. An array variable can **not** have the same name as a simple variable.

The number of dimensions of the array is called the RANK. Arrays may have a RANK from one to six. You can specify both the lower and upper bound of each dimension. If the lower bound is not specified then the current OPTION BASE of the context is used as the lower bound. The default OPTION BASE is zero.

# Declaration

Local array variables are declared using the COMPLEX, INTEGER, LONG, REAL, STATIC and DIM statements. The ALLOCATE statement is used to dynamically declare a local array. The COM statement is used to declare a global array. Normally, all array variables that are not declared will be declared automatically with the default lower bound, an upper bound of 10, and a RANK matching the number of subscripts in the first reference to the array. To disable automatic array declaration, use CONFIGURE DIM OFF.

# String Arrays

A string array may be defined where each element of the array is a separate string of the dimensioned length. To dimension a string array named S$ with four elements (assuming the default OPTION BASE 0), each of which can have a maximum length of 20 characters, use the following syntax:

```
DIM S$(3)[20]
```

# Examples

A few examples follow showing array declarations.

```
DIM X(3)          ! declares an array of 4 REALs.
```

This example defines an array with elements numbered 0,1,2,3. If you have set the context OPTION BASE to 1, then it would define an array with elements numbered 1,2,3.

```
INTEGER A(50:100) ! declares an array of 51 integers.
```

This example defines an integer array with 51 elements numbered 50 through 100.

One-dimensional arrays are always referenced by one subscript in parentheses following the array variable. For example, A(2) refers to the third element. Two-dimensional arrays are referenced by two subscripts, where the first subscript refers to the row, and the second subscript refers to the column. For example, A(1,2) refers to the second row, the third column. An element of an array can be used wherever a simple variable of the same type can be used.

Once an array variable is declared, it can be assigned elements via:

- [LET] statement
- MAT statement
- INPUT statement
- ENTER statement
- READ statement.

To use the [LET] statement to assign values to array elements, you must first DIMension the array.

```
100 DIM A(1:2,1:2)
```

This statement sets the working size of A to two rows and two columns, and enough memory space is reserved to store four numeric values; one value for each element.

Next, you assign a numeric constant to each element. Each element must be assigned a value using subscripts on the array variable as follows:

```
110 A(1,1) = 5
120 A(1,2) = 6
130 A(2,1) = 7
140 A(2,2) = 8
```

The MAT statement can be used to assign an entire array. For example:

```
100 MAT A = (1)   ! all elements in A are set to 1
```

Another way to assign values to the elements of an array is to use the INPUT statement:

```
100 DIM A(1:2,1:2)
110 INPUT A(*)
```

When these statements are executed, the variable A is dimensioned to a 2 by 2 matrix. The INPUT statement then reads the keyboard entries into the elements of the array A. The elements are input in row major order. For example, the following input assigns the same values as the lines 110 to 140 above:

```
5,6,7,8
```

Each entry must be separated by a comma or by pressing the RETURN key. The input request prompt is displayed again and again until an entry is made for each element in the array.

The READ statement can also be used to assign values to the elements in an array. For example:

```
100 DIM A(1:2,1:2)
110 DATA 5,6,7,8
120 READ A(*)
```

When these statements are executed, the numeric constants in the DATA statement are assigned to array A in row major order.

You can also use a FOR/NEXT loop to assign values in some other order or starting point. For example:

```
100 DIM Beta(1:99)
110 FOR J = 40 TO 50
120 ENTER @Path;Beta(J)
130 NEXT J
```

# Summary

This chapter discussed the HTBasic Language Elements. It summarized the concepts of program lines, program statements, keywords, contexts, COMPLEX, INTEGER, LONG, STATIC and REAL numbers, strings, I/O paths, constants, variable names, variable types, variable scopes, and array variables. More detailed information about each of the mentioned program statements is contained in the on-line *Reference Manual*.

# Program Flow Control

This chapter describes the program execution states, conditional branching statements, looping statements, subroutines, subprogram contexts, user defined functions, event and error handling statements, and the EXECUTE statement.

# Program Execution

An HTBasic program is started by clicking the RUN|RUN Menu, the RUN icon on the Control Toolbar, by pressing the RUN key or entering the RUN statement directly from the keyboard. A prerun pass is made over each context before the program begins execution. The program then executes normally until it encounters an END, PAUSE, or STOP statement. While running, a message is displayed on the Status Bar to indicate that a program is running.

The WAIT statement can be used to temporarily suspend the execution of the next program line for a specified number of seconds.

# The Paused State

Program execution can be halted by selecting the RUN|PAUSE menu, the PAUSE icon on the Control Toolbar, by pressing the PAUSE key, executing a PAUSE statement in the program, or entering the PAUSE statement. While PAUSEd, the values of variables in the current context can be printed or changed and the program can be examined with the LIST or EDIT statements. The program can be restarted by clicking the RUN|CONTINUE menu, the CONTINUE icon on the Control Toolbar, by pressing the CONTINUE key, or entering the CONT statement. A program error will also PAUSE the program as will the CLR I/O key if pressed during an I/O statement. Changing a program line while PAUSEd, will change the state of the computer to a stopped state.

# The Stopped State

The program can be forced to halt at some point other than the END statement by selecting the RUN|STOP menu, the STOP icon on the Control Toolbar, by pressing the STOP key, executing a STOP statement in the program, or entering the STOP statement. After the program halts, the values of the variables in the main context can be examined and changed. However, changing any program line will cause the current values of the local variables to be discarded.

Pressing the RESET key, selecting the RUN|BASIC RESET menu or clicking the BASIC RESET icon on the Control Toolbar will also stop a running program, but also resets the HTBasic environment. The STOP, PAUSE, and CLR I/O keys are preferred.

# Returning to the Operating System

Control is returned to the operating system by executing the QUIT ALL statement from the keyboard or in a program line. HTBasic performs an orderly shutdown by closing all files and then it returns to Windows.

# Branching

Branching allows program execution to jump to a statement other than the next statement. The GOTO statement allows you to make an unconditional transfer to another program line in the same context, but its use is discouraged in favor of structured flow control statements. The GOTO can specify a line number or label. For example:

```
GOTO 120
GOTO Fix_up
```

# Conditional Statements

Conditional statements allow you to make decisions. The most common conditional statement is the IF...THEN statement. It allows program execution to change depending on the result of the specified expression.

The simple form of the IF...THEN statement allows either program execution to jump to another statement in the same context or a single program statement to be executed. Here are some simple IF...THEN examples:

```
IF A$="Q" THEN 700            ! conditional transfer
IF A$="N" THEN PRINT "No !"   ! single statement
```

The block IF...THEN statement allows any number of statements to be conditionally executed. If the expression evaluates true, all of the statements enclosed by the IF...THEN statement down to the END IF or ELSE statement are executed. The optional ELSE statement can be used within a block IF...THEN statement to enclose any number of statements. If the condition evaluates false, all of the statements enclosed by the ELSE statement down to the END IF are executed. A block IF...THEN statement can be nested inside another block IF...THEN statement.

```
100  IF Xlimit>Upper THEN
110    PRINT "The current setting is ";Xlimit
120    READ Xlimit
130    PRINT "The new setting is ";Xlimit
140  ELSE
150    Xlimit = Xlimit+1
160  END IF
```

In this example line 100 starts a "block" IF structure. Lines 110, 120, and 130 are all executed if Xlimit>Upper is true; otherwise, line 150 is executed.

# ON...GOTO Statement

The ON...GOTO statement provides a multi-way branch depending on the value of the specified expression. Control is transferred to one of the program lines, in the same context, selected from the list of line numbers or labels whose position in the list matches the value of the expression. If the value is 1 then the first line number is used, if the value is 2 then the second line number is used, etc. If the value is less than one or is larger than the number of line numbers or labels specified, an error is generated.

```
ON J GOTO 300,400,500,600 ! value of J determines GOTO
```

# ON...GOSUB Statement

The ON...GOSUB statement is like the ON...GOTO statement except that program control is sent to one of the subroutines, in the same context, specified in the line number or label list. When the subroutine executes the RETURN statement, control returns to the line following the ON...GOSUB statement. For more information on the GOSUB statement and subroutines, see the subroutine explanation later in this chapter.

```
ON X GOSUB 300,400,500,600 ! value of X determines GOSUB
```

# SELECT and CASE Statements

The SELECT statement begins a block which executes alternative statement blocks based on the value of the expression specified. Just as a block IF...THEN ends with an END IF statement, a SELECT block ends with an END SELECT statement. Within the SELECT block, CASE statements enclose alternative statement blocks. SELECT statements can be nested. This means that one SELECT statement can be nested inside another.

When the SELECT statement is executed, the SELECT expression is first evaluated and then the resulting value is tested against the list of values in each CASE statement until either a CASE statement matches the SELECT value or until an optional CASE ELSE statement is encountered. The enclosed program statements up to the next CASE or END SELECT statement are then executed. Control is then transferred to the line following the END SELECT statement. If no CASE statement matches and no CASE ELSE statement is encountered, control is immediately transferred to the line following the END SELECT statement.

The CASE statement specifies a list of expressions each separated by a comma. The type of the expressions, either numeric or string, must match the type of the SELECT expression. Each expression may specify either a match value, a relational operator (<, <=, =, >=, >, or <>) followed by a match value, or a range specified by a lower and an upper match value. Each expression is evaluated one at a time and the resulting value is tested against the SELECT expression result. If any expression matches, then the CASE statement matches and the statements up to the next CASE or END SELECT statement are executed.

Because the first matching CASE statement will be executed regardless of the later CASE statements, care must be exercised in selecting the order of the CASE statements.

The following example illustrates the use of the SELECT and CASE statements:

```
100  INPUT "What is your age? ",Age
110  SELECT Age
120  CASE <1,>100
130    PRINT "Do you expect me to believe that?"
140    GOTO 100
150  CASE <12
160    Price = 2
170  CASE 12 TO 59
180    Price = 6
190  CASE 60
200    PRINT "Special Rate Tonight:"
210    Price = 4.5
220  CASE ELSE
230    Price = 5
240  END SELECT
250  PRINT USING """Movie price is $"",D.2D ";Price
260  END
```

# Loops

A program loop allows the repeated execution of a set of statements. There are four types of program loops: FOR/NEXT, LOOP/END LOOP, REPEAT/UNTIL, and WHILE/END WHILE.

# FOR Loops

FOR/NEXT loops let you specify how many times to repeat a block of statements. You should use FOR loops when the block will be executed a fixed number of times. It is legal to use a GOTO statement to jump out of the FOR loop, but this violates the philosophy that the block is to be executed a fixed number of times. You may specify an optional STEP value by which the variable is to be incremented or decremented. If no STEP value is specified, it defaults to one. The value of the variable is tested against the termination value before the loop is executed the first time. If it is beyond the termination value, control transfers to the line following the NEXT statement.

The NEXT statement adds the STEP value to the value of the variable and then tests it against the termination value. If it is not beyond the termination value, control transfers to the line following the FOR statement. If it is beyond the termination value, the loop terminates and the value of the variable is left as it is. An example FOR/NEXT loop follows:

```
10  FOR J=50 TO 100 STEP 2
20    READ A(J)
30  NEXT J
```

# Other Loop Types

The other types of program loops repeatedly execute their statement block until the exit condition is satisfied. Depending on the loop statements used, the test for loop termination can be made at the beginning, the end, or at any place inside the loop. A loop can be nested inside another loop. The following examples illustrate the three types of loop termination:

```
Start of loop      End of loop        Middle of loop
100 WHILE X<>4     100 REPEAT         100 LOOP
. . .              . . .              . . .
. . .              . . .              150 EXIT IF X=4
. . .              . . .              . . .
200 END WHILE      200 UNTIL X=4      200 END LOOP
```

# Subroutines

A subroutine is accessed by a GOSUB statement and consists of one or more HTBasic statements whose last statement is a RETURN. When the GOSUB statement is encountered, the current line number is saved and control is transferred to the specified line number or line label in the same context. Execution continues until the RETURN statement is encountered, at that time control is returned to the line following the GOSUB statement. A subroutine example follows.

```
100  GOSUB 200
. . .
200  X = Y*45/Z
210  RETURN
```

This subroutine can be called from many places in the program to save having to duplicate the subroutine statements many times.

# Program Contexts

HTBasic programs are made up of a collection of contexts. Contexts are program units that have their own environment, including local variables. There are four types of contexts: the main context, a subprogram context, a user defined function context, and a CSUB context.

By default, all context variables are local. A local variable cannot be changed by another context unless it is passed by reference. It exists temporarily, only while its program context is being executed. If a context is invoked recursively, each invocation of the context has its own set of local variables.

Global variables defined in a COM statement can be accessed from any program context in the HTBasic program that includes the proper COM statement. Chapter 1, "Language Elements," contains more information on variables.

# Main Context

The main context includes all of the program lines from the first line up to and including the END statement. This context executes first and may call other subprogram contexts.

# Subprogram Contexts

Subprogram contexts are program units that begin with a SUB statement, optionally define parameters, end with an SUBEND statement, can be invoked recursively by other contexts, and can be passed arguments. They are similar to procedures and functions in other structured languages and are sometimes referred to as "true" subroutines. Subprogram contexts allow arguments to be passed, local variables to be declared, and global variables to be referenced. Subprograms are called with the CALL statement, or with a FN reference in a numeric expression. Arguments can be passed either by reference or by value.

**Note**: The term "parameters" refers to the list of variables in a SUB (or DEF FN) statement. The term "arguments" refers to the corresponding list in a CALL statement (or FN function).

**Pass by reference** means a subprogram is told the location of a variable. Therefore, altering the parameter is the same as altering the original variable. In effect, the parameter name becomes a synonym for the original variable name. I/O path variables, numeric and string variables, and array elements are all passed by reference.

**Pass by value** means a subprogram is told the value of a variable, but not where the original variable is stored. The subprogram can change the value of the parameter, but since the subprogram doesn't know where the original variable is stored, it can not modify it. Expressions, constants, and literals are passed by value. Place parentheses around a variable or array element to pass it by value instead of by reference.

A subprogram context example follows:

```
100  CALL Do_it(X,4,(Y))
. . .
199  END
. . .
200  SUB Do_it(D1,D2,D3)
210    D1=D1+1
220    D2=D2+1
230    D3=D3+1
. . .
300  SUBEND
```

Line 100 will cause X to be incremented because it is passed by reference. The 4 is passed by value and Do_it is not told where the 4 came from. The same is true for Y; it will not be incremented.

# Subprogram Pointers

A subprogram is typically referenced by explicitly naming it. For example, to call a subprogram named Wendell, use the statement

```
CALL Wendell
```

In several statements, it is also possible to name the subprogram using a string expression. This allows the name of the subprogram to change dynamically as the program runs. The subprogram must be specified with the initial character in uppercase, and subsequent characters in lowercase. For example:

```
CALL A$
```

If A$="Wendell", the statement will call the subprogram named Wendell. If A$ has some other value, the statement will call some other subprogram. The string expression specifying the subprogram name is called a "subprogram pointer" because it "points" to the subprogram rather than explicitly naming it. As the expression changes, the pointer points to different subprograms. Subprogram pointers are allowed in CALL, INMEM, LOADSUB, DELSUB, and XREF statements.

This example shows one use for subprogram pointers:

```
10  IF Case=1 THEN
20    Method$="Real"
30  ELSE
40    Method$="Complex"
50  END IF
60  IF NOT INMEM(Method$) THEN LOADSUB Method$
70  CALL Method$ WITH(X,Y,Z)
80  DELSUB Method$
90  END
```

# User Defined Function Contexts

A User Defined Function context begins with a DEF FN statement, optionally defines parameters, ends with a FNEND statement, can be invoked from within an expression by referencing its name, and can be passed arguments, either by reference or by value. When it terminates, it returns a value with a RETURN statement. The expression then continues to evaluate, using the returned value in place of the function reference.

The defined function can return either a numeric value or a string value. If it returns a string value, the function name must end with a dollar sign ($) and the RETURN statement must specify a string value. For example:

```
100  PRINT "Today is: ";FNToday$
110  END
. . .
120  DEF FNToday$
130    A$=DATE$(TIMEDATE)
140    RETURN A$[1,6]
150  FNEND
```

# CSUB Contexts

A CSUB context is a compiled subprogram created with special tools outside of HTBasic. It is loaded into memory with the LOADSUB statement and removed from memory with the DELSUB statement. It is invoked with a CALL statement.

# Interrupting Program Flow

Normal program flow can be interrupted by any of several events: CYCLE, DELAY, END, ERROR, INTR, KBD, KEY, KNOB, SIGNAL, TIME, and TIMEOUT.

The ON statement defines the action to take when an event occurs. It defines the event type, the servicing priority, the type of branch used, and the service routine. Event branching occurs between program lines and can be a GOTO, a GOSUB, a CALL, or a RECOVER.

The destination of a GOTO or GOSUB must be a program line in the present context. If the event occurs while execution is in a different context, the event is logged and execution continues. When control returns to the proper context, the branch then takes place.

The destination of a RECOVER must also be a program line in the present context. However, if the event occurs while execution is in a different context, then SUBEXITs are automatically performed until control is returned to the proper context.

The destination of a CALL must be a SUB context that defines no parameters. When the event occurs, a CALL is performed to the SUB context. Upon exit of the SUB context, control is returned to the context that was executing when the event occurred.

# Priority

The event priority designates which events can interrupt other event service routines. An event can only interrupt a lower priority routine. If the present SYSTEM PRIORITY is equal to or larger than the priority of an event handling routine, the event is logged and serviced later when the SYSTEM PRIORITY allows it. The main context begins running at a priority of 0, allowing any event to be serviced. The event priority may be specified from 1 to 15 and if it is not specified, defaults to 1.

If the branch type is a CALL or GOSUB, then when the event is serviced the SYSTEM PRIORITY is changed to the specified event priority. When a SUBEXIT or RETURN is executed, the SYSTEM PRIORITY is restored to its value before the event was serviced. If the branch type is a GOTO, the system priority is not changed. If the branch type is a RECOVER, the automatic SUBEXITs restore the SYSTEM PRIORITY to the value it was when the defining context invoked another context.

The ON END, ERROR, and TIMEOUT events form a special class and each indicates an error condition. The priority of the END and TIMEOUT events is 15 so that no normal event can interrupt their service routines. However, they can interrupt service routines for any event, including another END or TIMEOUT event. The priority of an ERROR event is 17. It cannot be set or changed with the SYSTEM PRIORITY statement.

# Global and Local Aspects

The ON/OFF state of an event and the key labels are local to each context. The initial state is inherited from the invoking context. When returning from a context, the state is restored to what it was when the context was invoked. CYCLE interval, DELAY time, KBD ALL modifier, KNOB interval, TIME value, and TIMEOUT values are global. Changing them in a context overrides their values specified in previous contexts. Consider this example:

```
10    ON KEY 1 LABEL "Done" RECOVER 40
20    CALL S
30 Spin: GOTO Spin
40    END
50    SUB S
60      ON KEY 2 LABEL "More" GOSUB More
70      SUBEXIT
80 More: PRINT "More"
90      RETURN
100   SUBEND
```

When Spin is reached, only KEY 1 will be defined. This is because events are local to the defining context and the contexts called from that context. When the SUBEXIT statement is executed in line 70, the ON KEY 2 in line 60 is discarded.

# Disabling Events

The DISABLE statement disables all defined event branches except END, ERROR, and TIMEOUT. While disabled, the first event of each type that occurs is logged. When event branching is re-enabled with the ENABLE statement, all logged events are serviced in the order of their event priorities.

An event branch definition is removed with an OFF statement specifying the matching event type. This may include an I/O path name, an interface select code, a key number, or a signal number.

```
10   A$=FNInkey$
20   PRINT LEN(A$),A$
30   END
40   DEF FNInkey$       ! Input one key without echo,
50     ON KBD GOTO 80   !  and a 10 second timeout
60     ON DELAY 10 GOTO 90
70     GOTO 70
80     RETURN KBD$
90     RETURN ""
100  FNEND
```

This example illustrates use of two ON event statements to implement an Inkey$ function. The function, as defined in the example, inputs one keystroke. If no key is pressed within ten seconds then the null string is returned.

# Error Handling

HTBasic includes many features for handling execution errors. A user written subroutine or subprogram, called an error handler, can be executed when an error occurs.

# Defining an Error Handler

The ON ERROR statement defines the action to take when an error occurs. It defines the type of branch used and the service routine. The branch type can be a GOTO, a GOSUB, a CALL, or a RECOVER. If no error handler is defined, the error message is displayed and the program PAUSEs. For example:

```
ON ERROR GOSUB 200
ON ERROR GOTO Fix_it
ON ERROR RECOVER 1510
ON ERROR CALL Handler
```

An ON ERROR statement is canceled by an OFF ERROR statement and is not disabled by the DISABLE statement.

The destination of a GOTO or GOSUB must be a program line in the present context. If the error occurs while execution is in a different context, the ON ERROR definition is ignored, the error message is displayed, and the program is PAUSEd.

The destination of a RECOVER must also be a program line in the present context. However, if the error occurs while execution is in a different context, then SUBEXITs are automatically performed until control is returned to the proper context.

The destination of a CALL must be a SUB context that defines no parameters. When the error occurs, a CALL is performed to the SUB context.

An ON ERROR can interrupt any event service routine since it has a priority of 17 which is higher than any event branch. It cannot be set or changed with the SYSTEM PRIORITY statement. If another ERROR occurs while the system is at this priority (a "double fault"), then the program is PAUSEd even though an ON ERROR definition is in effect.

# The Error Handler Routine

If the branch type is a CALL or GOSUB, then when the error is serviced the system priority is changed to 17. When a SUBEXIT or RETURN is executed, the system priority is restored to its value before the error was serviced. If the branch type is a GOTO, the system priority is not changed. If the branch type is a RECOVER, the automatic SUBEXITs restore the system priority to the value it was when the defining context invoked another context.

If an error occurs in the service routine of an ON ERROR GOSUB or CALL, it is reported to the user and the program is PAUSEd. If an error occurs in the service routine of an ON ERROR GOTO or RECOVER, an infinite loop between the error line and the error routine can result.

If there is not enough memory to run the service routine, the original error is reported to the user and the program is PAUSEd.

There are several error indicator functions that can be used by an error handler routine for decision making. The value of the error number (ERRN) and the line number where the error occurred (ERRLN) are updated when an error occurs. If no error has occurred since start-up or SCRATCH A, then a zero is returned. The ERRL function returns a one if ERRLN is equal to the specified line and a zero otherwise. The specified line must be in the current context. The ERRL function is not keyboard executable. The ERRN, ERRLN, and ERRL functions may be used in IF statements to direct program flow in error handler routines.

The ERRM$ function returns the line number (ERRLN), error number (ERRN), and the associated error message string. The null string is returned if no error has been generated since start-up, LOAD, GET, SCRATCH, or CLEAR ERROR.

Errors that occur in connection with background TRANSFER statements are not reported until the associated I/O path variable is accessed. In this case ERRLN is the number of the program line referencing the I/O path, not the TRANSFER statement. Also, ERRN is not updated.

The CLEAR ERROR statement resets ERRL, ERRLN, ERRM$, and ERRN to their default start-up values.

Error handler subroutines ending with RETURN and subprograms ending with SUBEND or SUBEXIT re-execute the line in error. If the error handler does not correct the cause of the error, the error will occur again, causing an infinite loop until the error is corrected. Subroutines ending with ERROR RETURN and subprograms ending with ERROR SUBEXIT do *not* re-execute the line in error. These statements return to the line following the line that caused the error. For example:

```
100  ON ERROR GOSUB 500
. . .
. . .
500  INPUT "Value too Large. Try again: ",N
510  ERROR RETURN
```

During program debugging it is helpful to be able to generate an error just as if it were generated by a running program. The CAUSE ERROR statement allows you to do this. When the statement is executed, it is as though the error specified actually occurred and the normal error functions: ERRL, ERRLN, ERRM$, and ERRN are updated. CAUSE ERROR is also useful in debugging error handlers.

# EXECUTE Statement

The EXECUTE statement executes an operating system command or user program. This powerful command allows you to control and intermix the execution of other user programs with your HTBasic program.

The default command interpreter for your operating system is invoked and given the command specified for execution. For example:

```
EXECUTE "sol.exe"
EXECUTE "dir"
```

executes the program or operating system command. When the command has completed, control is returned to HTBasic. If the command argument is not specified, the default command interpreter is invoked, you are given a prompt, and you may issue one or more commands. You must terminate the command interpreter to return to HTBasic by typing "EXIT".

After the command has completed execution, if the WAIT OFF option is not specified, the message "Hit any key to continue" will be displayed and HTBasic waits until you press any keyboard key. If the WAIT OFF option is specified, control immediately returns to the next HTBasic statement.

If the RETURN option is specified, the executed program's termination error value is returned in the numeric variable. This is the command interpreter's termination value.

When control is returned to HTBasic, an attempt is made to service any events that occurred while the command interpreter had control. Some events may be lost or ignored during this time period.

# Summary

This chapter described the program execution states, conditional branching statements, looping statements, subroutines, subprogram contexts, user defined functions, event and error handling statements, and the EXECUTE statement. More detailed information about these statements is available in the on-line *Reference Manual*.

# Mathematics

This chapter describes the mathematical capabilities of HTBasic. Numeric, string, and matrix expressions are made up of operands and operators. Operands can be variables, constants, or the results of expressions. Operators can be infix operators like + and -, built-in functions like COS and EXP, or user defined functions. This chapter also describes automatic data type conversions, execution precedence, and the matrix inversion function.

# Numeric Expressions

A numeric expression is defined as any legal combination of operands and operators joined together in such a way that the expression as a whole can be reduced to a numeric value. The following syntax diagram defines the legal combination of operands and operators. Precedence rules place additional constraints that are explained later in this chapter.

numeric-expression =
    { + | - | NOT } numeric-expression |
    ( numeric-expression ) |
    numeric-expression operator numeric-expression |
    numeric-constant |
    numeric-name |
    numeric-array-element |
    numeric-function [ ( param [,param...] ) ] |
    FN function-name [ ( param [,param...] ) ] |
    string-expression compare-operator string-expression

where:
operator = + | - | * | / | DIV | MOD | MODULO | ^ |
    AND | OR | EXOR | compare-operator
compare-operator = <> | = | < | > | <= | >=
numeric-function = a function, like COS, which returns
    a numeric value.
param = legal parameters for numeric functions and user
    defined functions are explained in the on-line *Reference Manual*.

# String Expressions

A string expression is any legal combination of operands and operators joined together in such a way that the expression as a whole can be reduced to a string value. The following syntax diagram defines the legal combination of operands and operators. Additionally, precedence rules should be kept in mind.

string-expression =
    ( string-expression ) |
    string-expression & string-expression |
    "string-literal" |
    string-name |
    string-array-element |
    sub-string |
    string-function [ ( param [,param...] ) ] |
    FN function-name$ [ ( param [,param...] ) ]

where:
string-function = CHR$ | COMMAND$ | DATE$ | DVAL$ |
    ENVIRON$ | ERRM$ | IVAL$ | KBD$ | LWC$ | REV$ |
    RPT$ | SYSTEM$ |TIME$ | TRIM$ | UPC$ | VAL$
param = legal parameters for string functions and user
    defined functions are explained in the on-line *Reference Manual*.

# Matrix Expressions

A matrix expression is any legal combination of operands and operators joined together in such a way that the expression as a whole can be reduced to a matrix value. The following syntax diagram defines the legal combination of operands and operators.

```
MAT string-array$ = string-array$ | (string-expression)
MAT sub-array[$] = sub-array[$]
MAT numeric-array = numeric-array [operator numeric-array]
MAT numeric-array = (numeric-expression) [operator numeric-array]
MAT numeric-array = numeric-array operator (numeric-expression)
MAT matrix = matrix-function [ ( matrix [,matrix] ) ]
```

where:

```
operator = + | - | . | / | < | <= | = | <> | >= | > | *
sub-array = array-name( { * | lower-bound:upper-bound | subscript }... )
matrix-function = RSUM | CSUM | INV | TRN | IDN |
      REAL | IMAG | ARG | ABS | CONJG | CMPLX
param = legal parameters for matrix functions are explained
      in the on-line Reference Manual.
```

# Operands

An operand is defined as that which an operator operates on. For example, in the equation:

```
A*PI+4/N
```

the operands are A, PI, 4, and N, while "*", "+", and "/" are the operators. Operands can be strings, I/O paths, complex, real, long or integer numbers, arrays or array elements. They can be in the form of constants, variables, array elements, or in some cases, entire arrays.

# Operators

An explanation of operators that begin with a letter of the alphabet can be found by looking up the name of the operator in the on-line *Reference Manual*. The following charts list all of the operators, grouping them by category.

# Arithmetic

The arithmetic operators provide the standard arithmetic operations as well as the INTEGER division, remainder, and modulo operators.

| Operator | Meaning | Example | Result |
|---|---|---|---|
| + | Addition (dyadic/binary) | 3+4 | 7 |
| + | Positive (monadic/unary) | +4 | 4 |
| - | Subtraction (dyadic/binary) | 3-4 | -1 |
| - | Negation (monadic/unary) | -3 | -3 |
| * | Multiplication | 3*4 | 12 |
| / | Division | 3/4 | 0.75 |
| ^ | Exponentiation | 3^4 | 81 |
| DIV | Integer Division | 4 DIV 3 | 1 |
| MOD | Remainder | 4 MOD 3 | 1 |
| MODULO | Modulo | 4 MODULO 3 | 1 |

Several of these operations can generate errors. The following table outlines the possible errors. In general, the error numbers returned by HTBasic are the same as those returned by HP BASIC. But in some instances the operating system or environment in which HTBasic runs makes it impossible or impractical to return the same number.

| Math Operation | Cause of Error | Example |
|---|---|---|
| Integer +-* DIV | Result too big | 32760+32760 |
| Long +-* DIV | Result too big | 2,147,483,647+1 |
| DIV | Divide by zero | 1 DIV 0 |
| Real +-*/ | Result too big | 1E200*1E200 |
| Real +-*/ | Result too small | 1E-200*1E-200 |
| / | Divide by zero | 1/0 |
| MOD/MODULO | MOD by 0 | 1 MOD 0 |
| A^B | Result too big | 1E200^1E200 |
| A^B | A<0 and B non-integer | (-2)^6.5 |
| A^B | A=0 and B<0 | 0^(-1) |

# Binary

Binary functions perform bit-wise operations on integer numeric values. They may be used to manipulate bits or to perform conditional operations based on their logical result.

| Operator | Meaning | Example | Result |
|---|---|---|---|
| BINAND | Bit-wise "AND" | BINAND(3,4) | 0 |
| BINCMP | Bit-wise complement | BINCMP(5) | -6 |
| BINEOR | Bit-wise Exclusive Or | BINEOR(3,5) | 6 |
| BINEQV | Bit-wise Equivalence | BINEQV(3,5) | -7 |
| BINIMP | Bit-wise Implication | BINIMP(3,5) | -3 |
| BINIOR | Bit-wise "OR" | BINIOR(3,4) | 7 |
| BIT | Bit-wise test | BIT(4,2) | 1 |
| ROTATE | Bit-wise rotation | ROTATE(3,-4) | 48 |
| SHIFT | Bit-wise logical shift | SHIFT(3,-4) | 48 |

# Conversions

Conversion functions change an operand from one data type to another. Numeric/string conversions can operate either on ASCII character values (i.e. CHR$(65)) or string representations of numbers (i.e. "65").

| Operator | Meaning | Example | Result |
|---|---|---|---|
| CINT | Real to integer | CINT(16.0) | 16 |
| CMPLX | Real/integer to complex | CMPLX(2,1) | 2+*i* |
| CHR$ | Numeric to ASCII string | CHR$(65) | "A" |
| DVAL | Base N to base 10 (32 bit) | DVAL("A",16) | 10 |
| DVAL$ | Base 10 to base N (32 bit) | DVAL$(11,16) | "B" |
| &H | Hexadecimal constant | &H10 | 16 |
| IVAL | Base N to base 10 (16 bit) | IVAL("A",16) | 10 |
| IVAL$ | Base 10 to base N (16 bit) | IVAL$(11,16) | "B" |
| NUM | ASCII character to numeric | NUM("A") | 65 |
| &O | Octal constant | &O20 | 16 |
| REAL | Integer/complex to real | REAL(16) | 16.0 |
| VAL | String to numeric | VAL("65") | 65 |
| VAL$ | Numeric to string | VAL$(65) | "65" |

# Date/Time

Date and time functions read the system time and convert time values in seconds to the familiar human readable forms and vice versa.

| Operator | Meaning | Example | Result |
|---|---|---|---|
| DATE | String to seconds | DATE("1 JAN 1980") | 2.11182336E+11 |
| DATE$ | Seconds to string | DATE$(2.11182336E+11) | "1 JAN 1980" |
| TIME | String to seconds | TIME("01:00:00") | 3600 |
| TIME$ | Seconds to string | TIME$(2.11182336E+11) | "00:00:00" |
| TIMEDATE | Present time/date | TIMEDATE | seconds |

# Environment

Environment functions return information about the HTBasic environment. The SYSTEM$ function, particularly, provides a wealth of information and is explained in detail later in this chapter.

| Operator | Meaning | Example | Result |
|---|---|---|---|
| CHRX | Character cell width | CHRX | 8 |
| CHRY | Character cell height | CHRY | 14 |
| COMMAND$ | Command line | COMMAND$ | "-O -Z 2" |
| CRT | Display ISC | CRT | 1 |
| ENVIRON$ | Environment variable | ENVIRON$("PATH") | "C:\;C:\DOS" |
| FRE | Available Memory | FRE | 300000 |
| KBD | Keyboard ISC | KBD | 2 |
| MAXREAL | Largest REAL number | MAXREAL | 1.798E+308 |
| MINREAL | Smallest REAL number | MINREAL | 2.225E-308 |
| NPAR | Number of parameters | NPAR | 0 |
| PRT | Printer device selector | PRT | 10 |
| RATIO | Graphic screen ratio | RATIO | 1.48 |
| SYSTEM$ | System information | SYSTEM$("MSI") | "C:\" |

# Error

Error functions give information about the latest error that occurred. This information is useful in an error handling routine established by ON ERROR.

| Operator | Meaning | Example | Result |
|---|---|---|---|
| ERRL | Test for error line | ERRL(100) | 0 |
| ERRLN | Line number | ERRLN | 10 |
| ERRM$ | Error Message | ERRM$ | "ERROR 31 IN "... |
| ERRN | Error Number | ERRN | 31 |

# I/O Functions

I/O functions complement the regular I/O statements by providing additional information about I/O operations, devices, and paths.

| Operator | Meaning | Example | Result |
|---|---|---|---|
| KBD$ | ON KBD, keyboard buffer | KBD$ | "A" |
| KNOBX | ON KNOB, x movement | KNOBX | 217 |
| KNOBY | ON KNOB, y movement | KNOBY | -45 |
| PPOLL | Parallel poll on IEEE-488 | PPOLL(7) | 8 |
| READIO | Read hardware register | READIO(9,0) | 7 |
| SC | Select Code in I/O path | SC(@Io) | 10 |
| SPOLL | Serial poll on IEEE-488 | SPOLL(701) | 0 |
| STATUS | Read interface register | STATUS(CRT,1) | 1 |

# Logical

Logical operators can be used on integer or real numbers. The two values are first converted to logical (TRUE=1, FALSE=0) values, the operation is done, and the result is converted to an integer. When converting numbers to logical values, zero is converted to FALSE and non-zero is converted to TRUE. When converting the result to an integer, FALSE is converted to a zero, and TRUE is converted to a one.

| Operator | Meaning | Example | Result |
|---|---|---|---|
| AND | Logical "and" | 2 AND 3 | 1 |
| EXOR | Logical "exclusive or" | 2 EXOR 3 | 0 |
| OR | Logical "or" | 1 OR 0 | 1 |
| NOT | Logical "not" | NOT 1 | 0 |

# Relational

Relational operators can be used on numbers or strings. Relational operators can be used in assignment statements, IF statements, and any other place a numeric expression is legal. For example:

```
10 X = 4*(Y>Z)+J*(A=B AND R<T)
```

Relational operators may be used on strings to compare the LEXICAL ORDER of the two strings. By default, ASCII values are used to determine relative order. "A" is **less than** "B". If two strings of different length are the same up to the end of the shorter string, then the shorter string is **less than** the longer string. For example, "ABCDE" < "ABCDEF". The LEXICAL ORDER IS statement affects the relational ordering of strings.

| Operator | Meaning | Example | Result |
|---|---|---|---|
| < | Less than | 3<4 | 1 |
| <= | Less than or equal | 3<=4 | 1 |
| = | Equals | "3"="4" | 0 |
| >= | Greater than or equal | 3>=4 | 0 |
| > | Greater than | "3">"4" | 0 |
| <> | Not equal | 3<>4 | 1 |

# String Functions

As you have seen, many other operators already described also have string operands or string results. The functions presented here are especially useful for operations with strings. The examples in the following table assume that A$ has the value "HOTDOG".

| Operator | Meaning | Example | Result |
|---|---|---|---|
| & | Concatenation | "HOT"&"DOG" | "HOTDOG" |
| [s] | Sub-string, start | A$[4] | "DOG" |
| [s,e] | Sub-string, start, end | A$[1,3] | "HOT" |
| [s;l] | Sub-string, start, length | A$[2;4] | "OTDO" |
| COMMAND$ | Command line | COMMAND$ | "-O -Z 2" |
| CVT$ | Converts alphabets | CVT$(X$,Y$) | |
| ENVIRON$ | Environment variable | ENVIRON$("PATH") | "C:\;C:\WINDOWS" |
| FBYTE | Test for First byte | FBYTE(X$) | 0 or 1 |
| LEN | Present length | LEN("AB") | 2 |
| LWC$ | Lowercase | LWC$("AB") | "ab" |
| MAXLEN | Dimensioned length | MAXLEN(A$) | 18 |
| POS | Position of a sub-string | POS("AB","B") | 2 |
| REV$ | Reverse | REV$("AB") | "BA" |
| RPT$ | Repeat | RPT$("AB",3) | "ABABAB" |
| SBYTE | Test for Second byte | SBYTE(X$) | 0 or 1 |
| TRIM$ | Trim lead/trailing space | TRIM$(" A B ") | "A B" |
| UPC$ | Uppercase | UPC$("ab") | "AB" |

A substring defines a portion of a string variable or string array element. The capability of specifying a sub-string of a string variable or string array element is quite powerful. This capability replaces the RIGHT$, LEFT$, MID$, REP$, and SEG$ functions of other BASICs. A sub-string is selected by specifying a starting position within the string value, and optionally, either the length of the sub-string or the ending position within the string value. If only the starting position is specified, the rest of the string value from that point on is used for the sub-string. String positions are one-based, i.e., the first character of a string is in position one.

# Transcendental and Trigonometric

The standard transcendental and trigonometric functions are provided along with the ability to specify degree or radian operations. Many other less common functions are available, contact TransEra for information.

| Function | Meaning |
|---|---|
| ACS | Returns the arc cosine of an expression |
| ASN | Returns the arc sine of an expression |
| ATN | Returns the arc tangent of an expression |
| ATN2 | Returns the angle to a point |
| COS | Returns the cosine of an expression |
| DEG | Statement to set degree mode for trig functions |
| EXP | Return the exponential of an expression |
| LGT | Computes common (base 10) logarithms |
| LOG | Computes natural (base e) logarithms |
| PI | Returns the numeric value 3.14159... |
| RAD | Statement to set radian mode for trig functions |
| SIN | Returns the sine of an expression |
| SQR | (SQRT) Returns the square root of an expression |
| TAN | Returns the tangent of an expression |

# Other Functions

The other functions provide number manipulation for the sign, the fractional or integral parts, rounds to specific decimal places, finds the largest or the smallest value, and generates pseudo-random numbers.

| Operator | Functionality |
|---|---|
| ABS | Absolute value of an expression |
| CINT | Convert to Integer |
| DROUND | The number rounded to specified number of digits |
| FIX | Discard fractional part of a number |
| FRACT | Fractional part of a number |
| INT | Greatest integer part of a real number |
| MIN | Smallest number from list of values and arrays |
| MAX | Largest number from list of values and arrays |
| PROUND | The number rounded to the specified decimal place |
| RES | Result of last live keyboard expression |
| RND | Random number |
| SGN | Arithmetic sign of an expression |

Notice the differences among CINT, FIX, and INT. CINT converts a REAL value to an INTEGER by substituting the closest INTEGER to the value. FIX returns the closest integral value between the REAL value and zero. INT returns the closest integral value between the REAL value and negative infinity. Also, CINT actually changes the type from REAL to INTEGER while INT and FIX return integral results, but the type is not changed. The following table helps illustrate the differences:

| X | CINT(x) | FIX(x) | INT(x) |
|---|---|---|---|
| 2.6 | 3 | 2.0 | 2.0 |
| 2.2 | 2 | 2.0 | 2.0 |
| -2.2 | -2 | -2.0 | -3.0 |
| -2.6 | -3 | -2.0 | -3.0 |

# User Defined Functions

The DEF FN function statement defines a subprogram function context. This function is executed whenever the function name is referenced in a numeric or string expression. You can define as many functions in this way as you wish. Chapter 2, "Program Flow," introduces subprogram contexts and explains how to define and pass arguments to them.

# Automatic Conversions

Conversions from REAL to INTEGER or LONG and from INTEGER or LONG to REAL are done automatically in HTBasic. Basic operations are done in INTEGER math if both operands are INTEGER or LONG. Otherwise, REAL math is used. For example:

```
INTEGER J          ! J is now an integer type variable
X = 1.234          ! X is a real number (by default)
J = X              ! The real value of X is converted to
                   ! integer and assigned to J.
X = J              ! This conversion is from integer to real
X = 1.0            ! Faster than X=1 (no convert required)
X = 1              ! This requires a convert to real
X = PI DIV 2.0*10 ! X will equal ten.
```

The last example above is kind of tricky. The first operation to take place will be INTEGER division. The INTEGER division operation will convert PI to an INTEGER 3 and 2.0 to INTEGER 2. 3 DIV 2 equals INTEGER 1. The multiply will be an INTEGER multiply because 1 and 10 are both INTEGERs. 1*10 equals INTEGER 10, which is converted to REAL 10.0 to be stored in the REAL variable, X.

The same concepts are extended in versions of HTBasic with COMPLEX support. If one operand is COMPLEX, the other is automatically converted to COMPLEX if needed. In cases where a real number is required, a complex number will automatically be converted to a real number by discarding the imaginary portion. In cases where an integer or long number is required, a complex number will cause an error. In this situation, use the REAL function to force the real part of the complex number to be used.

Conversions may take a noticeable amount of time if many iterations occur. They should, therefore, be avoided whenever speed is a priority.

# Execution Precedence

Mathematical precedence describes the order in which operators in an expression are evaluated. For example, the correct answer to the formula:

```
1+2*3+4
```

is 11, not 13. This is because multiplication (2*3) has a higher precedence than addition (1+2). If the two operators are on the same row in the precedence chart, the operations occur in left to right order (i.e. 1+2-3+4).

HP BASIC (and HTBasic) has an odd quirk in its definition of precedence that you should be aware of. Most computer languages place all monadic operators (operators that operate on one operand) at a higher precedence than dyadic operators (operators that operate on two operands). However, HP BASIC (and HTBasic) place monadic + and - below some of the dyadic operators. The following is one example of an expression that will evaluate differently because of this:

```
-4^0.5
```

With HTBasic, this is equivalent to -(4^0.5) which is equal to -2. With most other computer languages, this is equivalent to (-4)^0.5 which is an illegal operation.

| **Precedence** | **Operators/Functions** |
|---|---|
| 1 | Parentheses () and sub-strings [] |
| 2 | Functions: built in and user defined. |
| 3 | Exponentiation Operator ^ |
| 4 | Multiplicative Operators *,/,DIV,MODULO,MOD |
| 5 | Monadic + and - |
| 6 | Dyadic + and - |
| 7 | String Concatenation & |
| 8 | Relational Operators =,<>,<,>,<=,>= |
| 9 | Monadic Logical Operator NOT |
| 10 | Logical Operator AND |
| 11 | Logical Operators OR and EXOR |

# Matrix Operators

One of the powerful features of HTBasic is its ability to do operations on complete arrays without the use of loops. This means that programs will run much faster. Many operators that can operate on two simple variables can operate on arrays. Array/array operations or array/scalar (simple variable) operations can be done. Portions or entire arrays can be transferred to another array or a portion of another array. For example:

```
100 DIM X(10),Y(10),Z(10)
110 MAT Y=(1)           ! defines every element of the array
120 MAT X=Y*(5)         ! array/scalar operation
130 MAT Z=X+Y           ! array/array operation
140 MAT Z(2:3)=Z(9:10) ! sub-array assignment
```

The operators + - . / < <= = >= > <> require that the operand arrays have the same RANK and that each dimension has the same SIZE. The result array will be REDIMed if needed. However, the usual rules for REDIM apply and if the array cannot be redimensioned, an error is returned. Each of these operators work on the array element by element. The "." operator does an element by element multiply.

The "*" operator performs classical matrix multiplication. The definition of matrix multiplication is given in the following BASIC SUB:

```
10   SUB Matmpy(A(*),B(*),C(*)) ! Equivalent to MAT C=A*B
20     OPTION BASE 1
30     INTEGER I,J,K,M,N,R
40     M=SIZE(A,1)
50     N=SIZE(A,2)
60     K=SIZE(B,2)
70     IF N<>SIZE(B,1) THEN CAUSE ERROR 16
80     REDIM C(M,K)
90     FOR I=1 TO M
100      FOR J=1 TO K
110        Sum=0
120        FOR R=1 TO N
130          Sum=Sum+A(I,R)*B(R,J)
140        NEXT R
150        C(I,J)=Sum
160      NEXT J
170    NEXT I
180  SUBEND
```

# Matrix Operators with Matrix Result

Besides applying these simple operators to arrays, operators especially designed for arrays can be used:

| Operator | Functionality |
|---|---|
| CSUM | Returns the sum of each column of a 2D array in a vector |
| IDN | The identity matrix (1's along diagonal, 0's elsewhere) |
| INV | Sets one array to the inverse of another |
| REORDER | Reorders the elements of an array |
| RSUM | Returns the sum up each row of a 2D array in a vector |
| SEARCH | Searches for elements in an array |
| SORT | Sorts arrays in ascending or descending order |
| TRN | Transposes a matrix (rows to columns, columns to rows) |

# Matrix Operators with Scalar Result

The following operators take a matrix operand and return a scalar result.

| Operator | Functionality |
|---|---|
| BASE | Returns the lowest legal subscript for a dimension |
| DET | Returns the determinant of a matrix |
| DOT | Dot, or inner product of two vectors |
| MAX | Returns largest element of an array and/or scalars |
| MIN | Returns smallest element of an array and/or scalars |
| RANK | Number of dimensions in a matrix |
| SIZE | Upper bound - lower bound + 1 of a dimension |
| SUM | Adds up all the elements in an array |

# Matrix Sub-array Assignments

Sub-array assignments (sometimes called array slices) require that the number of ranges specified in the source match the number of ranges specified in the destination. If a complete array is specified, the number of ranges equals the rank of the array. In corresponding ranges of the source and destination, the number of elements must be the same. The following examples will help you visualize these rules:

```
10 DIM X(1:3),Y(1:10)
20 DIM D(3,4,5),S(4,2,5)
30 MAT X=Y(2:4)           ! One range, three elements
40 MAT D(3,*,*)=S(*,2,*)  ! Range 1 has 5 elements, 2 has 6
50 MAT Y(1:6)=S(0,0,*)    ! One range, 6 elements
```

# Matrix Searching

The MAT SEARCH statement searches a numeric or string array for certain conditions. The array can be searched for the following:

- The location of first element that is less than, greater than, equal to, or not equal to a given value
- A count of the number of locations that are less than, etc. to a given value
- The location of the maximum or minimum value in the array
- The value which is the maximum or minimum value in the array

The syntax for MAT SEARCH is:

MAT SEARCH numeric-array [num-key], rule; return [,start]
MAT SEARCH string-array$ [str-key], rule; return [,start]

where:
num-key = [search-subscripts] [DES]
str-key = [search-subscripts [sub-string]] [DES]
search-subscripts = ( {subscript|*} [,...] )
    The '*' must appear once.
rule =
    [#]LOC ([relational] value) |
    LOC MAX |
    LOC MIN |
    MIN |
    MAX
relational = < | <= | = | <> | => | >
return = variable-name
start = numeric-expression
value = string-or-numeric-expression

The keyword DES specifies descending search order. The optional start value specifies the starting subscript. If not specified, searching begins with the first element for ascending searches and the last element for DEScending searches. The meaning of the search rule is:

| Operator | Functionality |
|---|---|
| LOC | Subscript of first element satisfying operator |
| #LOC | Count the number of elements satisfying operator |
| LOC MAX | Subscript of maximum value |
| LOC MIN | Subscript of minimum value |
| MAX | Find and return the maximum value |
| MIN | Find and return the minimum value |

# Matrix Inversion

One of the more complex matrix functions is the INV function used to calculate the inverse of a matrix. Several precautions are in order when using the INV function. The inverse of a matrix **A** is defined to be that matrix **B**, such that

**AB = BA = I**

where **I** is the identity matrix. For any matrix **A**, there is no guarantee that a matrix **B** exists such that the above relationship can be satisfied. When this is the case, **A** is called a singular matrix and has a determinant value of zero.

This leads to the **first caution** to observe when using computer methods for matrix inversion. After using the INV() function to find the inverse of a matrix, the DET function should be tested to determine whether the matrix was singular or non-singular. The determinant of a matrix is a by-product of the computer's inversion algorithm. When the INV function is used, the determinant of that matrix is assigned to the DET function. In the case of a non-singular matrix, it is therefore faster to do the inversion first and then check the DET function. The example below shows both methods.

```
110 ! The fast way
120 MAT B=INV(A)
130 IF NOT DET THEN PRINT "A is singular"
140 !
150 ! The slow way
160 !
170 IF NOT DET(A) THEN PRINT "A is singular"
180 MAT B=INV(A)
```

No error is generated when a singular matrix is inverted, but the values assigned to the result matrix are meaningless. You should therefore check the determinant value when doing a matrix inversion.

The **second caution** is related to the first. When the determinant of a matrix is very near zero, compared to the other elements of the matrix, then the inexact arithmetic used by a computer causes errors in the calculation of the inverse. The closer to zero, the more error is introduced into the result. To test for this condition, multiply the original matrix and its inverse together and compare the result to the identity matrix. If the difference is greater than what is acceptable for your application, then you will not be able to use BASIC to invert that matrix.

The following example illustrates a matrix whose determinant is small compared to the elements of the matrix.

```
10   REM SMALLDET.BAS
20   DATA 100,200,100.000000000001,200
30   DIM A(1,1),B(1,1),Ab(1,1)
40   READ A(*)
50   MAT B=INV(A)
60   MAT Ab=A*B
70 D2x2: IMAGE K,/,2(2(SD.15DE,2X),/)
80   PRINT USING D2x2;"A=",A(*)
90   PRINT USING D2x2;"B=",B(*)
100  PRINT USING D2x2;"Ab=",Ab(*)
110  PRINT "DET = ";DET
120  END
```

The output from this program is shown below. Although the product **AB** is not exactly the identity matrix, it is close enough for many applications.

```
A=
+1.000000000000000E+02  +2.000000000000000E+02
+1.000000000000010E+02  +2.000000000000000E+02

B=
-1.005267773966630E+12  +1.005267773966630E+12
+5.026338869833190E+11  -5.026338869833140E+11

Ab=
+9.956054687500000E-01  +5.371093750000000E-03
-4.394531250000000E-03  +1.005371093750000E+00

DET = -1.98951966013E-10
```

# Complex Operators

Several operators and statements have been added to work with complex numbers. Automatic conversion rules have also been extended to handle cases where complex and real arguments are mixed. See "Automatic Conversions" earlier in this chapter.

# Extended Statements and Operators

| Operator | Functionality |
|---|---|
| + - * / | Operate on complex numbers |
| = <> | Operate on complex numbers |
| ABS | Return absolute value (magnitude or modulus) |
| ATN | Return arctangent of complex number |
| COS | Return cosine of complex number |
| ALLOCATE | COMPLEX can be specified |
| COM | COMPLEX can be specified |
| DATA | Use rectangular form separated by comma |
| DISP | Display in rectangular form separated by space |
| DEF | COMPLEX can be specified |
| ENTER | Enter in rectangular form separated by non-numeric |
| EXP | Return "e" raised to complex power |
| IMAGE | Treat complex like two REALs |
| INPUT | Input in rectangular form |
| LGT | Return base 10 log of complex number |
| LOG | Return natural log of complex number |
| MAT | ABS, ARG, CMPLX, CONJG, IMAG, REAL |
| MAT REORDER | Reorder complex arrays |
| MAT SEARCH | Search complex arrays |
| OUTPUT | Output in rectangular form separated by comma |
| PRINT | Print in rectangular form separated by space |
| READ | Read in rectangular form |
| SIN | Return sine of complex number |
| SQR | Return square root of complex number |
| SUB | COMPLEX can be specified |
| TAN | Return tangent of complex number |

Notice that only two of the relational operators, <> and =, are extended for complex numbers. The other relational operators depend on the linear ordering of the data type. In other words, all REAL, LONG or INTEGER numbers could be laid out on a number line and, of any two different numbers, one would be closer to positive infinity than the other. Since linear ordering is not defined for the complex data type, the <, >, <=, and >= operators are also not defined.

# Added Statements and Operators

| Operator | Functionality |
|---|---|
| ACSH | Hyperbolic arccosine of complex or real |
| ARG | Argument (angle) of complex number in polar form |
| ASNH | Hyperbolic arcsine of complex or real |
| ATNH | Hyperbolic arctangent of complex or real |
| CMPLX | Combine two REALs into a complex |
| COMPLEX | Declare complex variables |
| CONJG | Conjugate of complex |
| COSH | Hyperbolic cosine of complex or real |
| IMAG | Return imaginary part of complex number |
| REAL | Return real part of complex number |
| SINH | Hyperbolic sine of complex or real |
| TANH | Hyperbolic tangent of complex or real |

# Summary

This chapter has described numeric, string, and matrix expressions; operands, operators, and functions; automatic data type conversions, execution precedence, and the matrix inversion function. More information about each of these operators and functions can be found in the on-line *Reference Manual*.

# Graphics

HTBasic contains an extensive assortment of powerful graphic statements. These allow you to use convenient data units, not just pixels, in defining your graphic display. Your data units are automatically scaled to the correct graphic device units. Also, the same program can use a variety of graphic devices (screen, plotter, or printer) without having to modify the graphics statements.

This chapter coordinates information and concepts relating to the HTBasic graphics system. The graphics statements are introduced beginning with the simple statements and progressing to the more complex ones. Several examples are given to help you see as well as understand the concepts that are presented.

Because of the large number of HTBasic graphic statements, not all of the options and syntax details are explained in this chapter. For more detailed syntax information or attribute values refer to the statement descriptions in the on-line *Reference Manual* as you read through this chapter.

# Simple Graphics Statements

The simplest graphics statements initialize the graphics system, clear the GRAPHICS and ALPHA screens, and control the pen movement to draw graphic lines. The HTBasic statements used to perform these functions as well as the graphic coordinate system used by HTBasic are described in the following paragraphs.

# GINIT Statement

The GINIT statement resets all the graphics parameters to their default values. It terminates any graphics input device or active plotter. If graphics output is directed to a file, the file is closed. It also causes the graphics screen to be cleared before the next graphics statement is executed. If you enter GINIT followed by a DRAW 50,50 statement the following occurs: GINIT resets all graphics parameters to their default values. Before the DRAW command is executed the screen is cleared and then a line is be drawn from the origin to 50,50.

# GCLEAR and CLS Statements

The GCLEAR statement erases both the GRAPHIC and the ALPHA screen, then re-displays the ALPHA screen. To clear the ALPHA screen, use the CLEAR SCREEN or CLS commands.

# Graphics Coordinate System

The HTBasic graphics system is based on the Cartesian coordinate system. This system uses a pair of values to define the location of each point in a graph relative to the origin at (0,0). The first value specifies how far the point is to the right of the origin and the second value specifies how far the point is above the origin. Negative values specify locations to the left of or below the origin. The horizontal line passing through the origin is called the X axis and a vertical line passing through the origin is called the Y axis.

The default origin, (0,0), is the lower left corner of the display screen. The default top vertical value is 100. The default right horizontal value depends upon the display aspect ratio and is usually about 148.

Let us now examine the simple graphic statements used to control the pen movement to generate graphic lines. They are the MOVE, DRAW, PLOT, and PENUP statements.

# MOVE and DRAW Statements

The MOVE statement raises the pen and then moves it to the specified position. The DRAW statement draws a line from the current position to the specified position using the current line type and pen number. MOVE always lifts the pen before moving to the specified position. DRAW always begins with the pen down, draws to the new position, and ends with the pen down. Let's now try an example:

```
10 GINIT
20 DRAW 100,100
30 MOVE 100,0
40 DRAW 0,100
50 END
```

This example draws a large X on the graphics screen. If you were not already in GRAPHICS mode the display mode is switched to GRAPHICS mode and any ALPHA text is repainted on the graphics screen. The first DRAW statement lowers the pen at the current position (0,0 because of the GINIT statement) and draws a line to position 100,100. The MOVE statement raises the pen and moves it to position 100,0. The next DRAW statement lowers the pen at the current position 100,0 and draws a line to position 0,100.

# PLOT Statement

The PLOT statement, like the MOVE and DRAW statements, moves the pen to the specified location and optionally specifies when the pen is to be raised or lowered. For example:

```
PLOT 45,80,-1
```

first lowers the pen and then moves it to location 45,80. If the optional pen-control value is not specified, the pen is lowered after a move. The pen-control value is interpreted as follows.

| Pen Control | Action |
|---|---|
| positive even #, & zero | pen is raised after a move |
| positive odd # | pen is lowered after a move |
| negative even # | pen is raised before a move |
| negative odd # | pen is lowered before a move |

Negative values cause the pen action to occur **before** the move and positive values cause the pen action to occur **after** the move. Even numbers cause the pen to be **raised**, and odd numbers cause the pen to be **lowered**.

A two or three column numeric array can be used to supply the coordinate and optional pen-control values. If a three-column array is specified, the third-column specifies the pen-control value to use for each row. It can also specify many other operations as covered later in this chapter. The earlier MOVE/DRAW example could have used the PLOT statement as follows:

```
10 GINIT
20 DATA 100,100,-1, 100,0,-2, 0,100,-1
30 INTEGER A(2,2)
40 READ A(*)
50 PLOT A(*)
60 END
```

This example draws the same large X on the screen using the PLOT statement and a three-column data array to specify the coordinates and the pen-control values.

# PENUP Statement

The PENUP statement raises the pen without changing its position. This is used with plotters when you don't want the pen to "bleed" onto the paper while it is not moving.

# User Defined Graphic Units

Up to this point we have been working in the default graphic units. We now turn our attention to specifying the graphic units that are most convenient for the display of your data values.

The computer screen is, in effect, our viewport into the entire cartesian coordinate system. Only the graphic points that fall within the viewport will be displayed; all other points are eliminated. Lines that cross through the viewport are clipped at the boundaries. The portion of the screen that is to be used to display graphics is specified by the VIEWPORT statement. The CLIP statement allows you to specify clipping boundaries that are different than the VIEWPORT. The SHOW and WINDOW statements specify which portion of the cartesian coordinate system is mapped into the VIEWPORT for display.

# VIEWPORT Statement

The VIEWPORT statement specifies the area of the screen or graphic device to be used for graphics output and it also sets the soft-clip boundary limits to match the viewport bounds. The VIEWPORT parameters control the proportions, size, and position of the drawing surface. All graphic output is automatically scaled to fit this drawing surface. The coordinate of the left edge must be less than that of the right edge and the bottom edge must be less than the top edge. It is specified as follows:

`VIEWPORT` *`Left,Right,Bottom,Top`*

The VIEWPORT boundary parameters are defined in GDUs (Graphic Display Units). GDUs are units that describe the physical bounds of the display area on the graphic output device. By definition, Graphic Display Units are 1/100 of the shorter axis of a plotting device. A unit in the X direction and a unit in the Y direction are the same length. The RATIO function returns the ratio of the X to Y physical bounds for the PLOTTER IS device and can be used to determine the VIEWPORT soft-clip limits.

If the ratio is less than 1, the X axis is 100 GDUs and the Y axis is (100*RATIO) GDUs long; if the ratio is greater than 1, the Y axis is 100 GDUs and the X axis is (100*RATIO) GDUs long. The VIEWPORT soft-clip limits should not exceed the physical bounds of the device. By default the left limit is zero, the right limit is the X axis physical bound, the bottom limit is zero, and the top limit is the Y axis physical bound.

Changing the VIEWPORT does not affect any currently displayed graphics, only graphics that you subsequently generate.

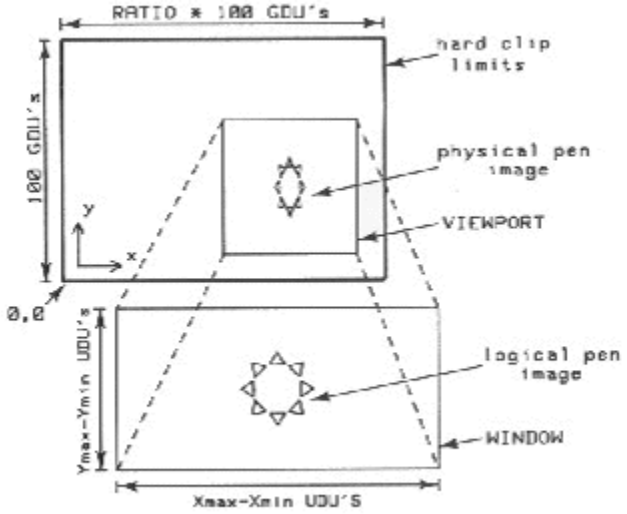



**Figure 4-1: Simplified Graphics Mapping**

As mentioned any graphic points that fall outside the VIEWPORT are eliminated and any lines that cross through the VIEWPORT are clipped at the boundaries. These boundaries are called the soft-clip bounds. They must be within the device physical limits or the hard-clip bounds.

The following program outputs the same graphics information to three different VIEWPORTs.

```
10  GINIT
20  VIEWPORT 10,50,60,85        !Viewport #1
30  GOSUB 100
40  VIEWPORT 60,100,60,85       !Viewport #2
50  GOSUB 100
60  VIEWPORT 10,100,30,50       !Viewport #3
70  GOSUB 100
80  STOP
90  !
100 FRAME
110 AXES 10,10,20,20,2,3
120 RETURN
130 END
```

The first and second viewports are exactly the same size, they are just located at different sections of the screen. The third viewport stretches the X axis and compresses the Y axis, causing the image to be distorted.

# CLIP Statement

The CLIP statement allows you to specify new soft-clip limits without changing the current VIEWPORT values and it enables and disables clipping at the soft-clip boundaries. If no CLIP statement is executed, the soft-clip boundaries are the most recently defined by either a VIEWPORT (soft-clip) or PLOTTER IS (hard-clip) statement.

Use the CLIP ON statement prior to any graphic statements that might generate points outside the soft-clip area. If CLIP ON is active, a theoretical move or draw to any point that is outside the defined soft-clip area is executed. If a draw is executed, then only that portion of the vector which lies inside the soft-clip area is drawn. The portion of the vector that lies outside the soft-clip area is clipped (chopped off) at the edge of the soft-clip boundary. If both the current logical position and the specified position are outside the soft-clip area the logical position is updated but no physical pen movement is made.

Execute a CLIP OFF statement to disable clipping so you may add labels, comments, graphics or any other plotting that is to be done outside the soft-clip area. When clipping is disabled, clipping will only be done on the physical device limits. If the soft-clip limits are smaller than the physical device boundaries, then CLIP OFF allows you to generate graphic coordinates that fall outside the soft-clip limits but inside the device physical boundaries. For example:

```
10 GINIT
20 VIEWPORT 10,100,50,80
30 FRAME
40 AXES 3,3,0,0,2,3
50 CLIP OFF
60 MOVE 15,-25
70 LABEL "LABEL OUTSIDE CLIP AREA"
80 END
```

The default WINDOW setting (0,RATIO*100,0,100) is mapped into the new VIEWPORT area of (10,100,50,80). We enclose the VIEWPORT area with the FRAME statement so you can see the area. Clipping is turned off and we move outside the soft-clip area to (15,-25) and draw a line.

The difference between the CLIP and VIEWPORT statements can be confusing. The following examples should help you see the different effects these commands have on graphics scaling and clipping.

**Example A**

```
10 GINIT
20 VIEWPORT 20,60,20,60
25 WINDOW 0,148,0,100
30 FRAME
40 MOVE 10,10
50 DRAW 80,70
60 CLIP OFF
70 DRAW 90,70
80 END
```

**Example B**

```
10 GINIT
20 CLIP 20,60,20,60
25 WINDOW 0,148,0,100
30 FRAME
40 MOVE 10,10
50 DRAW 80,70
60 CLIP OFF
70 DRAW 90,70
80 END
```

In example A, the WINDOW values (0,148,0,100) are mapped into the new VIEWPORT area (20,60,20,60). All the MOVE and DRAW X,Y values are within the window and thus no clipping is done. In example B, the VIEWPORT remains the entire screen and the WINDOW values are again mapped into the VIEWPORT area, but the CLIP command specifies that any line outside the range of the clipping area (20,60,20,60) is not displayed. After CLIP OFF, the final DRAW is displayed.

You specify the bounds and the units of your coordinate system with either the SHOW or the WINDOW statement. They both specify a rectangular area with dimensions as large or as small as you like. The units that you thus define are known as *User Defined Units* (UDUs) and are used by all the graphic drawing statements. The meaning of each unit is entirely up to you. They can be any units of measure you wish to work with (inches, miles, years, etc.). For example, if you are plotting a sine wave that has a domain of 0 to 2*PI and a range of -1 to +1, you would use these values as the bounds of your coordinate system.

# SHOW Statement

The SHOW statement specifies the bounds of the data values to be displayed within the VIEWPORT in isotropic units so that the X and Y units are of equal length. You specify the left, right, bottom, and top coordinate bounds as follows:

```
SHOW Left,Right,Bottom,Top
```

The SHOW values are manipulated internally to give you isotropic units in both the X and Y directions. The SHOW statement finds the difference between the X and Y ranges and the smaller range is scaled into the larger, causing the specified area to be centered within the plotting area. For example:

```
SHOW -100,100,2,10
```

For a screen with square pixels the calculations would be: an X difference of 200 and Y difference of 8 is found by subtracting the lower bound from the upper. The smaller Y range is scaled into the larger X range. The difference between the X and Y ranges is 192 (200-8), and half of this, 96, is applied to each Y value. The new minimum Y value is -94 (2-96) and the maximum value is 106 (10+96).

For screens that don't have square pixels, the values are automatically adjusted to prevent distortion.

# WINDOW Statement

The WINDOW statement specifies the bounds of the data values to be displayed within the VIEWPORT in non-isotropic units where the X and Y units are of different lengths. If not specified, the default WINDOW is equal to the default VIEWPORT setting. It is specified as follows.

```
WINDOW Left,Right,Bottom,Top
```

The SHOW and WINDOW statements only differ in how they map data onto the VIEWPORT.

An image can be "mirrored" about the X or Y axis by reversing the order of the limits for each dimension by specifying the larger value before the smaller value. This is true for both the SHOW and WINDOW statements. For example:

```
SHOW 0,RATIO*100,100,0     !Mirror about Y Axis
WINDOW RATIO*100,0,0,100   !Mirror about X Axis
```

**Please note:** You do not have to set the WINDOW bounds to whole units. Set them to the units most convenient for your data. The coordinates are always translated to the units required for the full resolution of your graphic device.

# WINDOW and VIEWPORT Effects

We will now use the RECTANGLE statement to see how it is affected by different WINDOW and VIEWPORT values. There are several forms of the RECTANGLE statement. The simplest form specifies the desired width and height and draws only the border lines. For example:

```
10 GINIT
20 MOVE 10,20
30 RECTANGLE 10,10
40 END
```

draws a small square near the origin. You can change the WINDOW to a smaller range to cause the square to be drawn larger. Add the following statement.

```
24 WINDOW 0,30,0,30
```

The VIEWPORT is still the entire screen, but the range of values mapped into the VIEWPORT is now smaller. The WINDOW values are (0,30,0,30) instead of the default (0,Ratio*100,0,100). This creates a larger box when the same RECTANGLE statement is used. Note that the square is now stretched onto the VIEWPORT. This is because the VIEWPORT is not square and the WINDOW was not adjusted to compensate. To get a square on the screen either the VIEWPORT or the WINDOW must be changed. The simplest way is to select a square VIEWPORT by adding the following statement.

```
22 VIEWPORT 0,100,0,100
```

Note that this displays the square again, but it is shifted to the left, since the right side of the screen is inaccessible with this VIEWPORT. Try experimenting with different WINDOW and VIEWPORT settings to understand how they interact with each other.

# Annotating Charts and Graphs

Now that you know how to specify a convenient coordinate system and its units of measure we turn our attention to the HTBasic statements used to annotate charts and graphs.

# AXES and GRID Statements

By including an axis or grid with appropriate tic marks and labels you can make data plots and graphs more readable and meaningful. The AXES and GRID statements make this easy. The AXES statement draws a single X-Y axis across the soft-clip area, while the GRID statement generates grid lines across the entire soft-clip area. They are specified as follows:

```
AXES [xtic [,ytic [,xorg [,yorg [,xcnt [,ycnt [,size]]]]]]]
GRID [xtic [,ytic [,xorg [,yorg [,xcnt [,ycnt [,size]]]]]]]

where:
x/ytic = tic spacing
x/yorg = origin of axis
x/ycnt = major tic counts
size   = major tic size
```

The default values for the X and Y tic spacing and the axis or grid location are 0,0,0,0. The X and Y major tic counts specify the number of tic intervals between major tic marks. Their default values are 1 indicating that every tic interval is major. The default major tic length is two graphic display units. The minor tic marks are half the length of the major tic marks.

The AXES statement produces tic marks that are symmetric about the axis and that extend to the soft-clip boundaries. If the X or Y axis is outside the soft-clip area, tic marks are drawn in the soft-clip area. The AXES lines and tic marks are drawn in the current line type and pen number. A major tic is placed at the axis origin.

The GRID statement generates major tic marks lines across the entire soft-clip area. Cross tic marks are generated at the minor tic mark intersections. The grid is drawn with the current line type and pen number. The pen position after a GRID statement is at the axis origin. For example:

```
10 GINIT
20 AXES 5,5,50,50,2,3
30 DRAW 60,60
40 END
```

produces an axis with the origin at 50,50. A tic mark is placed every 5 units in both the X and Y direction. In the X direction every second tic mark is twice as big as the others, because it is a major tic. Every third tic mark in the Y direction is a major tic. The DRAW statement shows you the current X,Y location. Now change line 20 to the GRID statement.

```
20 GRID 5,5,50,50,2,3
```

The origin of the GRID is at 50,50 just like the AXES. The major tic marks extend across the entire soft-clip area. Where the minor tic marks cross, a small tic is placed. The DRAW statement again shows you the current X,Y location.

To create a fully enclosed box with tic marks along the outside use two AXES statements, one with an intercept in the lower left corner of the screen, and the other in the upper right corner of the screen.

# FRAME Statement

The FRAME statement draws a line around the soft-clip area using the current pen and line type. It ends with the pen up and is positioned in the lower left corner of the FRAME.

# LABEL Statement

To annotate a graphics image with text the LABEL statement is used. The CSIZE, LDIR, LORG and GFONT IS statements control the graphic text size, direction, origin and font, respectively.

The LABEL statement draws graphic text beginning at the current pen position, in the current pen number and line type. Labels are clipped at the soft-clip boundary just like any other graphics. The scaling of the SHOW and WINDOW statements have no effect on the LABEL statement. This keeps them from becoming distorted by the scaling of graphic data.

The LABEL statement is similar to the PRINT statement except that the text is drawn on the graphics screen. See the PRINT statement for an explanation of arrays, numeric and string fields, and numeric and string formats. Also the following control characters have a special meaning when processed by the LABEL statement:

| Keystroke | Character | Action |
|---|---|---|
| CTRL-H | CHR$(8) | moves pen left one character cell |
| CTRL-J | CHR$(10) | moves pen down one character cell |
| CTRL-M | CHR$(13) | moves pen left length of completed label |

**CSIZE Statement**

The CSIZE statement sets the character size (height) and optionally the expansion factor (width/ height) for the text generated by the LABEL statement. For example:

```
CSIZE 10,.8
```

Both values are specified in graphic display units. The default character height is five and the default expansion factor is 0.6. These values are in effect at start-up, or when GINIT, or RESET are executed. A negative height or expansion-factor inverts the character in relation to that dimension.

# LDIR Statement

The LDIR statement specifies the angle of rotation from the X-axis that the LABEL is drawn. A value of zero specifies drawing along the positive X-axis. Positive values specify a counter-clockwise direction. The current trigonometric mode (RAD or DEG) determines the units for the angle. The default is radians. For example:

```
LDIR 0.56
```

# LORG Statement

The LORG statement specifies the relative position of the LABEL with respect to the current pen position. Its argument has a range of one through nine. The default LORG origin is one. The values are defined as follows:

| Left Values | Middle Values | Right Values |
|---|---|---|
| 3 - left-top | 6 - middle-top | 9 - right-top |
| 2 - left-center | 5 - middle-center | 8 - right-center |
| 1 - left-bottom | 4 - middle-bottom | 7 - right-bottom |

If the LABEL is an odd number of characters, the center of the string is the center of the middle character. The following program demonstrates all nine LORG values and the effect it has on LABELS.

```
10 FOR I=1 TO 9
20   GINIT
30   DRAW 50,50
40   LORG I
50   LABEL "LORG ";VAL$(I)
60   WAIT 1
70 NEXT I
80 END
```

# GFONT IS Statement

The GFONT IS statement specifies the font to use with LABEL statements. This can be set to any valid font in the Windows font directory. Non-proportional fonts are recommended. Proportionality of proportional fonts is not maintained.

# Graphic Attributes

We now cover the concepts relating to, and the statements used to specify the graphic attributes that can be used to modify displays. These are the graphic line types, colors, and writing modes. They are described in the following paragraphs.

# LINE TYPE Statement

The LINE TYPE statement sets the style or dash pattern and optionally the repeat length of drawn graphic lines. The repeat factor is the GDU line length before the line pattern is repeated.

The default LINE TYPE is number one, a solid line. Dotted and dashed lines of various types may be specified in a manner similar to this example:

```
LINE TYPE 5
```

When the GRAPHIC device is not the screen, the line types are device dependent. The following example demonstrates the screen line types.

```
10 GINIT
20 FOR T=1 TO 10
30   LINE TYPE T
40   Y = T*10
50   MOVE 10,Y
60   DRAW 90,Y
70 NEXT T
80 END
```

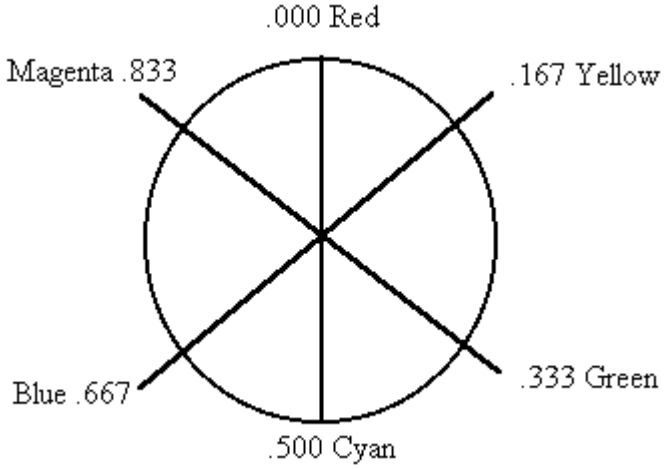
# Color Graphics

HTBasic provides powerful statements for generating color graphic displays. Both the HSL (Hue, Saturation, Lightness) and the RGB (Red, Green, Blue) color definitions can be used to specify a color. An alternate drawing mode and many graphics writing modes allow great flexibility in creating graphic displays. Each of these topics is covered in the following paragraphs.

# HSL Color Space

The HSL color space is designed to be intuitive and follows the model of mixing paints. An artist preparing a color for a painting, first selects a *hue* (pure color pigment). He may then add black or white paint to arrive at the desired color. Adding white serves to wash out the color. In technical terms, we say this affects the *saturation* of the color. The artist may then adjust the brightness by adding black paint. This affects the amount of light reflected by the pigment. We call this the *luminosity*.

Saturation ranges from zero (white) to one (pure color - no added white). Luminosity ranges from zero (black) to one (pure color - no added black). Hue ranges from zero to one. The diagram below gives an indication of where several colors occur in that range:



# RGB Color Space

The RGB color space is designed to match the way in which our eyes work, and in turn, the way in which television and computer displays are designed. The display has three color guns: Red, Green, and Blue. By specifying a number in the range zero (corresponding to zero intensity) to one (corresponding to maximum intensity) for each of the three guns, you can uniquely define all the colors that can be produced by that display.

# Pen Numbers

The graphics color is specified with the PEN statement as shown by this example:

```
PEN 4
```

If the current graphic device is the plotter it selects which pen is used to draw the lines. If the current graphic device is the computer system display, the pen number selects the color in which all graphics lines are drawn. Some display systems can operate in more than one graphics mode, and the number of available colors depends on the current graphics mode.

The following table gives the pen number to RGB number assignments. For monochrome displays, only the first two entries apply.

| PEN | COLOR | Red | Green | Blue |
|---|---|---|---|---|
| 0 | black | 0 | 0 | 0 |
| 1 | white | 1 | 1 | 1 |
| 2 | red | 1 | 0 | 0 |
| 3 | yellow | 1 | 1 | 0 |
| 4 | green | 0 | 1 | 0 |
| 5 | cyan | 0 | 1 | 1 |
| 6 | blue | 0 | 0 | 1 |
| 7 | magenta | 1 | 0 | 1 |

# COLOR MAP Mode

The Color Map mode is always set and does not need to be specified in the PLOTTER IS statement. A Color Map mode allows any color to be assigned to any pen. The SET PEN statement is used to assign colors to pens. The following table gives the default color to pen assignments.

| Pen | Color | Pen | Color |
|---|---|---|---|
| 0 | black | 8 | black |
| 1 | white | 9 | olive green |
| 2 | red | 10 | aqua |
| 3 | yellow | 11 | royal blue |
| 4 | green | 12 | maroon |
| 5 | cyan | 13 | brick red |
| 6 | blue | 14 | orange |
| 7 | magenta | 15 | brown |

# SET PEN Statement

The SET PEN statement defines part or all of the color map. A color may be specified in either the RGB or the HSL color space by using the INTENSITY or the COLOR keywords respectively. You may redefine an individual pen by specifying one HSL or RGB color value or multiple pens by specifying an array. In either case, the pen-number specifies the first entry in the color map to be defined. For example:

```
SET PEN pen-number COLOR h, s, l
SET PEN pen-number COLOR numeric-array(*)
SET PEN pen-number INTENSITY r, g, b
SET PEN pen-number INTENSITY numeric-array(*)
```

The pen-number should be in the range 0 to n-1, where n is the number of colors supported by the map. The closest possible color will be used if the computer display cannot display the color you select.

**Immediate Effect.** Any pixels already drawn with the specified pen are changed to the new color. All SET PEN statements take effect immediately upon execution. The effects of all SET PEN statements last until the next SET PEN statement of the same pen, or until GINIT, PLOTTER IS, SEPARATE/MERGE ALPHA, or QUIT. In cases where dithering is used, changing the color map changes the colors available to the dithering process, and changes the colors of areas already drawn with dithering that use that particular pen as part of the dither pattern.

In addition to the PEN and SET PEN statements the GESCAPE statement can affect the graphics color.

# GESCAPE Statement

The GESCAPE statement exchanges device-specific information with a graphic device. It is specified as follows:

```
GESCAPE device-selector, code [,param(*)][;return(*)]
```

The device selector specifies the graphic device. The code value specifies the type of operation. The param array sends information to the device and the return array receives information from the device. The type, size, and shape of the arrays must be appropriate for the requested operation.

| Code | Description |
|---|---|
| 1 | returns number of color map entries |
| 2 | returns color map values |
| 3 | returns hard-clip values |
| 4 | sets normal drawing mode |
| 5 | sets alternate drawing mode |
| 6 | returns graphic display mask |
| 7 | sets graphic display mask |
| 102 | returns the current VIEWPORT and WINDOW |
| 103 | returns the current PEN and AREA PEN assignments |
| 104 | sets PLOTTER device specific variables |
| 105 | sets GRAPHICS INPUT device specific variables |
| 106 | sets DUMP device specific variables |

# Code 1

Code 1 returns the number of color map entries. The return array must be a one dimensional INTEGER array and have at least one element. The first element is assigned the number of color map entries. The following program shows how to return the number of color map entries.

```
10 INTEGER A_return(0)
20 GESCAPE CRT,1;A_return(*)
30 PRINT A_return(0)
40 END
```

# Code 2

Code 2 returns color map values. The return array must be a two dimensional REAL array, with at least one row and three columns. The first row contains color information for pen 0, second row for pen 1, etc. If the array does not have enough rows, only part of the color map is returned. If the array has too many rows, only part of the array will be used. The first column contains the information for red, the second for green, and the third for blue. The color information ranges in value from zero to one. Color values are multiples of 1/N,

where N is the number of non-black shades available for each color.

```
A_return(0,0) - Pen 0 red color map value
A_return(0,1) - Pen 0 green color map value
A_return(0,2) - Pen 0 blue color map value
. . . .    .
. . . .    .
. . . .    .
A_return(15,0) - Pen 15 red color map value
A_return(15,1) - Pen 15 green color map value
A_return(15,2) - Pen 15 blue color map value
```

The following program shows how to return the color map values.

```
10 REAL A_return(15,2)
20 GESCAPE CRT,2;A_return(*)
30 PRINT A_return(*)
40 END
```

# Code 3

Code 3 returns the hard-clip values and GSTORE array size. The values are returned in plotter units or pixels. The return array must be a one dimensional INTEGER array and must contain at least four elements. The first four elements of the array are assigned the values, X min, Y min, X max, Y max, respectively. For a CRT, the fifth and sixth elements give the INTEGER array dimensions needed by the GSTORE command to store the screen image.

```
A_return(0) - X minimum hard clip value
A_return(1) - Y minimum hard clip value
A_return(2) - X maximum hard clip value
A_return(3) - Y maximum hard clip value
A_return(4) - # of rows that GSTORE requires
A_return(5) - # of columns that GSTORE requires
```

The following program shows how to return the hard-clip & GSTORE values.

```
10 INTEGER A_return(5)
20 GESCAPE CRT,3;A_return(*)
30 PRINT A_return(*)
40 END
```

# Codes 4 & 5

Codes 4 and 5 change the graphics writing mode. If the code is 4, the drawing mode is set to normal. If 5 is specified, the drawing mode is set to alternate. The graphics writing mode provides a great deal of flexibility in the generation of graphic displays. It defines the method used to modify the pixel color bits. These include clearing, setting, inclusive or, exclusive or, complementing, and anding the color bits with the complement of the current pixel color bits. It is specified by a combination of the drawing mode and the sign of the current pen number.

The writing mode of the pen is specified by the current drawing mode and the sign of the pen number. The following table defines the different writing modes available. P is a positive pen number, X is the present value of a pixel.

| | **GESCAPE CRT,4** | **GESCAPE CRT,5** |
|---|---|---|
| **Statement** | **Normal** | **Alternate** |
| PEN P | P | BINIOR(X,P) |
| AREA PEN P | P | BINIOR(X,P) |
| PEN 0 | BINCMP(X)* | 0 |
| AREA PEN 0 | 0 | 0 |
| PEN –P | BINAND(X,BINCMP(P)) | BINEOR(X,P) |
| AREA PEN –P | BINAND(X,BINCMP(P)) | BINAND(X,BINCMP(P)) |

GESCAPE codes 4 and 5 are not supported on monochrome graphic displays such as CGA and Hercules (HGC). The following statements show how to change the graphics writing mode.

```
GESCAPE CRT,4   !Set to Normal Drawing Mode
GESCAPE CRT,5   !Set to Alternate Drawing Mode
```

# Code 6

Code 6 returns the graphics display mask. The return array must be a one dimensional INTEGER array, and must have at least one element. The first element is assigned the value of the graphics write-enable mask. The second element, if present, is assigned the value of the graphics display-enable mask. Each bit in the mask corresponds to one of the bit planes. Bit 0 corresponds to the first plane.

```
A_return(0) - graphics write enable mask
A_return(1) - graphics display enable mask
```

The following program shows how to return the graphics enable masks.

```
10 INTEGER A_return(1)
20 GESCAPE CRT,6;A_return(*)
30 PRINT A_return(*)
40 END
```

# Code 7

Code 7 sets the graphics display mask. *This operation is not supported by HTBasic.* The param array must be a one dimensional INTEGER array, and must have at least one element. The first element is assigned to the graphics write-enable mask. The second element, if present, is assigned to the graphics display-enable mask.

```
A_param(0) - graphics write enable mask
A_param(1) - graphics display enable mask
```

The following program shows how to set the graphics enable masks.

```
10 INTEGER A_param(1)
20 A_param(0)=8
30 A_param(1)=15
40 GESCAPE CRT,7,A_param(*)
50 END
```

# Basic Window Manipulation Codes

Several GESCAPE codes allow manipulation of the HTBasic window.

| Code | Operation |
|---|---|
| 30 | Maximize the window |
| 31 | Hide the window |
| 32 | Restore the window |
| 33 | Set interior client of the app window position and size |
| 34 | Get interior client of the app window position and size |
| 35 | Bring the window to the top |
| 36 | Get the screen size |
| 37 | Returns the Title Bar enable flag |
| 38 | Hide / restore title bar |
| 39 | Set the DUMP size (% of paper width) |
| 41 | Minimize the window |

The following GESCAPE CRT codes have been added for manipulation of the program window.

| Code | Operation |
|---|---|
| 46 | Turn the Control Toolbar Off |
| 47 | Turn the Control Toolbar On |
| 48 | Turn the Status Bar Off |
| 49 | Turn the Status Bar On |
| 50 | Remove Main Menu |
| 51 | Restore Main Menu |
| 52 | Disable Borders on Parent Window |
| 53 | Enable Borders on Parent Window |
| 54 | Disable Minimize button on Parent Window |
| 55 | Enable Minimize button on Parent Window |
| 56 | Disable Maximize button on the Parent Window |
| 57 | Enable Maximize button on the Parent Window |
| 58 | Disable Close button on the Parent Window |
| 59 | Enable Close button on the Parent Window |
| 60 | Turn the Bookmark Toolbar Off |
| 61 | Turn the Bookmark Toolbar On |
| 62 | Turn the Debug Toolbar Off |
| 63 | Turn the Debug Toolbar On |
| 64 | Filename in titlebar off |
| 65 | Filename in titlebar on |

The following GESCAPE CRT codes have been added for manipulation of the program child window.

| Code | Operation |
|---|---|
| 130 | Maximize the window |
| 131 | Hide the window |
| 132 | Restore the window |
| 135 | Bring the window to the top |
| 137 | Returns the Title bar enable flag |
| 138 | Hide / Restore the Title bar (Toggle switch) |
| 141 | Minimize the window |
| 152 | Disable Borders on Child Window |
| 153 | Enable Borders on Child Window |

The following example shows the syntax for several of the GESCAPES. Note that codes that set information have a comma before the array name while codes that get information have a semicolon.

```
10  INTEGER Get4(1:4),Set4(1:4),Get2(1:2),Set1(1:1)
20  DATA 90,100,500,300    ! Position of upper left corner:
30                         ! 90,100), Width = 500, Height = 300
40  READ Set4(*)
50  GESCAPE CRT,30          ! Maximize the window
60  GESCAPE CRT,31          ! Hide the window
70  GESCAPE CRT,32          ! Restore the window
80  GESCAPE CRT,33,Set4(*) ! Set position and size: X,Y,W,H
```

```
90  GESCAPE CRT,34;Get4(*) ! Get position and size: X,Y,W,H
100 GESCAPE CRT,35         ! Bring the window to the top
110 GESCAPE CRT,36;Get2(*) ! Get the screen size: W,H
120 GESCAPE CRT,37;Get3(*) ! Get the title bar enable flag
130 PRINT Get(2)           ! Print the Screen Size
140 PRINT Get(3)           ! Print the title bar enable flag
150 Set1(1)=50             ! Set the DUMP size to 50%
160 GESCAPE CRT,38         ! Hide window Title Bar
170 GESCAPE CRT,38         ! Restore window Title Bar
180 GESCAPE CRT,39,Set1(*) ! Set the DUMP size (default is 100%)
190 GESCAPE CRT,41         ! Minimize the window
200 GESCAPE CRT,32         ! Restore the window
210 END
```

# Code 102

Code 102 returns the current VIEWPORT and WINDOW values. The *return* array should be a two dimensional REAL array with two rows and four columns. The first row is assigned the values of the current window. The second row is assigned the values of the current viewport. For each, the X min, X max, Y min, and Y max values are assigned to the first through fourth columns, respectively. The following program demonstrates this capability:

```
10  REAL W(1,3)
20  GESCAPE CRT,102;W(*)
30  PRINT "The current window   is";W(0,0),W(0,1),W(0,2),W(0,3)
40  PRINT "The current viewport is";W(1,0),W(1,1),W(1,2),W(1,3)
50  END
```

# Code 103

Code 103 returns the current PEN and AREA PEN assignments. The *return* array should be a one dimensional INTEGER array with two elements. The first element is assigned the current PEN assignment. The second element is assigned the current AREA PEN assignment. The following program demonstrates this capability:

```
10  INTEGER P(1)
20  GESCAPE CRT,103;P(*)
30  PRINT "The current      PEN is";P(0)
40  PRINT "The current AREA PEN is";P(1)
50  END
```

# Code 104

Code 104 sets device-specific information. The *param* array must be a one dimensional INTEGER array. The number of elements required depends on the device driver. Conventionally, it contains two elements. The first element is the operation number and the second element is the value associated with that operation. See the documentation for each driver for information on any GESCAPE 104 operations supported by that driver.

For the HPGL plotter driver, this code is used to enable HPGL/2 capabilities. When HPGL/2 is used, polygons are sent to the plotter for rendering. With many plotting devices, this allows the polygons to be filled. When generating an HPGL file for import into other programs, it is often more desirable for the polygon to import as a single unit, rather than a series of lines. To enable HPGL/2, use the following code. Substitute the ISC for the HPGL plotter in place of ISC in line 40.

```
10  INTEGER Param(1)
20  Param(0)=1    ! HPGL Operation Number: 1 = HPGL/2 Flag
30  Param(1)=1    ! Value: 1=enable, 0=disable
40  GESCAPE Isc,104,Param(*)
```

# Code 105

Sets device-specific information in the GRAPHICS INPUT IS device. The *param* array must be a one dimensional INTEGER array. The number of elements required depends on the device driver. The first element is the operation number and the subsequent elements are the values associated with that operation.

# Code 106

Sets device-specific information in the DUMP DEVICE IS device. The *param* array must be a one dimensional INTEGER array. The number of elements required depends on the device driver. The first element is the operation number and the subsequent elements are the values associated with that operation.

For the dump drivers, code 106, operation 1 is used to specify a portion of the screen to dump when DUMP GRAPHICS is executed. The syntax is:

```
GESCAPE PRT,106,param(*)
```

The *param* array must be a one dimensional INTEGER array of five elements. The first element is the operation number. The remaining elements specify the boundary for the DUMP. The boundary is specified in screen units:

```
param(1) - 1
param(2) - Beginning row
param(3) - Ending row
param(4) - Must be 0
param(5) - Must be 0
```

The row parameters will revert back to the default of full screen if any of the following conditions occur:

1. Beginning row greater than ending row
2. A new Plotter, Graphics, or Dump driver is loaded
3. A GINIT, SCRATCH A, PLOTTER IS, GRAPHICS INPUT, or CONFIGURE DUMP, commands are executed or a Basic reset is performed.

The CONFIGURE DUMP statement must be executed before the GESCAPE statement. The following program demonstrates this capability:

```
10   INTEGER A(1:5)
20   DUMP DEVICE IS PRT
30   CONFIGURE DUMP TO "PCL"
40   A(1)=1   ! operation code, always 1
50   A(2)=100 ! begin row, screen units
60   A(3)=300 ! end row, screen units
70   A(4)=0   ! reserved, must be 0
80   A(5)=0   ! reserved, must be 0
90   GESCAPE PRT,106,A(*)
100  FRAME
110  MOVE 0,0
120  DRAW 100,100
130  DUMP GRAPHICS
140  END
```

# Incremental and Relative Graphics

Incremental and relative graphic construction allows you to describe an object or symbol in incremental or relative coordinates, and then position it repeatedly in various locations on the screen (or output device) simply by performing an absolute MOVE to the starting location and then executing the incremental or relative graphics statements. The IMOVE, IDRAW, IPLOT, and RPLOT statements implement incremental and relative graphics.

# IMOVE Statement

The IMOVE statement lifts and moves the pen to the position calculated by adding the specified X and Y displacement to the current pen position. After the IMOVE statement is executed, the logical pen position is updated and the pen is left in the up position.

# IDRAW Statement

The IDRAW statement lowers the pen and then moves it to the position calculated by adding the specified X and Y displacement to the current pen position. After the IDRAW statement executes, the logical pen position is updated and the pen is left in the down position. IDRAW 0,0 draws a point.

# IPLOT Statement

The IPLOT statement moves the pen from its current position by the specified X and Y displacements. Like the PLOT statement discussed previously, you can also specify an optional pen control value or a two- or three-column array.

# RPLOT Statement

The RPLOT statement is the same as the IPLOT statement except that it moves the pen relative to the logical pen position. You specify when the pen is to be raised or lowered with the optional pen-control value.

If the arguments of an IMOVE, IDRAW, IPLOT, or RPLOT statement specify a destination point that is outside the soft-clip area, the logical position is set to that point but the pen is not moved. Only the portion of the vector that lies inside the clipping area is plotted. If both the current logical position and the specified position are outside the clip area the logical position is updated but no physical pen movement is made.

# Arcs, Circles and Rectangles

Special purpose graphic statements are used to generate arcs, circles, and rectangles. They are the POLYGON, POLYLINE, and RECTANGLE statements. Each is discussed in the following paragraphs.

# POLYGON and POLYLINE Statements

The POLYGON and POLYLINE statements generate variable sided polygons or circles. The pen starts and ends in the same position, and after execution the pen is up. The radius is the distance between the logical pen position and the polygon vertices where the first vertex is in the positive X axis direction. A negative radius will rotate the arc by 180 degrees. The PDIR statement can be used the specify the starting angle.

You can specify the number of chord segments in a full circle and the number to draw. By default there are 60 segments in a full circle. Here is an example of an arc and a circle:

```
POLYGON 10          !Circle with radius 10
POLYLINE 10,20,3    !Arc of 3/20ths circle
```

For the POLYGON statement, if the number of chords drawn is less than the specified total number of chords, the polygon closure is affected. If the pen is up when the POLYGON statement is executed, the polygon is closed by drawing the last vertex to the first vertex. If pen is down, the polygon is closed by drawing the last vertex to the center of the polygon and then drawing from the center to the first vertex. The following program shows the difference:

```
10 GINIT
20 MOVE 35,50
30 POLYGON 10,10,8
40 MOVE 65,50
50 DRAW 65,50
60 POLYGON 10,10,8
70 END
```

For the POLYLINE statement, if the number of chords drawn is less than the specified total number of chords, the polygon is not closed. If the pen is up when the POLYLINE statement was executed, the first vertex is on the perimeter. If the pen is down when the POLYLINE statement is executed, the first point (logical pen position) is drawn to the first point on the perimeter. Change the previous program so that line 30 and 60 use POLYLINE statements. Execute the program again to see the effect of the pen being up or down.

```
30 POLYLINE 10,10,8
60 POLYLINE 10,10,8
```

# RECTANGLE Statement

The RECTANGLE statement generates a four sided polygon described by its width and height displacement from the current pen position. The signs of the width and height determine the pen position after the RECTANGLE execution: If the width is positive, the pen position is on the left side of the rectangle. If the width is negative, the pen position is on the right side of the rectangle. If the height is positive, the pen position is in the lower corner of the rectangle and if it is negative, the pen position is in the upper corner. The following example will help clarify the last pen position.

```
10 MOVE 50,50
20 RECTANGLE 3,5         !Pen at lower left
30 RECTANGLE 3,-5        !Pen at upper left
40 RECTANGLE -3,5        !Pen at lower right
50 RECTANGLE -3,-5       !Pen at upper right
60 END
```

.The POLYGON and RECTANGLE statements may specify the FILL and EDGE options described later in this chapter. If neither are specified, EDGE is assumed. The POLYLINE statement cannot specify these options.

# User Defined Symbols

The SYMBOL statement uses a two-dimensional two- or three-column array to plot a user-defined symbol. Symbols are created with moves and draws in the symbol coordinate system; an area nine units wide and fifteen units high. The symbol may be defined to extend outside of this 9x15 cell; thus allowing you to create any size symbol you wish. The CSIZE, LDIR, and LORG statements affect the SYMBOL statement, but the SHOW or WINDOW to VIEWPORT scaling does not. This keeps the labels from becoming distorted by graphic data scaling.

The symbol data is drawn using the current pen control and line type and is clipped at the clip boundary. A move is always done to the first point and the current pen position is left at the last X,Y position specified in the array and is not updated to the next symbol position. This allows a SYMBOL to cover a group of character cells. The following program shows you how to define a SYMBOL to extend outside the character cell.

```
10 GINIT
20 DIM A(6,2)
30 READ A(*)
40 DATA 1,7,-2,16,7,-1,15,4,-1,21,8,-1,15,12,-1,16,9,-1,1,9,-1
50 DRAW 50,50
60 RECTANGLE 3,5
70 SYMBOL A(*)
80 END
```

Notice that the symbol is not closed at one end. Another X,Y data point could be added to the DATA statement to fix this, or the FILL and EDGE commands would also work. Taking the same data points as defined for the SYMBOL command, change line 70 to either the PLOT or RPLOT statements to see what happens.

```
70 PLOT A(*)     !Absolute Points
                 or
70 RPLOT A(*)    !Relative Points
```

The array is now affected by the scaling of the WINDOW values, and the data points are mapped onto the WINDOW. This causes the arrow to become larger. The PLOT statement is in absolute coordinates, that is why it is close to the origin of 0,0. RPLOT is defined in relative coordinates, being relative to the current position of 50,50.

# AREA Fill Styles and Colors

The area fill styles and colors used by the PLOT, IPLOT, RPLOT, POLYGON, RECTANGLE, and SYMBOL statements are specified with the AREA statement. Area fills are specified by the FILL and EDGE options. These are described in the following paragraphs.

# AREA Statement

The AREA statement defines the area fill style and color. It is qualified with either the PEN, COLOR, or INTENSITY keywords. The area color and style remain in effect until another AREA, GINIT, or SCRATCH A is executed. The IPLOT, PLOT, RPLOT, and SYMBOL statements can also change the area fill color. This is explained under "Array Specified Pen Control," later in this chapter.

When you use the AREA PEN statement the fill color is looked up in the color map table, and the fill style is set to non-dither (solid).

```
AREA PEN pen
```

Pen numbers have the same effect as described in the PEN statement except for the following two conditions. In normal drawing mode, pen 0 erases; it does not complement. In alternate drawing mode, negative pen values erase; they do not complement. The default area fill color is PEN one.

When you use the AREA COLOR or AREA INTENSITY statements the fill color is created with dithering when needed to better approximate the color specified. Use AREA COLOR to specify a color in the HSL color space and AREA INTENSITY to specify a color in the RGB color space. These are explained earlier in this chapter. Because of the calculations needed to set up dithering, AREA COLOR/INTENSITY executes slower than AREA PEN.

Run this example program to see how dithered colors look, and to see how the HSL color space works. Change line 90 to AREA INTENSITY to see how the RGB color space works.

```
10   REM HSL.BAS
20   GINIT
30   GCLEAR
40   KEY LABELS OFF
50   WINDOW 0,1.31,1.31,0
60   FOR L=0 TO 1 STEP .2
70     FOR H=0 TO 1 STEP 1/6
80       FOR S=0 TO 1 STEP .2
90         AREA COLOR H,S,L
100        MOVE H+.11*L,S+.11*L
110        RECTANGLE .09,.1,FILL,EDGE
120      NEXT S
130    NEXT H
140  NEXT L
150  LORG 7
160  MOVE 1.2,1.31
170  LABEL "z = Luminosity"
180  LORG 4
190  MOVE .6,1.31
200  LABEL "x = Hue"
210  LDIR PI/2
220  MOVE 1.31,.6
230  LABEL "y = Saturation"
240  END
```

When drawing an area in a certain color, it may be possible to produce the color faster and more accurately by specifying SET PEN followed by AREA PEN, rather than specifying AREA COLOR or AREA INTENSITY.

```
10   REM AREAPEN.BAS
20   GINIT
30   PLOTTER IS CRT,"INTERNAL";COLOR MAP
40   AREA INTENSITY 1/3,2/3,1
50   MOVE 10,50
60   RECTANGLE 20,20,FILL,EDGE
61   !
70   SET PEN 15 INTENSITY 1/3,2/3,1
80   AREA PEN 15
90   MOVE 40,50
100  RECTANGLE 20,20,FILL,EDGE
110  END
```

# FILL and EDGE Options

The FILL and EDGE options control the filling and edging of an area defined by the following statements: PLOT, IPLOT, RPLOT, POLYGON, RECTANGLE, or SYMBOL. If the FILL option is specified, the area is filled with the current area fill color and style. If the EDGE option is specified, the area is edged with the current line type and pen color. When both are specified the area is both filled and edged. If neither are specified, the area is just edged with the current line type and pen color.

The following program shows the effect that the FILL and EDGE options have on an area. To see their effect change line 50 to include the EDGE and then both the FILL and EDGE options.

```
10 GINIT
20 AREA PEN 2
30 LINE TYPE 4
40 MOVE 50,50
50 RECTANGLE 10,10, FILL
60 END
```

# Array Specified Pen Control

When large amounts of data are involved, or if you want to draw the same object at various places on the screen, it is often convenient to use arrays to describe the graphic coordinate values. In the PLOT, IPLOT, RPLOT, and SYMBOL statements you can specify a two or three column array. The first and second columns of the array specify the coordinate values. The optional third-column specifies the operation for each row of the array: pen-control, AREA PEN, AREA INTENSITY, LINE TYPE, PEN, FILL, and EDGE. If a two-column array is specified, the default pen control used on each row is a one, pen down after move.

This table shows the meaning of each column for each of the operations specified by column 3 of the array.

| Column 1 | Column 2 | Column 3 | Meaning |
|---|---|---|---|
| X value | Y value | < -2 | use even/odd pen control |
| X | Y | -2 | Pen up before moving |
| X | Y | -1 | Pen down before moving |
| X | Y | 0 | Pen up after moving |
| X | Y | 1 | Pen down after moving |
| X | Y | 2 | Pen up after moving |
| pen number | — | 3 | PEN |
| line type | repeat value | 4 | LINE TYPE |
| color | — | 5 | AREA INTENSITY |
| — | — | 6 | Start polygon mode w/FILL |
| — | — | 7 | End polygon mode |
| — | — | 8 | End of data for array |
| — | — | 9 | No operation, values ignored |
| — | — | 10 | Start polygon w/EDGE |
| — | — | 11 | Start polygon w/FILL & EDGE |
| — | — | 12 | Draw a FRAME |
| pen number | — | 13 | AREA PEN |
| red value | green value | 14 | AREA INTENSITY |
| blue value | — | 15 | AREA INTENSITY |
| — | — | > 15 | No operation, values ignored |

# AREA Color

Operation 5 in column 3 selects the AREA INTENSITY color. The column 1 value is divided into red, green, and blue numbers, each five bits in length (the sixteenth bit of column one is ignored). Each five-bit number specifies a value in the range zero to sixteen. This number is subtracted from sixteen to calculate the intensity value for each of the colors: red, green, blue. Intensities range in value from zero (darkest) to sixteen (most intense).

For example, if column 1 is set to zero, then each of the three groups in column 1 is set to zero. Sixteen minus zero yields sixteen for all three groups. Sixteen is full intensity, therefore, the area fill color will be white.

The following equation calculates the value for column 1 given R, G, B values in the range zero to one.

```
Column1 = 16-16*R + SHIFT(16-16*G,-5) + SHIFT(16-16*B,-10)
```

The AREA INTENSITY red, green, and blue values can also be selected with operations 14 and 15. The range of intensity is zero (no color) to 32,767 (full intensity). Operation 14 should be done before 15, and the operation takes effect when operation 15 is done.

# FILL and EDGE

A polygon is formed from a line sequence of 2 or more points with the optional FILL or EDGE specifiers. A polygon is drawn by plotting the first point, each successive point, and closed by drawing the final point back to the first point.

If FILL is specified, the polygon is filled with the current AREA fill color, and if EDGE is specified, the polygon is edged with the current PEN color. The array pen-control instructions supersede any other instructions on pen movement, LINE TYPE, and FILL and EDGE specifiers.

# Graphics Rotation

The PDIR and PIVOT statements cause rotations to be applied to graphic MOVEs and DRAWs. The angle specifies the direction and amount of rotation. It is measured in a counter-clockwise direction from the positive X-axis. The current trigonometric mode (RAD or DEG) determines the units for the angle. The default trigonometric mode is RAD.

# PDIR Statement

The PDIR statement specifies the rotation of IPLOT, RPLOT, POLYGON, POLYLINE, and the RECTANGLE statements. The rotation takes place about the logical position. The AXES, GRID, LABEL and SYMBOL statements are not affected by PDIR.

# PIVOT Statement

The PIVOT statement causes the rotation of all lines, except those generated by the AXES, GRID, LABEL and SYMBOL statements. This includes lines generated by the MOVE, DRAW, IMOVE, IDRAW, PLOT, IPLOT, RPLOT, POLYGON, POLYLINE, and RECTANGLE statements. This rotation takes place about the logical position. The PIVOT statement effects only the starting point of the LABEL and SYMBOL statements.

If both the PIVOT and PDIR angles are set, they both have an effect on the lines being generated. The transformation on the statements affected by the PDIR commands takes place first. Then the transformations on the lines affected by the PIVOT command are done. A few examples will help demonstrate this.

```
10 GINIT
20 DEG
30 DRAW 50,50
40 PIVOT 0
50 PDIR 0
60 POLYGON 10,10,8
70 IDRAW 10,10
80 END
```

No rotations are done since both angles are set to zero. Now change line 50 to have a rotation of 90 degrees.

```
50 PDIR 90
```

The rotation of 90 effects the POLYGON statement, but not the IDRAW. This is because PDIR only affects the IPLOT, RPLOT, POLYGON, POLYLINE, and RECTANGLE statements. Now change line 40 to have a rotation of 45 degrees.

```
40 PIVOT 45
```

The POLYGON statement is rotated a total of 135 degrees, by the combination of the PIVOT 45 and the PDIR 90 statements. You will notice that the IDRAW line has been rotated only 45 degrees by the effect of the PIVOT 45 statement. Now change line 50 back to 0 degrees.

```
50 PDIR 0
```

The IDRAW line is still rotated by 45 degrees. The POLYGON statement is rotated by 45 degrees as a result of the PIVOT angle. Hopefully this example will clear up some of the confusion when applying both the PIVOT and PDIR transforms to graphic lines.

# Screen Raster Images

The GSTORE statement allows you to save the current graphics screen raster image into an array and the GLOAD statement restores the graphics image to the graphics screen. The device-selector specifies the device, which must be a bit-mapped device. The CRT is assumed if no device selector is specified.

There are many uses for this capability. Images of various graphs and charts can be stored in different arrays, and then easily redisplayed. Graphs that may be very slow to generate, need only to be generated once and then displayed very fast. The graphics information in the arrays can also be written to a file for future use.

Two forms of the GLOAD and GSTORE statements are supported. The first form is compatible with the GLOAD/GSTORE statements in HP BASIC, and displays an image that fills the entire screen. The second form operates on an arbitrary sized rectangular portion of the screen. For users porting programs from HP BASIC that use the Bstore()/Bload() CSUBs supplied with HP BASIC, Chapter 10 of the *Installing and Using Guide*, presents Bstore()/Bload() SUBs that call GSTORE and GLOAD using the integrated syntax.

# Full Screen

The size of the array necessary to store a complete screen image for each display depends on the resolution and on the number of colors the display supports. GESCAPE CRT,3 can be used in a program to determine the size necessary:

```
10     INTEGER S(5)
20     GESCAPE CRT,3;S(*)
30     PRINT "Array size is";S(4);"rows and";S(5);"columns."
40     END
```

The following table gives the sizes for some PC display adapters. The array may be declared larger or smaller than the size given. If the array is not large enough to contain a full screen image, GLOAD stops when all the array contents have been transferred. If the array is too large, only part of the array will be used. If an attempt is made to GLOAD an image to a display that is different from the GSTORE display, unpredictable results will occur. If the color map has different values than when the image was GSTOREd, the colors will not match the original image.

| Display | Array Size |
|---|---|
| 640x480x16 | Image(1:160,1:480) |
| 800x600x16 | Image(1:200,1:600) |
| 1024x768x16 | Image(1:256,1:768) |
| 640x480x256 | Image(1:320,1:480) |
| 800x600x256 | Image(1:400,1:600) |
| 1024x768x256 | Image(1:512,1:768) |

Below is an example of the GSTORE and GLOAD statements for a 1024 x 768 x 256 screen. Change line 10 to the appropriate size for your resolution.

```
10 INTEGER Image(1:512,1:768)
20 FRAME
30 DRAW 50,50
40 POLYGON 10,10,FILL
50 GSTORE Image(*)
60 END
```

Now that the image is stored in the array, clear the graphics screen with the GCLEAR statement. To load the image back onto the screen, use the following statement:

```
GLOAD Image(*)
```

**Rectangular Blocks**

```
GSTORE Image(*),Width,Height,Rule,Xorigin,Yorigin
```

When a *Width* and *Height* are specified after the image array, only a rectangular block is transferred between the array and the display. *Width* and *Height* are specified in pixels. Optionally, a *Rule* can be specified that instructs GLOAD/GSTORE how to combine the contents of the array with the contents of the screen. *Presently, only a value of 3 is supported.* This causes the contents of the array (for GLOAD) or screen (for GSTORE) to totally overwrite the target. The block will be located with the upper left corner at the current graphic position. Alternately, a position can be specified with the *Xorigin*, *Yorigin* parameters. These parameters should be specified in the current WINDOW units, not pixels or VIEWPORT units (GDUs).

For displays with 8 planes or less (256 colors or less), the image is stored with one byte per pixel. This makes images somewhat transportable among different displays. It also means that the number of elements necessary to store the image is equal to Width*Height/2. If the width is even, the array could be declared as

```
INTEGER Image( 1:Width/2,1:Height)
```

If the array is too small, an error is given. If the array is too large, the extra array elements are ignored. If GLOAD is used to display an image on a display with less colors than the GSTORE display, the results are undefined. If the color map is different than the color map in effect when the image was GSTOREd, the colors will not match the original image. The format of the data in the array for full screen images is stored as a bitmap.

# Screen Dumps

The DUMP ALPHA and DUMP GRAPHICS statements and function keys copy the contents of either the ALPHA or the GRAPHICS screen to the DUMP DEVICE IS printing device.

# DUMP ALPHA Statement

The DUMP ALPHA statement sends alphanumeric characters compatible with any ASCII printer to the current DUMP DEVICE IS device. If the graphics are visible when this command is executed, an ALPHA ON is executed before the screen dump occurs. This eliminates the graphics from the information sent to the printer.

# DUMP GRAPHICS Statement

The DUMP GRAPHICS statement sends graphics information to the printer or to a file in the current printer language. The *Installing and Using* manual explains how to set the printer language in Chapter 7, "Printer and Pixel Image Device Drivers."

If the ALPHA and GRAPHICS screens are MERGEd, then the ALPHA text will also be dumped to the printer as part of the graphics data. If the graphics are not visible when this command is executed, a GRAPHICS ON is executed before the screen dump occurs. To remove the run indicator character in the bottom-right corner of the screen, use RUNLIGHT OFF before dumping the screen image.

# DUMP DEVICE IS Statement

The DUMP DEVICE IS statement specifies which device, file, or pipe receives the data when a DUMP ALPHA or a DUMP GRAPHICS statement is executed without a device selector. The GINIT statement causes the default DUMP DEVICE IS to be set to the value of PRT. If the optional EXPANDED keyword is included, the image is rotated by 90 degrees. For example:

```
DUMP DEVICE IS PRT,EXPANDED
```

If APPEND is specified and the DUMP is to a file, the file position is moved to the end-of-file before each DUMP. For some DUMP types, multiple images in a file are not supported. If APPEND is specified in these cases, the result is undefined. If APPEND is not specified, the file is overwritten with each DUMP.

The output can be sent to a device (usually a printer) or a file. If the destination is a file, it must be an ordinary file or a BDAT file.

Many computer displays and most printers do not have square pixels. This results in a distortion when the graphics image is printed. This is normal and can be partially compensated for, if needed, by adjusting the WINDOW to apply an inverse distortion to the image drawn on the display. HTBasic partially compensates for non-square pixels by printing more than one printer pixel for each display pixel in some instances.

In some cases the softkey labels and ALPHA text are dumped along with GRAPHICS by the DUMP GRAPHICS statement. This depends on whether ALPHA and GRAPHICS are MERGEd or SEPARATE. MERGE and SEPARATE ALPHA are explained later in this chapter.

This problem can be easily solved using the statement SEPARATE ALPHA FROM GRAPHICS before generating the plot. If you have not used SEPARATE ALPHA, to solve the problem you should clear the ALPHA screen, turn the runlight off, and turn the softkey labels off before starting the GRAPHICS plot. For example:

```
KEY LABELS OFF
RUNLIGHT OFF
CLEAR SCREEN
```

# Partial Screen Dumps

GESCAPE code 106 specifies a portion of the display screen to dump. The syntax is:

```
GESCAPE PRT,106,param(*)
```

The *param* array must be a one dimensional INTEGER array of five elements. The first element is the operation number (1 = sets boundaries for the DUMP commands). The remaining elements specify the boundary for the DUMP. The boundary is specified in screen units.

```
param(2) - Beginning row
param(3) - Ending row
param(4) - must be zero
param(5) - must be zero
```

The CONFIGURE DUMP, PLOTTER IS CRT,"INTERNAL", and GRAPHICS INPUT IS KBD,"KBD" commands reset the row parameters back to the defaults, full screen. The CONFIGURE DUMP command must be executed before the GESCAPE command. The following program demonstrates this capability:

```
10   INTEGER A(1:5)
20   A(1)=1   ! select code-always 1
30   A(2)=100 ! begin row-screen units
40   A(3)=300 ! End_row-screen units
50    A(4)=0   ! reserved
60   A(5)=0   ! reserved
70   DUMP DEVICE IS PRT
80   CONFIGURE DUMP TO "PCL"
90   GESCAPE PRT,106,A(*)
100  FRAME
110  MOVE 0,0
120  DRAW 480,480
130  DUMP GRAPHICS
140  END
```

# Graphics Devices

The default graphics output device is the CRT display attached to your computer. The default graphics input device is the keyboard. If a mouse is installed and active, it may also be used to specify graphic input. HTBasic allows you to specify alternate graphic devices for both input and output. The graphic output plotter language may also be specified. You can also send plotter commands directly to the output graphics device.

# PLOTTER IS Statement

The PLOTTER IS statement specifies the graphics output device or file. The default graphics output target is the CRT. Executing a PLOTTER IS statement directs all subsequent graphics output to the specified device or file. For example:

```
PLOTTER IS CRT,"INTERNAL";COLOR MAP
```

The plotter specifier string specifies the graphics device driver. To re-select the internal CRT graphics display specify "INTERNAL".

The "HPGL" plotter specifier string specifies the HP graphics language. This specifier can be used in conjunction with a plotter or disk file. If a file is specified, you must also specify the physical device (hard-clip) limits. If a device is specified, the device must be able to return the physical device limits when initialized. For example:

```
PLOTTER IS 705,"HPGL"
```

specifies an HPGL plotter connected to the IEEE-488 bus on address five. The plotter can be connected to the computer over the IEEE-488 bus or the serial interface.

The hard-clip limits of the plotter may be specified. If they are not and output is going to a file, they default to ±392.75 mm in the x axis and ±251.5 mm in the y axis. If hard-clip limits are not specified and output is going to a device, HTBasic asks the device to return the p-points. The hard-clip units are specified using plotter units equal to 0.025 millimeters. The file is positioned to the beginning and is closed when another PLOTTER IS, GINIT, or SCRATCH A statement is executed. For example:

```
PLOTTER IS "Pictfile","HPGL",5,250,7,136
```

The file can be later sent to a plotting device or imported into a word processor or a desktop publishing package that supports the HP graphics language. To reset your graphic output device to the internal display execute the command:

```
PLOTTER IS CRT,"INTERNAL"
```

# GSEND Statement

The GSEND statement sends command strings to the PLOTTER IS device. It is used to send commands (such as the pen speed, pen force, and character set selection commands) that are not generated by the high level graphics statements. For example:

```
GSEND "LBPlotter font characters."&CHR$(3)
```

sends the HPGL command to draw the specified string using the current plotter internal character font. Virtually any HPGL command string can be sent to the device using this statement.

# GRAPHICS INPUT IS Statement

The GRAPHICS INPUT IS statement is used to specify the graphic input device. See the *Installing and Using* manual, Chapter 8, "Graphic Input Drivers" for more information. The default graphics input device is the keyboard. The mouse may also be used for graphic input. An example GRAPHICS INPUT IS statement is:

```
GRAPHICS INPUT IS 702,"HPGL"
```

To perform graphic input, the following statements are used:

```
DIGITIZE X,Y,Status$     !Wait for Point
READ LOCATOR X,Y,Status$ !Immediate Return Point
```

Here, X and Y are the target variables for the graphic position, and Status$ is an optional string that receives information about the state of the GRAPHICS INPUT IS device. The returned coordinates are in the units defined in the current WINDOW or SHOW statement. The Status$ string contains eight bytes with the following information:

| Byte | Meaning |
|---|---|
| 1 | Indicates End of Stream for a device supporting continuous point stream digitizing. Byte 1 may be used as the pen control value in a PLOT. It is "0" if it is the last of a continuous point stream. It is "1" otherwise, including points from a device supporting only single point digitizing. |
| 2 | Comma delimiter character. |
| 3 | Clip Indicator - If the character is a "0", then the point is outside the hard-clip limits. If a "1", the point is inside the hard-clip limits, but outside the soft-clip limits (see CLIP). If a "2" then it is inside the soft-clip limits. |
| 4 | Comma delimiter character. |
| 5 | Tracking ON/OFF - If the character is a "0", then tracking is off; if a "1", then tracking is on. |
| 6 | Comma delimiter character. |
| 7-8 | Button Positions. If S$ is the status string and B is the button number you wish to test, then BIT(VAL(S$[7,8]),B-1) returns one if B is down, and zero if B is up. |

A point is digitized and the coordinates of that point are assigned to the variables when the keyboard ENTER or CONTINUE keys, a mouse button, or a digitizer button is pressed.

# READ LOCATOR Statement

The READ LOCATOR statement immediately reads the graphic position and stores it into the X and Y variables without waiting for a DIGITIZE operation.

# SET LOCATOR Statement

The SET LOCATOR statement establishes a reference point for any subsequent graphics input statements. SET LOCATOR is only valid for graphics input devices that use relative locators.

# WHERE Statement

The WHERE statement returns the current logical pen position in the x and y numeric variables and pen status information in the optional string variable. For example:

```
WHERE X,Y,Status$
```

The Status$ string contains three bytes with the following information:

| Byte | Meaning |
|---|---|
| 1 | Pen Status - Up/Down status of the Pen. If the character is a "1" then pen is down; if it is a "0" then the pen is up. |
| 2 | Comma delimiter character. |
| 3 | Clip Indicator - If the character is a "0", then the point is outside the P1, P2 limits. If a "1", the point is inside the P1, P2 limits, but outside the viewport. If a "2" then it's inside the viewport. |

# Tracking Graphics Input

The SET ECHO and the TRACK statements allow you to follow the movements of the graphics input device on the PLOTTER IS device.

# SET ECHO Statement

The SET ECHO statement displays a tracking cross on the screen or moves the plotter pen to the specified location. If the location is outside the clipping boundaries no action takes place.

# TRACK and DIGITIZE Statements

The TRACK statement controls tracking of the input device. It enables and disables the graphic locator from following the input device position on the PLOTTER IS device during DIGITIZE statements. Tracking stops when a point is digitized, and the tracking cross is left at the location of the digitized point. When the display device is a plotter, the pen position tracks the input device. When it is the CRT, the tracking cross tracks the input device.

Use the arrow keys on the keyboard or the mouse to move the tracking cross around. Run the following two programs to get a feel for how the DIGITIZE and READ LOCATOR statements differ. Also look at how the TRACK and SET ECHO statements can be used to follow the input device on the output device. Press the ENTER key or one of the mouse buttons to read the current X,Y location in program #1. Enter the "STOP" command to terminate program #2.

**Program #1**

```
10 GINIT
20 PLOTTER IS CRT,"INTERNAL"
30 GRAPHICS INPUT IS KBD,"KBD"
40 FRAME
50 TRACK CRT IS ON
60 DIGITIZE X,Y,S$
70 PRINT X,Y,S$
80 END
```

**Program #2**

```
10 GINIT
20 PLOTTER IS CRT,"INTERNAL"
30 GRAPHICS INPUT IS KBD,"KBD"
40 FRAME
50 READ LOCATOR X,Y,S$
60 SET ECHO X,Y
70 GOTO 50
80 END
```

# Mixing Output and Input Devices

As shown by the following examples you can do some interesting things with the DIGITIZE statement by mixing the possible PLOTTER IS and GRAPHICS INPUT IS devices. The first example uses the default PLOTTER and GRAPHICS devices. Use the mouse to move the tracking cross around. Press the ENTER key or one of the mouse buttons to read the current tracking cross location.

```
10 GINIT
20 PLOTTER IS CRT,"INTERNAL"
30 GRAPHICS INPUT IS KBD,"KBD"
40 TRACK CRT IS ON
50 FRAME
60 DIGITIZE X,Y,S$
70 PRINT X,Y,S$
80 END
```

If you have an HPGL plotter hooked up to your computer then you can try the following example. Change line 30 in the above program to setup the plotter as the input device. Chose one of the following lines for the type of communication that your plotter uses. If you are using the serial interface, make sure that the baud rate, parity, stop bits, and handshaking are setup correctly. For an IEEE-488 plotter, the device address needs to be set correctly.

```
30 GRAPHICS INPUT IS 9,"HPGL"   !Serial Interface
30 GRAPHICS INPUT IS 705,"HPGL" !IEEE-488 Address 5
```

When this program is run, the tracking cross will appear on the CRT display. By moving the pen around on the plotter, you will see the tracking cross on the screen follow the plotter movement.

You can also experiment with the other alternatives by setting the PLOTTER device to the HPGL plotter, and then varying the GRAPHICS device between the keyboard and the HPGL plotter.

# Separate and Merged Alpha

The SEPARATE ALPHA FROM GRAPHICS and MERGE ALPHA WITH GRAPHICS statements provide control over how ALPHA text and graphics are displayed and manipulated on the computer screen. Both statements are presented in the following section.

# Merged Alpha

The MERGE ALPHA WITH GRAPHICS statement causes all bit-planes to be used by both alpha and graphics. Alpha text is converted to graphic pixels and written into the graphic planes, overwriting any graphics data that might be present. Also, scrolling alpha text will scroll graphics, dumping either will dump both and the full range of colors are available for both alpha text and graphic output.

# Separate Alpha

The SEPARATE ALPHA FROM GRAPHICS mode is the opposite of MERGE ALPHA WITH GRAPHICS. When separate, one or more bit plane is reserved for alpha text and the remaining planes are reserved for graphic output. The alpha and graphic planes can then be turned on or off or DUMPed independently. However, ALPHA text color and graphic pens are limited as shown in the following table.

The following table shows the colors available when SEPARATE ALPHA FROM GRAPHICS is used, depending on the total number of colors available.

| Total Colors | Graph Pens | Black Alpha | White Alpha | Brown Alpha | Cyan Alpha |
|---|---|---|---|---|---|
| 16 | 0-7 | 0 | 8 | - | - |
| 256 | 0-63 | 0 | 64 | 128 | 192 |

# Porting Issues

HP BASIC assigns green to the first pen; HTBasic assigns white. If you prefer green or some other color, you must explicitly set a range of pen values to the color desired. The range starts with the white alpha pen value from the table above and continues to one less than the value of the brown alpha pen value. For 16 color systems, the last value should be 15. For example, the following code changes the alpha pen from white to green on a 16 color display:

```
10 SEPARATE ALPHA FROM GRAPHICS
20 PLOTTER IS CRT,"INTERNAL";COLOR MAP
30 FOR I=8 TO 15
40   SET PEN I INTENSITY 0,1,0
50 NEXT I
60 END
```

# Summary

HTBasic contains an extensive assortment of powerful graphic statements. You may use convenient data units in defining your graphic display and you may specify many graphic attributes. A detailed examination of the graphic capabilities requires the user to try some examples and see how the different graphic statements are used in concert to produce high quality graphics output.

# General Input and Output

This chapter discusses the general I/O (input/output) statements of HTBasic. General I/O statements apply equally to screen, keyboard, printer, files, interfaces, devices, strings, and buffers. The chapters following this one discuss these I/O destinations/sources in greater detail. You should read the information in this chapter first, and then read any of the following chapters you need. BEEP, READ, and DATA statements are also presented in this chapter.

There are two pairs of statements that are used in general I/O: ENTER/OUTPUT and STATUS/CONTROL. Some interfaces also support interrupts, which can be used to force program branching on different interface conditions. Some interfaces also support background transfers. The TRANSFER statement starts background transfers.

Associated with each I/O operation is an I/O path. An I/O path contains all of the routing information necessary for the computer to transfer data between your HTBasic program and the target entity (such as a printer, data acquisition device, string, file, etc.). The ASSIGN statement is used to set up an I/O path for use in later ENTER, OUTPUT, and TRANSFER statements.

# ASSIGN Statement

The ASSIGN statement is similar to the OPEN statement of other computer languages. ASSIGN makes a connection to a screen, keyboard, printer, file, interface, device, or buffer. All the information concerning this connection is saved by the ASSIGN statement in an I/O path variable. A number of attributes can be specified in the ASSIGN statement that affect how the I/O operation is done.

The I/O path variable is then used in I/O statements to specify the source or destination of the ENTER, OUTPUT, or TRANSFER. After the initial ASSIGN, subsequent ASSIGN statements can be used to redirect the I/O, change the attributes, or close the file or connection. Several I/O paths can be set up simultaneously. In fact, any number of I/O path variables may exist in your program, although some operating systems limit the number of files that can be open at one time.

# Syntax

The syntax of the ASSIGN statement is:

```
ASSIGN @io-path [TO target] [;attrib [,attrib...] ]

where:

@io-path = a legal I/O path variable name
target =
    device-selector [, device-selector...] |
    file-specifier | pipe-specifier |
    BUFFER {string-name$ | numeric-array(*) | ‘[‘buf-size’]’}
attrib =
    FORMAT {ON|OFF|MSB FIRST|LSB FIRST} |
    {BYTE|WORD} |
    EOL eol-chars [END] [DELAY seconds] |
    EOL OFF |
    APPEND |
    RETURN numeric-name
buf-size = size of the buffer in bytes
eol-chars = string expression of up to 8 characters
seconds = numeric-expression rounded to the nearest
    0.001 through 32.767 (default is 0)
```

The following paragraphs give some explanation of how to ASSIGN the different I/O targets: devices, files, and buffers. Then the different I/O attributes are explained.

# Devices

A device can be the screen, the keyboard, a printer, lab instrument, or data acquisition device. The device is specified with an interface select code (ISC) or device selector. Each interface, (IEEE-488, serial, parallel, etc.) that is connected to your computer has a unique number assigned to it. When you load a device driver for an interface, a default ISC is assigned, or you can specify another ISC. The following table gives some default ISCs.

| ISC | Device |
|---|---|
| 1 | CRT display |
| 2 | Keyboard |
| 3 | Graphic display |
| 6 | Bit mapped graphic display |
| 7 | IEEE-488 Board |
| 8 | 2nd IEEE-488 Board |
| 9 | Serial |
| 10 | Windows default printer via Print Manager |
| 11 | 2nd Serial |
| 12 | GPIO Board |
| 17 | Various Data Acquisition Boards (no analog capabilities) |
| 18 | Various Data Acquisition Boards (with analog capabilities) |
| 26 | Parallel port |
| 32 | Processor |

If multiple devices can be hooked to the interface simultaneously, as they can on the IEEE-488, then the primary address must be included with the ISC to uniquely identify the device. This is also true of data acquisition boards having one or more subsystem: A/D, D/A, DIO, etc. Together, the ISC and primary address are called a device selector. Some IEEE-488 devices also require one or more secondary addresses. Each primary or secondary address should be specified with two digits. Thus 1 should be specified as 01. The total length of the device selector can be 15 digits. The following examples illustrate these rules. To perform I/O with an IEEE-488 device (assuming the default ISC) at primary address 2 and secondary address 6, use this ASSIGN statement:

```
ASSIGN @Dvm TO 70206
```

To perform I/O with an IEEE-488 device at primary address 3:

```
ASSIGN @Scope TO 703
```

To use analog output with a data acquisition board set to ISC 18 and 02 as the primary address for analog output, use this ASSIGN:

```
ASSIGN @Daq TO 1802
```

To performI/O with a device hooked to a serial port at ISC 9, you could use:

```
ASSIGN @Dvm TO 9
```

A device can have more than one I/O path name, each with different attributes, associated with it.

An I/O path name can have more than one IEEE-488 device assigned to it. If multiple devices are specified, they must be on the same interface. When OUTPUT is made to an I/O path assigned to multiple devices, all the devices receive the data. When ENTER is made from multiple devices, the first device specified sends data to the computer and to all the other devices assigned to the I/O path name. When CLEAR, LOCAL, PPOLL CONFIGURE, PPOLL UNCONFIGURE, REMOTE, or TRIGGER are made on multiple devices, all the devices receive the IEEE-488 message.

It is possible to perform I/O with a device without using an I/O path. But, when an I/O statement does not specify an I/O path variable, a temporary I/O path is created internally, used for the duration of the statement, and then discarded. This is usually slower than to ASSIGN an I/O path once and use it throughout the program.

```
OUTPUT @Scope;A(*)     ! Usually faster
OUTPUT 703;A(*)        ! Usually slower
```

# Files

A file is opened when the ASSIGN statement specifies a file-specifier. The file's position pointer is set to the beginning if the APPEND option is not specified and set to the end if it is. The file position is updated to point to the next byte to be written or read after each ENTER or OUTPUT statement. The ASSIGN statement will not CREATE a file if it does not exist. You should use the CREATE statement before ASSIGNing the file if the file does not yet exist.

```
CREATE "Jonatha.n",1
ASSIGN @File TO "Jonatha.n"
```

# Pipes

Pipes are not supported by HTBasic.

# Buffers

Buffers are typically used as the source or destination of a TRANSFER. The statement

```
ASSIGN @Iopath TO BUFFER [300]
```

allocates an unnamed buffer and assigns it to an I/O path name. Unnamed buffers can only be accessed through their I/O path. The

```
ASSIGN @Another TO BUFFER X(*)
```

statement assigns an I/O path name to the variable X(*) which must be declared as a buffer in a COM, DIM, INTEGER, LONG or REAL statement. Numeric data stored in a named buffer should not be accessed through the name of the array if the byte order of the computer and the byte order of the data is different. In general, STATUS and CONTROL are the preferred method for accessing the data.

The buffers specified in these ASSIGN statements may now be used in ENTER, OUTPUT, or TRANSFER statements.

# Attributes

The attributes of an I/O path allow you to change certain aspects of how data is transferred. The attributes can be specified when the initial ASSIGN is made, or the attributes of a previously ASSIGNed I/O path may be individually changed by omitting the "TO target" portion of the statement:

```
ASSIGN @Jennifer;FORMAT OFF
```

Additional attributes of a particular device, such as the baud rate of a serial device, are changed using STATUS, CONTROL, READIO, and WRITEIO statements. These statements are explained later in this chapter.

# FORMAT Options

One piece of information stored in the I/O path is whether to transfer information in ASCII or binary (internal) format. ASCII transfers are called FORMAT-ON-format and binary transfers are called FORMAT-OFF-format. If FORMAT is not explicitly specified in the ASSIGN statement, a default format is used. For interfaces, buffers, devices, and LIF ASCII files the default is FORMAT ON; for BDAT and ordinary files, the default is FORMAT OFF.

This example explicitly specifies FORMAT ON:

```
10   ASSIGN @George TO "TEMP.TXT";FORMAT ON
```

When FORMAT ON is specified in the ASSIGN statement, data items are output in a readable ASCII format. Numeric items are output in the standard ASCII numeric format and the ASCII characters in a string are output. If the default output formats are not acceptable, the USING and IMAGE statements can be used to format the data as needed.

When FORMAT ON is specified, data items are entered with the data expected to be in readable ASCII format. Reading data with FORMAT ON works with most devices. For other devices, most formats can be handled using IMAGE and USING statements. Numeric data must be scanned to find legal combinations of characters that make up a numeric value. String data must be scanned for end-of-string terminators.

When FORMAT OFF is specified in the ASSIGN statement, data is transferred in internal format. LSB/MSB FIRST can be used in the ASSIGN statement to specify the order in which the data bytes are sent or received. If LSB/MSB FIRST is not specified, data sent to devices is sent MSB FIRST for compatibility with HP devices; data sent to files, and operating system devices is stored in the form most natural to the computer's processor. Of course, LSB/MSB FIRST can always be used to override these defaults.

```
20   ASSIGN @Dev1 TO 9;FORMAT LSB FIRST
```

The internal format for **INTEGER** numbers is a two byte, two's complement, binary integer. The internal format for **LONG** (integer) numbers is four byte, two's complement, binary integer. The internal format for **REAL** numbers is an eight byte, IEEE compatible floating point number (see *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std. 754-1985). The internal format for **COMPLEX** numbers is two real numbers. The first eight bytes contain the real part of the number and the last eight bytes contain the imaginary part.

The format for **strings** depends on the source or destination of the I/O statement. For ordinary files the internal format for strings is a null-terminated string.

The string format for devices and BDAT files consists of a four-byte-integer string length followed by the string contents. An even number of bytes is always transferred, therefore, if the string is odd in length an extra pad byte is transferred. The LSB/MSB FIRST option can be used to determine the byte ordering of the four byte string length.

For LIF ASCII files FORMAT OFF has no effect. Data is always transferred as ASCII strings proceeded by a two byte integer length and padded by a trailing space if necessary to make the string length even. The string length is always transferred with MSB FIRST.

# BYTE and WORD

When BYTE is included in the ASSIGN statement, the data is sent and received as bytes, even if the interface is capable of 16-bit transfers. The upper 8 bits are zeros. (Note: The "W" IMAGE specifier will send a word on a 16-bit interface, even if BYTE is specified in the I/O path.) The WORD attribute can only be included in the ASSIGN statement if the interface is capable of 16-bit transfers. The data is then transferred a word at a time. The WORD attribute is primarily intended for transfer of FORMAT OFF data and can have unexpected side effects with FORMAT ON.

When neither BYTE nor WORD is included in the ASSIGN statement, the default is BYTE. Syntax examples including the BYTE and WORD attributes are:

```
30   ASSIGN @Gpio TO 15;FORMAT OFF,WORD
40   ASSIGN @Meter TO 711,712;BYTE
```

# EOL

The default End-Of-Line for OUTPUT is a carriage return (CR) followed by a line feed (LF) sent with no END indication and followed by no DELAY. Specifying END causes an interface specific END indication to be sent with the EOL. On the IEEE-488, END causes EOI to be sent with the final character of the EOL. Specifying DELAY causes the computer to pause for the specified number of seconds after sending the EOL and before allowing the program to continue. The exact delay time depends on the timing resolution available on the computer you are using. On the PC, the timing resolution under Windows 95/98/ME is about 55 milliseconds under Windows NT & Windows 2000 it is about milliseconds. The default EOL (CR/LF) can be restored by specifying EOL OFF.

Examples:

```
50   ASSIGN @Slow_printer TO 10;EOL CHR$(10) DELAY .1
60   ASSIGN @Jet TO 701;EOL CHR$(13)&CHR$(10) END
70   ASSIGN @Ep TO PRT; EOL OFF ! Restore default
```

# APPEND

If APPEND is specified, the file position is moved to the end-of-file after the ASSIGN. If it is not specified, the file position is moved to the beginning of the file. APPEND is supported on BDAT and ordinary files, but not LIF ASCII files.

# RETURN

RETURN can be used with ASSIGN to test whether the ASSIGN operation was successful. If not successful, the error number is returned in the variable specified, otherwise a zero is returned. The following example shows one possible use for this feature: ASSIGNing a file, but creating the file only if it does not exist:

```
10    INTEGER Result
20    F$="KAREN.TXT"
30    REPEAT
40      ASSIGN @I TO F$;RETURN Error
50      IF Error THEN
60        IF Error=56 THEN
70          PRINT "File not found: creating ";F$
80          CREATE F$,10
90        ELSE
100         CAUSE ERROR Error ! Let the error occur
110       END IF
120     END IF
130   UNTIL NOT Error
140   END
```

# Closing an I/O Path

Closing an I/O path makes the path invalid. All subsequent ON event statements for the closed I/O path are not acted upon. If an I/O path name has not been declared in a COM statement it may be closed in the following ways:

1. Explicitly close a path by executing

```
ASSIGN @Io-path TO *
```

2. Re-assigning the I/O path

```
ASSIGN @Io-path TO target
```

3. Exiting the subprogram with SUBEND, SUBEXIT, ON...RECOVER, or RETURN *value*

4. Stopping the program with END, GET, LOAD, SCRATCH, SCRATCH A, SCRATCH C, or STOP.

If an I/O path name has been declared in a COM statement it may be closed in the following ways:

1. Explicitly close a path by executing

```
ASSIGN @Io-path TO *
```

2. Re-assigning the I/O path with

```
ASSIGN @Io-path TO target
```

3. Executing SCRATCH A, SCRATCH C, SUBEND, SUBEXIT, ON...RECOVER, or RETURN *value*.

4. Executing EDIT, GET, or LOAD in a program that has a COM statement that does not match the COM statement that contains the I/O path name.

# OUTPUT Statement

OUTPUT can be used to send numeric data, array elements, character strings, sub-strings, or full arrays. Full arrays are specified with the full array specifier, "(*)", and are output in row major order. Row major order means the right-most subscript is incremented before the subscript to the left of it. If the default output formats are not acceptable, USING/IMAGE can be used to format the data as needed. If the data is sent with FORMAT OFF, then just the internal representations are sent, with no item separators. If the data is output with FORMAT ON, then data is sent in ASCII, printable characters. Numeric and string items are sent as described below.

# Numeric Items

Numeric items are converted to standard numeric format: The number is rounded to twelve significant digits. If the absolute value is outside the range 1E-4 to 1E+6, then the number is output in scientific notation. If the number is negative, a leading minus sign is sent; if the number is positive, a leading space is sent instead. Trailing zeros or decimal points are not output. If a comma follows the output item, then a comma is sent as an item separator. If a semicolon follows the output item, then no item separator is sent. If a full array is sent, a comma will be sent to separate each element unless a semicolon follows the output item (the array).

Complex items are output in rectangular form, real part first, then a comma, and finally, the imaginary part. Real and imaginary parts are formatted in standard numeric format as explained in the previous paragraph. If a semicolon follows the complex item then the comma separating the real and imaginary parts is suppressed.

# String Items

String items are sent by sending each character in the string. If a comma follows the output item, then CR/LF are sent as an item separator. If a semicolon follows the output item, then no item separator is sent. If a full array is sent, an item separator will be sent to separate each element unless a semicolon follows the output item (the array).

**Note:** CR/LF is the string item separator and is not affected by the EOL sequence definition in the ASSIGN statement.

# End of Line

After all of the OUTPUT items have been sent and if the statement did not end with a semicolon or comma, then an End-of-Line (EOL) Sequence is sent. The default EOL is CR/LF, but may be redefined in the ASSIGN statement.

The following examples illustrate most of these rules. For the examples, assume that the following lines have first been executed:

```
10    REM OUTPUTEX.BAS
20    DIM R(1),A$(1)[1]
30    R(0)=-1
40    R(1)=+1
50    MAT A$=("A")
60    ASSIGN @I TO some device
```

In the Characters Output, the EOL (which defaults to CR/LF), is shown as <eol>, the string item separator (which is CR/LF), is shown as <crlf>. Spaces are shown as underlines, "_", to make them stand out.

| Program lines | | Characters output |
|---|---|---|
| 70 | OUTPUT @I;1.E+5,1.E+7 | _100000,_1.E+7<eol> |
| 80 | OUTPUT @I;1;-1 | _1-1<eol> |
| 90 | OUTPUT @I;R(*), | -1,_1, |
| 100 | OUTPUT @I;CMPLX(1,1.23456789012345E+7) | _1,_1.23456789012E+7<eol> |
| 110 | OUTPUT @I;CMPLX(1,1); | _1_1 |
| 120 | OUTPUT @I;"B";"C","D" | BC<crlf>D<eol> |
| 130 | OUTPUT @I;A$(*); | AA |

In line 70, 1.E+7 is listed in scientific notation because it is outside the range 1E-4 to 1E+6. The numbers are both positive so a leading blank is output before each. The default EOL is output at the end. In line 80, the -1 has no leading space; the space is replaced with a negative sign. And the semicolon between the two numbers suppresses the comma that would normally be output between the two.

In line 90, an array is output. Notice the full array specifier. The comma following the array causes an item separator to be output between each element. The numeric item separator is a comma. Line 120 shows an example of a string item separator, CR/LF. Line 90 also shows that a trailing comma will suppress the EOL.

Line 100 shows that the real and imaginary parts of a complex number are separated by a comma. This line also shows the default rounding of numbers to 12 significant digits. Remember that if the default output formats are not suitable for your application, you may use OUTPUT USING to define your own. Line 110 shows that a semicolon will suppress the comma between the real and imaginary parts. It also shows that a trailing semicolon will suppress the EOL.

Line 120 demonstrates the effect on string output of commas and semicolons. The semicolon suppresses the item separator that would normally follow the "B". The comma causes the normal string item separator, CR/LF, to be output after the "C". And if no semicolon or comma is at the end of the statement, an EOL is output after the final string.

# END

An optional END may be used after the last data item. If USING is not specified, then END: 1) suppresses the EOL sequence from being output after the last item, 2) sends an EOI signal with the last character of the last item sent to an IEEE-488 device, and 3) truncates a file. A comma before the END will output an item terminator (a comma for numeric items or a CR/LF for string items). For example:

```
OUTPUT @I;"Time: ";TIME$(TIMEDATE) END
```

# ENTER Statement

The ENTER statement can be used to enter numeric data, array elements, character strings, sub-strings, or full arrays. Full arrays are specified with the full array specifier, "(*)", and are entered in row major order. If the default ENTER formats are not acceptable, USING/IMAGE can be used to format the data to the specifications of the I/O device. If the ENTER is done with FORMAT OFF, then data is expected in the internal representations given above, with no item or statement terminators. If FORMAT is ON, then data is expected in ASCII printable characters as explained below.

# Numeric Items

The number builder is used with numeric items to change ASCII data to numeric data for assignment to a numeric variable. Leading non-numeric characters are ignored. Blanks are ignored where ever they occur. The number is expected to be in the form:

[sign] mantissa [E exponent] item-terminator

where:

sign = + | -
mantissa = [digits] [.] [digits]
exponent = [sign] digits

The mantissa must have at least one numeric digit, whether it be before or after a decimal point. Because REAL numbers have only fifteen digits of precision, any mantissa digits after the 15th digit are considered to be "0". The "E" before the exponent can be either upper or lower case. The item-terminator is any non-numeric character. The item-terminator can either be an EOI on the last character of the number, or any non-numeric character. The entire ENTER statement must also be properly terminated.

```
10  DIM A$[30]
20  A$="hello + 1 2 3 goodbye"&CHR$(13)&CHR$(10)
30  ENTER A$;X,B$
```

This example will assign 123 to X and "oodbye" to B$. The "g" was used to terminate X and thus is not present in B$.

A complex item is entered in rectangular form, real part first, followed by the imaginary part. The rules for each part are the same as the rules for numeric items given above.

# String Items

String items are terminated with either a LF, a CR/LF, an EOI signal, or upon filling the dimensioned length of the string. The LF or CR/LF terminators are not entered into the string.

# Statement Termination

For both string and numeric items, if the last item was not terminated with a LF, CR/LF, or an EOI signal, then additional characters are read (but thrown away) until one of these terminators is received. If no terminator is found within 255 characters, an error is reported.

```
10  DIM A$[30],B$[2]
20  A$="hello + 1 2 3 goodbye"&CHR$(13)&CHR$(10)
30  ENTER A$;B$
```

In this example, because B$ has a dimensioned length of two characters, it is assigned the value "he". The remaining characters are thrown away, and the CR/LF terminate the ENTER.

**Note:** LF or CR/LF are always used for the termination of ENTER, regardless of the setting of EOL in the ASSIGN statement.

# Formatted I/O

OUTPUT, ENTER, PRINT, DISP, and LABEL allow you to control the format of both output and input data. The format is specified as an IMAGE. The IMAGE string can immediately follow the USING keyword as a string (literal or variable) or a line number or label can follow the USING keyword to reference an IMAGE statement that contains the format data. For example:

```
10     OUTPUT CRT USING "ZZZ.DDD";1.2
20     IMAGE ZZZ.DDD
30     OUTPUT CRT USING 20;1.2
```

An IMAGE string consists of a list of one or more image items. The items specified in the image list are acted upon as they are encountered. Each image list item should have a matching output (or enter) item. Processing of the image list stops when no matching output item is found. However, when the last item in the image list is used, the image list is reused starting at the beginning to provide matches for all remaining output items. FORMAT ON is used in connection with USING, even if FORMAT OFF has been specified.

If more decimal places to the left of the decimal point are required to output a numeric item than are specified in the image specifier, an error is generated. If M or S is not specified, then a minus sign will take up one digit place. If the number contains more decimal places to the right of the decimal point than are specified in the image field, the output is rounded to fit.

If the number of characters specified in an image specifier for a string is less than the number of characters in a string, then the remaining characters are ignored. If the number of characters specified is greater than the number of characters in a string then trailing blanks are used to fill out the image field.

# IMAGE Interpretation

Each character in the IMAGE string has a special meaning. For example, the letter **A** means "an alphanumeric character must be printed here", the letter **D** means "a decimal numeric value must be printed here", and the letter **X** means "a space character must be printed here." The order in which data items are specified in the USING statement must match the IMAGE string. For Example:

```
100 PRINT USING 110;"Base Price:",6995
110 IMAGE 11A,X,4D
```

In this example, the print format is specified in the IMAGE statement in line 110. The IMAGE string 11A,X,4D defines two "print fields" or "print zones" separated by one extra space (X). The first print field (11A) specifies that the first data item in the PRINT USING statement must be a character string of not more than eleven characters in length. The second print field (4D) means that the second data item must be a numeric value of not more than four digits to the left of the decimal point. The fractional part of the number, if any, is rounded off in this case. Here is the resulting output:

```
Base Price: 6995
```

When line 100 is executed, the alphanumeric string "Base Price:" and the numeric constant "6995" are sent to the specified output device. The default print format normally used for FORMAT ON output is suppressed.

Notice that each print field specified in the format string has an accompanying data item specified in the PRINT USING statement. The data items are also of the correct type, a character string for the A field and a numeric value for the D field. If the data items were reversed in line 100, a data mismatch would occur, and program execution would halt with an error. If too many data items were specified in line 100 (three data items, for example instead of two), then the IMAGE string would have been reused and the third data item would need to be a string or an error would occur, and program execution would again halt. So, it is important that the type of data items specified in a PRINT USING statement, and the order in which they are specified, match the specifications of the format string.

Complex values and variables are treated as if they were two real variables. Therefore, no special item specifiers are needed for complex numbers.

# Syntax

Some item specifiers can be preceded by a numeric integer constant from 1 to 32,767 that specifies the size of the field or the number of repetitions. The exact syntax of the image specifier is:

```
image-specifier =
    # | % | K | -K | H | -H | B | W | Y | + | - |
    [repeat-factor] A... |
    [repeat-factor] X... |
    [repeat-factor] /... |
    [repeat-factor] L... |
    [repeat-factor] @... |
    numeric-specifier |
    "string-literal"
numeric-specifier =
    [S|M] [left-digits] [.|R] [right-digits] [exp]
left-digits = [repeat-factor] {D|Z|*}...
right-digits = [repeat-factor] D...
exp = E | ESZ | ESZZ | ESZZZ
repeat-factor = integer-constant (1 to 32,767)
```

# OUTPUT, etc.

IMAGE specifiers have the following meanings in DISP, LABEL, OUTPUT, and PRINT statements:

| Specifier | Meaning |
|---|---|
| # | Suppress automatic output of EOL following the last item. |
| % | Is ignored in OUTPUT images. |
| K | Output a number or string in default format, with a period for the radix. |
| -K | Means the same thing as K. |
| H | Output a number or string, default format, comma radix. |
| -H | Means the same thing as H. |

| Specifier | Meaning |
|---|---|
| B | Output a byte, like the CHR$ function. If the value is larger than 32767, 255 is sent. If the value is smaller than -32768, 0 is sent. If the value is in between, it is rounded to an integer and the least significant byte (CINT(value) MOD 256) is sent. |
| W | Output a word in 2's complement 16-bit integer form. If the value is larger than 32767, 32767 is sent. If the value is smaller than -32768, -32768 is sent. If the interface is 16-bit, the word is output in one operation (even if the BYTE attribute was used in the I/O path). If the interface is 8-bit, the byte ordering depends on the LSB/MSB attribute of the I/O path. If the destination is a string, native byte ordering is always used (LSB FIRST on a PC, MSB FIRST on a Sun or HP Workstation). If the WORD attribute was specified in the I/O path, a pad byte will be output before the word when necessary to achieve word alignment. |
| Y | Means the same as W, except that word alignment is not done, and the BYTE attribute is not ignored. |
| + | Change the automatic output of EOL to carriage-return after the last item. |
| - | Change the automatic output of EOL to line-feed after the last item. |
| M | Output a minus sign if negative, a blank if positive. |
| S | Output the sign of the number (+ or -). |
| D | Output one numeric digit character. The leading zeros are replaced by blanks; a minus sign is displayed on negative numbers. |
| Z | Means the same thing as D except leading zeros are displayed. |
| * | Means the same thing as D except leading zeros are replaced with asterisks. |
| .(period) | Output a decimal-point radix indicator. |
| R | Output a comma radix indicator. |
| E | Output an 'E', a sign character, and a two-digit exponent. |
| ESZ | Output an 'E', a sign character, and a one-digit exponent. |
| ESZZ | Output an 'E', a sign character, and a two-digit exponent. |
| ESZZZ | Output an 'E', a sign character, and a three-digit exponent. |
| A | Output an alphanumeric string character. |
| X | Output a blank. |
| / | Output a carriage-return and line-feed. |
| L | Output the current EOL sequence. The default is CR/LF. |
| @ | Output a form-feed character. |
| "string-literal" | Output the characters in the string literal. Remember to double the quote marks when the IMAGE is inside a string literal. |

# ENTER

IMAGE specifiers have the following meanings in an ENTER statement:

| Specifier | Meaning |
|---|---|
| # | Causes the statement to terminate when the last item is terminated. No statement terminator is needed, EOI and LF are item terminators, and early termination is not allowed. |
| % | Is the same as # except EOI causes early statement termination when it terminates an item. |
| K | Allows free-field entry. For **numerics**, entered characters are sent to the number builder, leading non-numeric characters and blanks are ignored, trailing non-numeric characters and characters sent with EOI true are delimiters. For **strings**, entered characters are sent to the string. A CR may be sent to the string if it is not followed by a LF. The string is terminated by CR/LF, LF, character received with EOI true, or the string dimensioned length being filled. |
| -K | Is like K except LF and CR/LF are not terminators. |
| H | Is the same as K except a comma is the radix indicator and a period is a non-numeric character. |
| -H | Means the same as -K for strings, and H for numbers. |
| B | Demands one Byte, like the NUM function. |
| W | Demands a 16-bit Word (2's complement integer). If the interface is 16-bit, the word is entered in one operation (even if the BYTE attribute was used in the I/O path). If the interface is 8-bit, the byte ordering depends on the LSB/MSB of the I/O path. If the source is a string, native byte ordering is always used (LSB FIRST on a PC, MSB FIRST on a Sun or HP Workstation). If the WORD attribute was specified in the I/O path, a pad byte will be entered before the word when necessary to achieve word alignment. |
| Y | Is the same as W, except that word alignment is not done, and the BYTE attribute is not ignored. |
| + | Indicates an END (EOI) is needed with the last character of the last item to terminate the ENTER statement. LFs are no longer statement terminators, but are still item terminators. |
| - | Indicates a LF is needed to terminate the ENTER statement. EOI is ignored; other END indicators cause an error. |
| S | Same meaning as D. |
| M | Same meaning as D. |
| D | Demands one character for each D, or repeat count. Non-numerics are consumed while fulfilling the count but also delimit the number. Blanks embedded in the number are ignored. |
| Z | Same meaning as D. |
| * | Same meaning as D. |
| .(period) | Same meaning as D. |
| R | Has the same meaning as D, plus the number builder is instructed to use a comma as the radix indicator and a period as a non-numeric character. |
| E | Is treated the same as 4D. |
| ESZ | Same as 3D. |
| ESZZ | Same as 4D. |
| ESZZZ | Same as 5D. |
| A | Demands one alphanumeric string character. |
| X | Enters a character and discards it. |
| / | Skips all characters to the next LF and EOI is ignored. |
| L | Ignored in ENTER. |
| @ | Ignored in ENTER. |
| "string-literal" | One character is skipped for each character in the string literal. Remember to double the quote marks when the IMAGE is inside a string literal. |

# Creating Format Strings

The following examples illustrate how to combine characters into IMAGE strings. These examples use PRINT, but DISP, LABEL, or OUTPUT could also have been used to direct the output to various destinations.

The A character defines a field for alphanumeric character strings. The operator is specified in the form nA where n is an integer from 1 through 32767. If n is not specified, then 1 is used.

# PRINT Examples

This example shows creation of an alphanumeric field:

```
110 A$ = "Greg Doe"
120 IMAGE 15A,2X,24A
130 PRINT USING 120;"Student Name:",A$
```

In line 120 above, the IMAGE item defines an alphanumeric field with 15 alphanumeric positions, 2 extra spaces, and then 24 more character positions. In line 130, the string constant "Student Name:" is printed in the first field, and "Greg Doe" in the last field.

The following shows how the output is printed relative to the field description. The first line shows the fields specified by the IMAGE string. The second line shows the PRINT output.

```
AAAAAAAAAAAAAAAXXAAAAAAAAAAAAAAAAAAAAAAA
Student Name:    Greg Doe
```

Notice that the character strings are left justified in the fields. This means that the first character in the string is printed in the left-most position in the field. The character string "Student Name:" only fills 13 character positions, the remaining positions are filled with space characters. Greg Doe fills 8 positions and the remaining 16 positions in that field are filled with space characters. Character strings larger than the specified field will be truncated.

The print position is set to a new line after each line is printed unless the IMAGE format ends with "#" to suppress the EOL. You would want to do this if you needed to continue output on the same line with another output statement.

Here is another PRINT USING example:

```
10    IMAGE "ERROR: ",2D,", ",K
20    Esc$=CHR$(27)
30    PRINT USING "A, ""[H"", A, ""[J"", #";Esc$,Esc$
40    PRINT USING 10;ERRN,"in line "&VAL$(ERRLN)
50    PRINT USING 10;19,"Out of range!"
60    PRINT USING 10;Error_code,"Oops!"
70    PRINT USING " ""The value of PI is"",X,1D.10D ";PI
```

# ENTER Example

The following examples demonstrate how the ENTER USING rules are used to input floating point and integer numbers.

```
10    DIM A$[22]
20    A$="Dollars: $34.56 Total"
30    ENTER A$ USING "K";X1
40    ENTER A$ USING "14D";X2
```

In line 30, the number builder ignores non-numeric characters preceding the number and stops at the first character that is not part of the number so that X1 equals 34.56. In line 40 the number builder stops after 14 characters ("Dollars: $34.5") are input so that X2 equals 34.5.

```
10 A$ = "2XX+3Y"
20 ENTER A$ USING "D,5D";X,Y
```

In this example, the variable X is assigned the value 2, and Y is assigned 3.

# END

An optional END may be used after the last data item. If USING is specified, then the effect of the END is quite different than if USING is not specified. An END after the last item in an OUTPUT USING statement will have the following effect:

1) If the last image specifier does not have an output item (X, /, @, L, and "literals"), then EOL is suppressed, including alternate EOLs specified by + or -, and no EOI is sent. Exception 1: If the IMAGE specifier is L and END is specified in the ASSIGN then EOI is sent with the EOL whenever the EOL is sent.

2) If the last image specifier does have an output item (K, H, D, Z, *, A, B, W, and Y) then EOL is unaffected and EOI is sent with the last byte sent. Exception 2.1: If END is also used in the ASSIGN statement then two EOIs are sent, one with the last output item and one with the EOL. This is the exception that HTBasic does not emulate; the general rule 2 is followed. Exception 2.2: If the last output item is a string with a length of zero, and the image specifier is K, and # has suppressed the EOL, then no EOI is sent. As an example, this statement will output one character, "A", with no EOI:

```
OUTPUT 717 USING "K,K,#";"A","" END
```

3) If the last image specifier changes the default EOL (#, +, and -), then use the next to last specifier to determine whether to use rule 1 or 2, above.

# TRANSFER

The TRANSFER statement sets up unformatted data transfers between memory and a device. The data transfer normally occurs in the "background." That is, the BASIC program continues to run in the "foreground" simultaneously with the background transfer. Optionally, the TRANSFER statement can wait until the transfer is complete before continuing. The syntax is

TRANSFER @*source-io-path* TO @*dest-io-path*
[; [*term-list*] [,] [EOR(*term-list*)] [,] [*type*] ]

Use the ASSIGN statement to initialize the source and destination I/O paths. The optional commas are only needed when items occur on both sides of the comma.

## Examples

```
TRANSFER @Device TO @Buffer
TRANSFER @Buff TO @Logger;CONT
TRANSFER @Rs232 TO @Buff;DELIM CHR$(13)
TRANSFER @Path TO @Buff;RECORDS 16,EOR(END)
```

# Support

TRANSFER is currently supported for files, RS-232 and GPIB.

# Buffers

The transfer operation must be between a buffer and a device. A buffer must be declared as the source for an outbound transfer, or as the destination of an inbound transfer. One buffer can simultaneously be used for an outbound transfer and an inbound transfer. A transfer directly between two devices is not supported.

Buffers may be unnamed or named. An unnamed buffer is created, assigned an I/O path, and given its size by the ASSIGN statement. A named buffer is a previously declared REAL, INTEGER, LONG or COMPLEX array, or a string scalar (declared in a COM, STATIC, DIM, INTEGER, LONG, REAL, or COMPLEX statement) that has been ASSIGNed to an I/O path. Unnamed buffers are usually preferred because the size can be as large as available memory and no side-affects are possible by accessing the buffer through its variable name.

Buffers are circular; each buffer has a fill and empty pointer as well as a count. The fill pointer is used by an inbound transfer to identify the next location for data to be stored (inserted). The empty pointer is used by an outbound transfer and points to the next location for data to be output (removed). A value of one for either pointer means the first byte of the buffer. When the fill and empty pointers have the same value, the count can be examined to determine whether the buffer is empty or full.

The I/O path assigned to the buffer is called the buffer-I/O path. The I/O path assigned to the device is called the non-buffer-I/O path. The buffer should be accessed only with the buffer-I/O path. The count, fill, and empty pointers can be examined using STATUS on the buffer-I/O path. OUTPUT @buf or an inbound transfer are used to place data into a buffer. ENTER @buf or an outbound transfer are used to read and remove data from a buffer. The variable name of a named buffer should generally not be used to access the data in the buffer since the data in the buffer is unformatted and may even have the wrong byte order.

# Transfer Type

The *type* of the transfer can be specified as CONT, WAIT, or left unspecified.

If WAIT is specified, the transfer executes in foreground mode. Program execution does not proceed beyond the TRANSFER statement until the transfer terminates. If an error occurs, it is reported with the line number of the TRANSFER statement. If WAIT is not specified, execution continues past the TRANSFER statement and the transfer takes place in the background. Then if an error occurs, the error is not reported until the non-buffer-I/O path is referenced. The error line reported is not that of the TRANSFER, but of the statement where the non-buffer-I/O path was referenced.

If CONT is specified, TRANSFER executes continuously. For an inbound transfer, execution pauses when the buffer is full and continues when space is available in the buffer. For an outbound transfer, execution pauses when the buffer is empty and continues when the buffer has data available. If CONT is not specified, the end-of-transfer occurs when an outbound transfer empties the buffer or an in-bound transfer fills the buffer. Or if a termination method has been specified as explained below, the transfer terminates when the condition occurs.

Both WAIT and CONT can be specified together if a transfer is already active for the buffer in the opposite direction. The transfer will be continuous, but will run in the foreground.

If neither WAIT nor CONT is specified, the transfer occurs in the background. The end-of-transfer occurs when an outbound transfer empties the buffer or an in-bound transfer fills the buffer. Or if a termination method has been specified as explained below, the transfer terminates when the condition occurs.

# Transfer Method

A couple of methods are available for accomplishing the transfer: DMA (direct memory access) and interrupts. DMA is the fastest method and will be used automatically if possible. A DMA channel must be available, the interface must have the necessary hardware, and DELIM can not have been specified. If DMA can not be used, interrupts are used.

# Transfer Termination

A *term-list* can be used to specify a list of conditions that cause the transfer to end. One or more of the following conditions can be used:

- COUNT *Bytes*
- DELIM *Character*
- END
- RECORDS *Number*

If COUNT is specified, the transfer terminates after the specified number of bytes has been transferred.

If DELIM is specified for an inbound transfer, then the transfer is terminated after the specified character is detected. DELIM is not allowed with outbound transfers. If the delimiter string is zero length, delimiter checking is disabled. DELIM prevents DMA from being used; interrupts will be used instead.

If END is specified for an inbound transfer, the transfer terminates when the device dependent signal is received. On the IEEE-488 interface, END is the EOI signal. When an inbound transfer is terminated in this way, bit 3 of register 10 is set. For an outbound transfer, END does not specify a termination condition, but rather specifies that the device dependent signal (EOI) is sent with the last byte sent.

If RECORDS is specified, the transfer terminates when the specified number of records has been transferred. An EOR(*term-list*) must be specified, defining what will be considered a record for the purpose of this particular transfer. For inbound transfers the legal end-of-record termination conditions are COUNT, DELIM, and END, or some combination of these three. For outbound transfers only COUNT can be used to define a record, although END can be used to specify that the device dependent signal (EOI) is sent with the last byte of each record.

# ON EOR and ON EOT

The ON EOR and ON EOT statements can be used to generate an event when an end-of-record or end-of-transfer occurs. The WAIT FOR EOR and WAIT FOR EOT statements can be used to stop further statement execution until an end-of-record or end-of-transfer occurs.

# Termination

To terminate a CONT, continuous mode, outbound transfer without leaving data in the buffer, use the following sequence of statements:

```
CONTROL @Buff,8;0
WAIT FOR EOT @Non_buff
```

# Hanging and Premature Termination

HTBasic will not enter a stopped state until all transfers are completed. Likewise, HTBasic will not exit a program context until transfers started in that context are finished. The following statements also cause the computer to "hang" until all transfers complete: GET, LOAD, RETURN, STOP, SUBEND, SUBEXIT, or modifying a program line.

The ABORTIO statement can be used to prematurely terminate a transfer and free the computer. The RESET key will also terminate any active transfers, but ABORTIO is preferred.

# Outbound TRANSFER

An outbound transfer has the form:

TRANSFER @Buff TO @Non_buff

If another outbound TRANSFER statement is executed while an outbound TRANSFER is occurring, HTBasic waits for completion of the first before starting the second. Any EOT/EOR events caused by the first transfer will then be logged and may be serviced before the next program line.

# Inbound TRANSFER

An inbound transfer has the form:

TRANSFER @Non_buff TO @Buff

If another inbound TRANSFER statement is executed while an inbound TRANSFER is occurring, HTBasic waits for completion of the first before starting the second. Any EOT/EOR events caused by the first transfer will then be logged and may be serviced before the next program line.

# STATUS, CONTROL, READIO and WRITEIO

I/O paths and many device interfaces have registers that control aspects of their operation and report their status. For example, screen colors can be controlled by writing to the registers of the CRT interface. The baud rate can be changed by writing to the registers of the serial interface. The status of an I/O path variable can be read from the I/O path registers. These and many other attributes can be controlled and read through interface registers.

In general, there are three different kinds of registers: I/O path registers, interface registers, and hardware registers. The following program shows access to all three types:

```
10    ASSIGN @Path TO 9
20    PRINT "I/O Path  Register 0:",STATUS(@Path,0)
30    PRINT "Interface Register 0:",STATUS(9,0)
40    PRINT "Hardware  Register 0:",READIO(9,0)
```

The CONTROL and STATUS statements are used to access two of the three kinds of registers: I/O path registers and interface registers. If the statement specifies an I/O path, I/O path registers are accessed. If the statement specifies an interface select code, interface registers are accessed.

The range of legal register numbers and their definitions differ depending on the I/O path assignment or the type of interface. I/O path registers are given below. The registers for each interface are given in the chapters that follow this one, or the documentation that came with the interface.

# CONTROL Statement

The CONTROL statement sends control information to an interface or an I/O path control register. Information is sent by specifying a starting register number, and a value to be sent to that register. If it is not specified, the starting register number is zero. If you specify more than one data value, the register number is incremented by one after sending each value. For example:

```
CONTROL 2;Column,Line
```

sends the value of the variable Column to register zero on interface number two and the value of the variable Line to register one.

# STATUS Statement and Function

The STATUS statement and STATUS() function return control information from an interface or an I/O Path status register. Using the STATUS statement, the values of several status registers are copied into a list of numeric variables, starting at the specified register number and continuing until the variable list is exhausted. If it is not specified, the starting register number is zero. For example:

```
STATUS 2;Column,Line
```

gets the value of status register zero on interface number two and stores it into the variable Column and then it gets the value of status register one and stores it into the variable line.

The STATUS() function complements the STATUS statement. It allows immediate access to a single register without need for a temporary variable or separate STATUS statement. However, the STATUS() function can only return the value of one register at a time, while the STATUS statement can return the values of multiple registers in a single statement.

The CONTROL and STATUS registers for I/O paths assigned to devices, files and buffers are described in the following paragraphs.

# Device I/O Path Registers

No CONTROL registers exist for I/O paths assigned to devices. The STATUS registers for I/O paths assigned to devices are:

0 - ASSIGN status: 0 - not assigned, 1 - assigned to a device
1 - The interface select code
2 - Number of devices
3 - Device selector of device 1
4 - Device selector of device 2 (if two or more devices are present)
n+2 - Device selector of device n (if n or more devices are present)

# ASCII File I/O Path Registers

No CONTROL registers exist for I/O paths assigned to ASCII files. The STATUS registers for I/O paths assigned to ASCII files are:

0 - ASSIGN status: 0 - not assigned, 2 - assigned to a file
1 - File type: 3 - ASCII file
2 - Always 4
3 - Number of records. Windows files are extendable and so the number of records can be more or less than the number given in the CREATE statement for the file
4 - Record size: 256 bytes
5 - Current record
6 - Current byte within record

# BDAT and Ordinary File I/O Path Registers

For I/O paths assigned to BDAT and ordinary files, all registers can be read with the STATUS command. Only registers greater than 4 can be set with the CONTROL command. Ordinary files are listed by CAT with a blank file type or the name of the operating system.

0 - ASSIGN status: 0 - not assigned, 2 - assigned to a file. A DOS device looks like a file to HTBasic when assigned using its DOS name
1 - File type: 2 - BDAT file, 4 - ordinary file
2 - Always 4
3 - Current number of records. Windows file lengths are extendible, so the number of records can be more or less than the number given in the CREATE. If the last record is only partially filled, it is still counted
4 - Record size. For ordinary files, the record length is always 1
5 - Current record
6 - Current byte within record
7 - EOF record
8 - Byte within EOF record

EOF record and byte define the position of the first byte after the last byte in the file.

# BUFFER I/O Path Registers

All registers can be read with the STATUS command. The following registers can be set with the CONTROL command: 3, 4, 5, 8, and 9.

0 - ASSIGN status: 0 - not assigned, 3 - assigned to a buffer
1 - Buffer named flag: 1 - named, 2 - unnamed
2 - Buffer size in bytes
3 - Current fill pointer
4 - Current number of bytes in buffer
5 - Current empty pointer
6 - Interface select code for in-coming TRANSFER
7 - Interface select code for out-going TRANSFER
8 - Continuous in-coming TRANSFER flag: zero - not continuous, non-zero - continuous
9 - Continuous out-going TRANSFER flag: zero - not continuous, non-zero - continuous
10 - In-coming TRANSFER status

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Always 0 |
| 6 | 64 | Active |
| 5 | 32 | Aborted |
| 4 | 16 | Error |
| 3 | 8 | Device caused termination |
| 2 | 4 | Byte count caused termination |
| 1 | 2 | Record count caused termination |
| 0 | 1 | Match char caused termination |

11 - Out-going TRANSFER termination status. Same meaning as above
12 - Count of bytes transferred by last in-coming TRANSFER
13 - Count of bytes transferred by last out-going TRANSFER

# Pipe I/O Path Registers

No CONTROL registers exist for I/O paths assigned to pipes.

# Interface Hardware Registers

Accessing hardware registers can cause your system to crash, data to be lost, or damage to your hardware. TransEra cannot be held responsible for any consequences.

The READIO and WRITEIO statements allow an HTBasic program to directly manipulate the interface hardware and physical memory locations. The hardware register numbers and their definitions are dependent on the actual hardware. The level of access to the computer hardware is also dependent on operating system protection methods. Do not mix READIO and WRITEIO operations with STATUS and CONTROL operations. In general, you are better off using STATUS and CONTROL operations and avoiding READIO and WRITEIO.

# READIO Function

The READIO function returns the contents of a hardware register of an interface. The value of the first argument specifies an interface select code and a valid interface register number is specified as the second argument. That hardware interface register is read and its contents are returned. For example:

```
PRINT "Register=";READIO(7,1)
```

prints the contents of interface register number one on interface number seven.

Extensions to READIO and WRITEIO allow PEEK and POKE operations. Other extensions allow the address of a numeric variable to be read, and assembly language subroutines to be called. All these extensions are explained later in this chapter.

# WRITEIO Statement

The WRITEIO statement writes a data value to an interface hardware register. If a valid interface select code and interface register number are specified, then the data value is written to the hardware interface register. For Example:

```
WRITEIO 7,3;Regdata
```

writes the value of the variable Regdata to register number three on interface number seven. Again, extensions to READIO and WRITEIO are explained below.

# PEEK/POKE Memory

The number 9826 is used to specify a peek or poke operation in a READIO or WRITEIO statement. If positive, a byte operation is done; if negative, a word operation is done. These combinations are shown in the table below. L specifies the address of the byte or word. If you specify a word operation and L is odd, the even address L-1 is used. V specifies the value to store into memory. L specifies an address within the HTBasic process.

READIO(9826,L)  PEEK byte
READIO(-9826,L)  PEEK word
WRITEIO 9826,L;V  POKE byte
WRITEIO -9826,L;V  POKE word

WARNING**:** Use PEEK and POKE only on addresses returned by READIO(9827,I)! Using other locations can cause your system to crash, data to be lost, or damage to your computer hardware. Use of this function for any other address is unsupported, and TransEra cannot be held responsible for any consequences.

# IN/OUT Operations

At the lowest level, the CPU in your computer must be able to input or output data. Some CPU's, like the Motorola 680x0 processors in an HP Workstation, use memory space for all CPU I/O, even for devices. Other CPU's, like the Intel 80x86 processors in a PC, have an I/O space in addition to memory space. PEEK/POKE access memory space; IN/OUT access I/O space. The following four statements are available in HTBasic to access I/O space.

INP(L) IN byte
INPW(L) IN word
OUT L,V OUT byte
OUTW L,V OUT word

L specifies the address of the byte or word. V specifies the value to read or write. IN/OUT operations are most typically used to access a data acquisition board plugged into your computer for which there is no HTBasic driver. Most boards come with example programs showing how to access them with IN/OUT instructions.

WARNING**:** Because incorrect use of IN and OUT can cause your system to crash, data to be lost, or damage to your computer hardware, use this function at your own risk. TransEra cannot be held responsible for any consequences.

# Locating a Numeric Variable

READIO(9827, variable-name ) is used to locate a numeric variable or an element of an array. This operation is useful in connection with assembly language subroutines because it allows you to identify the starting address of the subroutine so that the subroutine may be called with the WRITEIO statement.

```
10 INTEGER A(0:5)
20 Address = READIO(9827,A(0))
30 PRINT "The address of A(*) is ";Address
```

# Interrupts

Interrupts allow the computer to perform other tasks while you wait for some condition to occur. This eliminates the need to continually monitor for some event.

# ON INTR Statement

The ON INTR statement defines an event branch to be taken when an interface card generates an interrupt. You specify the interface select code, an optional priority and the branch type. The branch type may be either a GOTO, GOSUB, CALL, or RECOVER. For example:

```
ON INTR 7,4 GOSUB Repair
```

When an interrupt occurs a DISABLE INTR for the interface is automatically executed. Consequently, an ENABLE INTR statement must be used to explicitly re-enable interrupts.

The default priority is one. The highest priority that can be specified is fifteen. ON END, ON ERROR, and ON TIMEOUT have a higher priority than ON INTR. When an INTR initiated branch is taken with a GOTO the system priority is not changed. When an ON INTR branch specifies a CALL or GOSUB the system priority is changed to the specified priority.

RECOVER causes the program to SUBEXIT from contexts as needed to return to the defining context and resume execution at the specified program line. ON INTR statements that specify CALL or RECOVER will be serviced even if the program context has been changed to another subprogram. ON INTR statements that specify GOTO or GOSUB will be logged and then serviced when control returns to the defining program context.

ON INTR is canceled by OFF INTR, disabled by DISABLE or DISABLE INTR.

# OFF INTR Statement

The OFF INTR statement cancels event branches defined by ON INTR. Any INTR events that have been logged but not yet serviced are canceled. An OFF INTR statement without the optional interface select code disables event-initiated branches on all devices. If the interface select code is specified only that interface interrupt will be disabled. For example:

```
OFF INTR 7
```

cancels event branches for the IEEE-488 interface.

# Enabling and Disabling Interrupts

The DISABLE statement disables all defined event branches except END, ERROR, and TIMEOUT. While disabled, the first event of each type that occurs, is logged. When event branching is re-enabled with the ENABLE statement, all logged events are serviced in the order of their event priorities.

The DISABLE INTR Statement disables interrupts from just the specified interface. For example:

```
DISABLE INTR 7
```

disables interrupts from the IEEE-488 interface.

The ENABLE INTR statement enables interrupts from a specified interface. An optional bit mask is stored in the interface interrupt-enable register. The default bit mask is the previous bit mask for that interface, or if there is no previous bit mask then a bit mask of all zeros is used. The meaning of the bit mask depends on the interface; consult the interface documentation. For example:

```
ENABLE INTR 9;1
```

enables interrupts on the RS-232 interface and stores one into the interface interrupt-enable register. For the serial interface, one happens to mean "interrupt when a character is received."

# Interrupt Routines

When an interrupt occurs, the event handler would typically perform the following steps: 1) find out what action needs to be taken, 2) perform the needed action, 3) perform whatever interface specific action is necessary to acknowledge the interrupt, and 4) re-enable interrupts with the ENABLE INTR command. The following example shows the typical sequence of statements used for interrupt set up and handling:

```
10   ON KBD CALL To_modem
20   ON INTR 9 CALL From_modem     !Tell BASIC to interrupt
30   ENABLE INTR 9;1               !Tell interface to interrupt
40   LOOP
50     DISP TIME$(TIMEDATE)        !Now free to do something
60   END LOOP                      !while you wait
70   END
80   SUB From_modem
90     WHILE BINAND(STATUS(9,10),1)
100      PRINT CHR$(STATUS(9,6));  !Interface dependent ack.
110    END WHILE
120    ENABLE INTR 9;1             !re-enable the interrupt
130  SUBEND
140  SUB To_modem
150    OUTPUT 9;KBD$;
160  SUBEND
```

# Specialized I/O Statements

In addition to the general I/O statements explained above, several statements are provided for specialized I/O. PRINT, DISP, INPUT, and LINPUT allow specialized I/O with the CRT or keyboard. They are explained in Chapter 6, "CRT and Keyboard." The PRINT statement can also be redirected to a printer. READ and DATA allow data to be stored within the BASIC program itself. BEEP allows tone generation on computers with the necessary hardware.

# READ/DATA Statements

The READ statement is quite similar to the INPUT statement. However, the values are read from DATA statements instead of the keyboard. DATA statements contain string and/or numeric constants separated by commas. This provides a convenient method of embedding known data that your program requires, right in the program itself. The first READ statement in a context reads the first DATA statement in that context. Each READ statement thereafter maintains a DATA pointer that moves to the next item after each is read from the DATA statement. The DATA pointer can be reset to the beginning of any DATA statement in the context with the RESTORE statement.

READ statements can be useful for initializing the values of several program variables more compactly than with individual [LET] assignment statements. It is also handy for table data that you can READ into an array.

The following example shows the use of the DATA, READ and RESTORE statements.

```
100  DATA 1,2,"STRING CONSTANT",10
110  READ A,B,A$,C       ! read data from line 100
120  DATA 13,24,36,42,59
130  DIM D(4),E(4)
140  READ D(*)           ! read array data from line 120
150  MAT D=D*(B)
160  RESTORE 120
170  READ E(*)           ! read data from line 120 again
180  END
```

Note that line 160 specifies line 120 as the data statement to be restored since there is a previous data statement in line 100. If line 120 were the first data statement in the program, line 160 could be simply RESTORE with no line number.

# BEEP Statement

BEEP is a statement used to play music or tones by producing notes of a certain frequency and duration. HP BASIC rounds the frequency value to a multiple of 81.38 Hz and supports a range of 81 Hz to 5.208 KHz.

```
10   REM — Print table of musical notes.
20   REM — Also play C-major scale.
30   A4=440     ! Frequency of the reference note
40   R=2^(1/12) ! One octave doubles the frequency, 12 half-steps in octave
50   C4=A4/(R^9)! Scale goes from C to B
60   DATA C,C#,D,D#,E,F,F#,G,G#,A,A#,B
70   DIM Name$(11)[2]
80   READ Name$(*)
90   N=C4/8     ! start at C1
100  CLS        ! clear the screen
110  FOR Octave=1 TO 7
120      Col=1+(Octave-1)*11
130      PRINT TABXY(Col,1);"Note  Freq|";
140      PRINT TABXY(Col,2);"————+";
150      FOR Note=0 TO 11
160          OUTPUT A$ USING "3A,X,4D.D,#";
             Name$(Note)&VAL$(Octave),N
170          PRINT TABXY(Col,Note+3);A$;"|"
180          IF Octave>1 AND LEN(Name$(Note))=1 THEN BEEP N,.25
190          N=N*R
200      NEXT Note
210      PRINT TABXY(Col,15);"————+";
220  NEXT Octave
230  BEEP N,.5 ! complete the last scale
240  END
```

This example demonstrates the BEEP statement, while also producing a useful table. The program plays C Major scales for octaves two through seven. At the same time, it prints a table of musical notes and their associated frequencies. The frequencies printed are for the Equal Tempered Chromatic Scale adopted by the American Standards Association in 1936. The "A" note in the fourth octave (A4) is the reference note and has a value of 440 Hertz. An increase of one octave, to A5 doubles the frequency, while moving down one octave halves the frequency. Twelve half-steps compose an octave and are equally spaced geometrically (rather than arithmetically). That is, the ratio of frequencies between any two adjacent notes is a constant.

Another standard that exists uses A4=435. This was an International standard adopted in 1891. The program is easily modified to show this scale. Change the value 440 in line 30 to 435. Other modifications could change the base of the scale from "C" to something else, or change the number of notes in a scale.

On the PC, the period (not the frequency) is rounded to a multiple of 0.838 micro-seconds. The range of frequencies is 40.7 Hz to 32.767 Khz.

# Summary

This chapter discussed the general I/O (input/output) facilities of HTBasic for screen, keyboard, devices, strings, buffers and files. Formatted I/O with IMAGE and USING statements were explained. The special I/O statements BEEP and DATA statements were presented. Additional details and examples of I/O directed to files and devices are presented in the following chapters.

# CRT, Keyboard and Printer

This chapter discusses I/O (input/output) facilities for the screen and keyboard. The general I/O statements presented in Chapter 5 are given, as well as special statements for display, keyboard, and printer support. Controlling various attributes of the screen and keyboard, including screen colors, is explained. Finally, tables of the CRT and KBD registers are given.

# Display (CRT)

The CRT is the display on your computer. CRT stands for *Cathode Ray Tube* and is often used as a synonym for *display*, even when the display is not a tube. When HTBasic starts, it automatically uses the Windows display driver.

The interface select code (ISC) of the CRT is permanently set to 1. This ISC is used so often, a special function, CRT, can be used to return the value 1, while providing a mnemonic for the ISC.

```
OUTPUT CRT;"Hello World"
ASSIGN @Output TO CRT
```

# Display Organization

As explained in Chapter 1, "Language Elements," the computer display is organized in a very particular way by HTBasic. Different statements affect different parts of the display. To display information in the output area, use the OUTPUT or PRINT statements. To display information on the Display Line, use the DISP statement. The Input Lines show the information the user types in response to queries by your programs. Information can be pre-loaded into the Input Lines by using OUTPUT KBD. The message line shows various system messages as well as live keyboard calculator results. The softkey menus display the current softkey macros unless an ON KEY is active, in which case the ON KEY LABEL overwrites the softkey macro for that key. Each of these areas of the screen are discussed in more detail through out this chapter.

# OUTPUT CRT

The OUTPUT CRT statement sends information to the output area of the screen. OUTPUT CRT uses rules consistent with OUTPUT to other devices and is useful for verifying I/O program correctness. However, the PRINT statement is often better suited for screen output.

# PRINT

The PRINT statement is provided to allow easy printing of information on the screen or to a printer. By default, PRINT output goes to the screen, but the PRINTER IS statement can be used to redirect output from the PRINT statement to a printer, device, or file. The PRINTER IS statement can also set the width at which output should wrap, and the characters that should be sent at the end of a line. These features allow easy use of many different printers.

# DISP Statement

The DISP statement is very similar to the PRINT statement. However, DISP output goes to a part of the screen known as the display line. When the semicolon is used to end display output, the display line will scroll to the left if output goes off the right end of the display line. Most other aspects of the DISP statement match those of the PRINT statement.

# PRINT and DISP Statements

With both PRINT and DISP, if you separate arguments in a print list with commas, they will be printed in columns. The columns are ten characters wide. If you want "compressed" format, substitute a semicolon in place of the comma. In compressed format, numerics are printed with one additional trailing space and strings are printed with no additional spaces. You may also end a print list with a semicolon to suppress the usual CR/LF that forces a new line. Then you can continue output on the same line with another PRINT statement. If the default PRINT/DISP formats are not acceptable, USING can be specified to format the data as desired.

Numbers are printed with twelve significant digits. If the number is outside the range 1E-4 and 1E+6 then the number is printed in scientific notation. If the number is positive, it is always preceded by one space, even in the compressed format. If it is negative, the negative sign is printed in place of the leading space.

Complex numbers are printed in rectangular form, first the real part, then an extra space, and finally the imaginary part. The real and imaginary parts are printed using the print rules given in the previous paragraph.

You can position text on the screen with TAB(column) and TABXY(column,row). The left-most column is column one, and the top-most row is row one. TAB can also be used on a printer; TABXY can only be used on the screen. The following example illustrates use of TABXY.

```
10  ! Display time in upper-right corner for 30 seconds
20    St=TIMEDATE ! start time
30    REPEAT
40      PRINT TABXY(70,1);TIME$(TIMEDATE)
50    UNTIL TIMEDATE>St+30
60    END
```

A full array can be printed by using the array name with a full array specifier. The elements are printed in row major order, and in fields determined by the punctuation following the array.

# Display Functions

It is possible to disable the effect of the attribute characters just described, displaying them instead of executing them. This is useful when debugging OUTPUT. The DISPLAY FUNCTIONS ON statement causes all control characters to be displayed but not executed. The only exception is carriage return, CHR$(13), which is first displayed and then the print cursor is moved to column one of the next line. DISPLAY FUNCTIONS OFF returns execution of attribute characters to normal.

This function is the equivalent of  pressing the DISPLAY FCTNS key, or executing the CONTROL CRT,4 statement.

# CRT Related Statements

There are several statements that affect the CRT. Consult the on-line *Reference Manual* or use the HELP statement to obtain more information on each of these statements.

| Statement | What It Does |
|---|---|
| ALPHA ON/OFF | Controls ALPHA screen visibility |
| ALPHA HEIGHT | Sets the number of lines in ALPHA screen |
| ALPHA PEN | Sets the ALPHA display color |
| CLEAR LINE | Clears the keyboard input line |
| CLEAR SCREEN | Clears the ALPHA display |
| CLS | Abbreviation for CLEAR SCREEN |
| DUMP ALPHA | Prints the contents of the ALPHA display |
| GRAPHICS ON/OFF | Controls graphics screen visibility |
| KBD LINE PEN | Sets the color of the input line |
| MERGE ALPHA | Enables all planes for Alpha and Graphics |
| PRINT PEN | Sets the output area and display line color |
| RUNLIGHT ON/OFF | Controls run indicator visibility |
| SEPARATE ALPHA | Simulates independent alpha and graphics |

These statements affect the softkey menus:

| Statement | What It Does |
|---|---|
| EDIT KEY | Edits a softkey macro |
| KBD CMODE ON/OFF | Changes softkey labels to match HP keyboards |
| KEY LABELS ON/OFF | Controls the display of the softkey labels |
| KEY LABELS PEN | Sets the color for the softkey labels |
| LOAD KEY | Loads softkey macros from a file |
| ON/OFF KEY | Defines an event branch for a softkey |
| SCRATCH KEY | Deletes softkey macros |
| SET KEY | Defines one or more softkey macros |
| SYSTEM KEY | Displays the System Softkeys Menu |
| USER KEYS | Displays the specified User Softkey Menu |

# CONTROL/STATUS CRT

The CONTROL CRT statement can be used to control various CRT attributes, while the STATUS CRT can be used to read the status of those attributes. A complete list of the CONTROL and STATUS registers for the CRT is given at the end of this chapter. The following is an example:

```
10    ON CYCLE 1,15 CALL Time
20    LOOP
30      PRINT TABXY(RND*68+1,RND*(STATUS(CRT,13)-)+1);CHR$(32+RND*96);
40    END LOOP
50    END
60    SUB Time
70      STATUS CRT,0;Col,Row
80      CONTROL CRT,0;70,1
90      PRINT TIME$(TIMEDATE);
100     CONTROL CRT,0;Col,Row
110   SUBEND
```

In this example, each time the SUB "Time" is called, the print position is remembered, the time is written in the upper-right-hand corner of the screen, and then the print position is restored. Line 10 causes the sub to be called once a second. Lines 20 to 40 are just to keep the program doing something, to demonstrate the updating of the time. These lines could be deleted and replaced with whatever program you wish. Notice in line 30 the function form of STATUS and in line 70, the statement form.

# ENTER CRT

ENTER CRT enters information from the CRT screen just as if it had been sent to the computer from some external device. The present print position is the source of the data and is updated as data is entered. Trailing spaces on a line are ignored. The last character on the line is a line-feed (LF) with an EOI signal.

```
10 PRINT TABXY(1,1);"HELLO!";TABXY(1,1);
20 ENTER CRT;A$
```

The string variable A$ is assigned "HELLO!"

# Keyboard (KBD)

The interface select code (ISC) of the keyboard is permanently set to 2. This ISC is used so often, a special function, KBD, can be used to return the value 2, while providing a mnemonic for the ISC.

```
ENTER KBD;A$
ASSIGN @Input TO KBD
```

# ENTER KBD

ENTER KBD reads information from the keyboard. The INPUT and LINPUT statements also read information from the keyboard but are designed especially for the keyboard and provide some features not supported by ENTER. The number builder rules are not the same for the two. ENTER KBD uses rules consistent with ENTER from other sources and is useful for verifying I/O program correctness. Toward this end, an EOI signal can be generated from the keyboard by entering a control-E before the character to be sent with the EOI signal. This pseudo-EOI must be enabled with CONTROL KBD,12;1. No keystrokes are entered into ENTER items until either a CONTINUE key or ENTER key is pressed. If CONTINUE is pressed, the present keyboard buffer is transmitted through the I/O path to the ENTER statement with no appended characters. If ENTER is pressed, the buffer is transmitted with CR/LF appended to the buffer.

# INPUT Statement

The INPUT statement allows the user to assign a value to a variable by typing in the value on the keyboard. A prompt is displayed on the display line. The INPUT statement can specify the prompt, or a question mark (?) will be used by default. To suppress the prompt, specify a prompt string of "".

The input statement can input values for simple variables, full arrays, array elements, or sub-strings. A full array name must be followed by the full array specifier, "(*)". Values for the array must be entered in row major order.

Leading and trailing spaces are ignored. Data values may be entered individually or multiple values may be entered at once. If multiple values are entered, separate each value with a comma. If too many values are entered, the extra values are ignored. Both quoted and unquoted strings are allowed. Commas are not allowed in unquoted strings, but may appear in quoted strings. To embed one quotation mark in a quoted string, type in two quotation marks at the place you wish to have one appear.

Two consecutive commas cause the corresponding variable to retain its old value. Terminating an input line with a comma or pressing CONTINUE or ENTER without entering any data retains the original values for all remaining variables in the list.

Let's look at a simple program to INPUT a value and assign it to a variable and then examine how to improve it.

```
10 DISP "Enter a file name";
20 INPUT F$
30 END
```

Now let's see how to do this in just one statement:

```
10 INPUT "Enter a file name: ",F$
20 END
```

To provide a default answer we might do the following:

```
10 F$ = "STUDENT.DAT"
20 INPUT "Enter a file name <STUDENT.DAT>: ",F$
30 END
```

If the default answer is OK with the user, he need only press ENTER to accept the default. If it is not OK, he can enter the proper value. Of course, this behavior would need to be documented somewhere in the instructions given to the user.

# LINPUT Statement

The LINPUT statement differs slightly from the INPUT statement. Only one value can be input with each LINPUT statement, the variable must be a simple string, string element, or sub-string. All characters typed, including commas and quotation marks, are stored in the variable. Only the end-of-line character will delimit the data.

# OUTPUT KBD

OUTPUT KBD sends keystrokes to the keyboard buffer, just as if someone had typed them. This can be useful in giving the user a default response for INPUT that she can edit:

```
10   OUTPUT KBD;"STUDENT.DAT";
20   INPUT "Enter a filename:",F$
30   END
```

Function keys can be simulated by sending the two character sequences to the KBD device. The first character is a CHR$(255). This value is followed by a character that specifies the function key. A table giving the second character codes is found in Chapter 4 of the *Installing and Using* manual.

```
10    OUTPUT KBD;"""AUTOST File Complete"""&CHR$(255)&"E";
20    WAIT 1
30    OUTPUT KBD;CHR$(255)&"!"&"SCRATCH"&CHR$(255)&"E";
40    END
```

The first line in this example shows how to display your own messages on the message line. The ENTER key is output to the keyboard with the two characters, CHR$(255)&"E". The semicolon at the end of the line prevents carriage-return/line-feed characters from being sent to the keyboard. Line 30 is a handy line to execute at the end of your AUTOST file. The CHR$(255)&"!" characters are the STOP key, followed by a SCRATCH statement. This will cause your AUTOST file to be SCRATCHed from memory after it is executed.

The CHR$(255) character displays as a reverse-video "K" on an HP BASIC Workstation. On other computers, it displays differently. PCs using standard PC character sets (code pages) display the character as a space. It is possible to load an HP compatible character set. See Chapter 12, "International Language Support."

**CONTROL/STATUS KBD**

The CONTROL KBD statement can be used to control various keyboard attributes, while the STATUS KBD can be used to read the status of those attributes. A complete list of the CONTROL and STATUS registers for the keyboard is given at the end of this chapter. The following is an example:

```
CONTROL KBD,16;1
```

This register disables all scrolling keys. This is useful if you have displayed a screen image that you wish to prevent the user from scrolling off the screen.

# Using a Printer

A printer can be handled just like any other device. To use the printer, you need to know the device selector for it. The most common device selectors for printers are 10, 9, 26 and 701. The Windows default printer using the Print Manager is 10. The default device selector for the first centronix port is 26 (LPT1 on a PC). The default device selector for first serial port is 9 (COM1 on a PC). The default device selector for an IEEE-488 printer with primary address 1 is 701. These values are default values and may be different on your system if you specified different values when you loaded the interface device drivers.

```
100 SUB Print4(A$)    ! subprogram to support 4 printers at once
110   OUTPUT 9;A$     ! first the serial printer
111   PRINTER IS 10   ! second the Windows default printer
112   PRINT A$
113   PRINTER IS CRT  ! back to the CRT
120   OUTPUT 26;A$    ! next the parallel printer
130   OUTPUT 701;A$   ! last the IEEE-488 printer
140 SUBEND
```

This example is a subprogram for a user who wishes to use four printers simultaneously. One printer is hooked to the RS-232 port, one to the Centronix port, and one to the IEEE-488 bus. Print4(String-expression) is used to send output to all three printers. The point is that a printer can be treated just like any other device.

# The PRT function

Just as the special functions CRT and KBD exist because the screen and keyboard are accessed so frequently, there exists a special function, PRT, which can be used to access the printer. One problem exists, however. The ISCs for the screen and keyboard are permanently set but the printer's is not. The printer can be hooked to any interface. Or more than one printer may be hooked up at the same time. What value, then, should PRT return? HP BASIC approaches the dilemma by assigning 701 to PRT, regardless of where your printer is.

HTBasic solves the dilemma by letting you change the value of PRT. By default, it is 10, but you may execute a CONFIGURE PRT statement in your AUTOST file to change the value if needed. PRT can only assume a numeric value; it is not possible to assign a file to PRT. As shown in the Print4 example above, you don't have to use the PRT function to output to a printer. It is available as a convenience only.

```
10 PRINTER IS PRT
20 PRINT "PRT = ";PRT
30 END
```

This example prints the value of PRT to the printer. PRT, KBD, and CRT are just ordinary functions, and can be used anywhere a normal function is used.

# The PRINTER IS device

The CAT, LIST, PRINT, and XREF statements send output to the "PRINTER IS" device. By default this is the CRT, but it can be changed with the PRINTER IS statement to a printer (or any other device) or a file. Thus, screen output produced by these statements can easily be redirected to your printer.

The PRINTER IS statement has facilities for adapting printer output for various printers. The output can be set to wrap at a specified WIDTH, or wrap can be disabled altogether. The characters sent at the end of a line (EOL), CR/LF by default, can be changed to match that expected by your printer. The PRINTER IS statement can even specify that EOI be sent with the EOL for IEEE-488 printers. Many older printers lose characters sent while the printer is returning the carriage. This character loss can be prevented by specifying a delay after an EOL sequence.

When PRINTER IS is set to a file, the file is opened and the previous contents of the file are discarded. To append to the file rather than replace the current contents, use the APPEND option.

The following example program lists itself twice, once to the printer, and once to the CRT.

```
10 PRINTER IS 9   ! redirect output to the serial printer
20 LIST
30 PRINTER IS CRT ! back to the screen
40 LIST
50 END
```

# The PRINTALL IS device

The PRINTALL IS statement is related to the PRINTER IS statement. The PRINTALL IS statement assigns a logging device for operator interaction and error messages. It works in conjunction with the print-all mode. When the print-all mode is on, all messages output to the screen (including output area, DISP line, keyboard line, and message line) are printed on the PRINTALL device. When print-all mode is off, output appears only in the normal places, and no information is sent to the PRINTALL device.

The print-all mode is toggled between on and off each time the PRT ALL key is pressed. STATUS(KBD,1) returns a 1 if print-all mode is on, and 0 if it is off. A program can turn print-all mode on with CONTROL KBD,1;1 and off with CONTROL KBD,1;0

Print-all is a powerful debugging tool. Use it in connection with TRACE to print TRACE messages about program execution. Also, certain error conditions can produce more than one line of output. Only the last message is visible on the message line. With print-all on, all the messages can be read on the PRINTALL device. The PRINTALL IS command defines where these messages are printed. Sent to a printer, PRINTALL allows permanent logging of output. The PRINTALL device is set to the default CRT at startup and after a SCRATCH ALL.

# CRT and KBD Registers

The following tables give the CONTROL and STATUS registers for the CRT and KBD. No READIO or WRITEIO registers are available for these devices.

# CRT CONTROL Registers

The following CONTROL registers are supported.

0 - Set the current Print Column. The left most column is one.

1 - Set the current Print Row (line). The output area top line is one.

2 - Set Insert/Replace Mode. 1 - insert mode, 0 - replace mode.

3 - This register is undefined in both HTBasic and HP BASIC.

4 - Set/Reset Display Functions Mode. 1 - Control characters (including attribute characters CHR$(128) to CHR$(158)) are displayed instead of executed. 0 - control characters execute normally and are not displayed. This command is equivalent to the DISPLAY FUNCTIONS statement.

5 - Set the default ALPHA screen color (automatically sets registers 15, 16, and 17). This command is equivalent to the ALPHA PEN statement. For bit-mapped displays (CRTB), specify a pen number, 0 to 15. For alpha displays (CRTA), specify a color attribute character, 136 to 143:

| Value | Color |
|---|---|
| 136 | White |
| 137 | Red |
| 138 | Yellow |
| 139 | Green |
| 140 | Cyan |
| 141 | Blue |
| 142 | Magenta |
| 143 | Black |

6 - This register is undefined in both HTBasic and HP BASIC.

7 - The control register that extends the functionality of the SUSPEND INTERACTIVE

| Value | Meaning |
|---|---|
| 0 | Enable all keyboard keys. |
| 1 | isable all keyboard keys but RESET key. |
| 2 | Disable RESET key only. |
| 3 | Disable all keyboard keys. |

8 - Set the current print column of the display line. The left most column is one.

9 - This register is undefined in both HTBasic and HP BASIC.

10 - Set cursor visibility. 1 - cursor on. 0 - cursor off.

11 - CRT character mapping is not supported by HTBasic.

12 - Turn Softkey Menus (Function key labels) on or off.

| Value | Meaning |
|---|---|
| 0 | Same as 2, except that when running the menus are displayed only if an ON KEY is active in the current menu. |
| 1 | Softkey menus off. Same as KEY LABELS OFF. |
| 2 | Softkey menus on. Same as KEY LABELS ON. |

13 - Set the CRT Height. Sets the number of rows on the CRT that are actually used. The number includes the softkey menus, message line, input line, display line, a blank line, and the output area. Thus a value of 9 (the minimum allowed) provides for two lines in the output area.

14 - The Display Replacement Rule is not supported by HTBasic.

15 - Set the PRINT/DISP Color. Like CONTROL 5, but only affects the output area and the display line. This command is equivalent to the PRINT PEN statement.

16 - Set the Softkey Menu Color. Like CONTROL 5, but only affects the softkey menu color. This command is equivalent to the KEY LABELS PEN statement.

17 - Set the Input Line Color. Like CONTROL 5, but only affects the input and message lines. This command is equivalent to the KBD LINE PEN statement.

18 - The ALPHA Write-enable Mask is not supported by HTBasic. Use SEPARATE or MERGE ALPHA instead.

19 - This register is undefined in both HTBasic and HP BASIC.

20 - The ALPHA Display-enable Mask is not supported by HTBasic. Use SEPARATE or MERGE ALPHA instead.

21 - Select Compatibility Display is not supported by HTBasic. Use PLOTTER IS 3 or 6 instead.

100 - Set alternate Attribute Control range. The characters in the range CHR$(128) to CHR$(143) normally control text attributes and colors. Unfortunately, some character sets, including PC code pages use this range for international character support. This register can be used to reassign the attribute and color control characters to the range CHR$(16) to CHR$(31), leaving the international characters available for display. 1 - Use the alternate range CHR$(16) to CHR$(31) for attribute control. 0 - Use the normal range CHR$(128) to CHR$(143) for attribute control. This alternate range applies only to values used with the CHR$

function. Values used with CRT registers and the ALPHA PEN, etc. statements are left unchanged.
101 - Set font size. This register is not supported by HTBasic. Use the -FN command line switch instead.
110 - Toggle dithering on or off: 0 = Dither (default), if supported by system or 1 = No dither.

# CRT STATUS Registers

The following STATUS registers are supported.

0 - Get the current Print Column. The left most column is one.

1 - Get the current Print Row (line). The output area top line is one.

2 - Get Insert/Replace Mode. 1 - insert mode, 0 - replace mode.

3 - Get the number of lines in the extended output area that are above the top line of the screen.

4 - Get Display Functions Mode. 1 - on, 0 - off.

5 - Get the default ALPHA screen color. The value does not show changes made using registers 15, 16, 17 and CHR$() character attributes.

6 - Get the ALPHA ON/OFF flag.

7 - Get the GRAPHICS ON/OFF flag.

8 - Get the current print column of the display line. The left most column is one.

9 - Get the Screen Width.

10 - Get the cursor visibility. 1 - cursor on. 0 - cursor off.

11 - Get the CRT character-mapping-disable flag. Always a 0 in HTBasic.

12 - Return the Softkey Menus mode.

13 - Get the CRT Height.

14 - Get the Display Replacement Rule. Not supported by HTBasic.

15 - Get the PRINT/DISP Color.

16 - Get the Softkey Menu Color.

17 - Get the Input Line Color.

18 - Get the alpha write-enable mask. The write-enable mask is set by the MERGE ALPHA and SEPARATE ALPHA statements on bit-mapped displays.

19 - Get the value of ALPHA MASK. Not supported by HTBasic.

20 - Get the ALPHA display-enable mask. Not supported by HTBasic.

21 - Get the compatibility mode flag. Not supported by HTBasic.

100 - Get alternate Attribute Control range flag. 1 - Alternate range, 0 - normal range.

110 - Get Dithering status: 0 = Dithering possible; 1 = No Dither.

# KBD CONTROL Registers

The following CONTROL registers are supported.

0 - Set CAPS LOCK flag. 1 - set CAPS LOCK on, 0 - set CAPS LOCK off.

1 - Turn PRINTALL mode on/off. 1 - on, 0 - off. The PRT ALL key can also be used to toggle the mode on and off. See PRINTALL IS earlier in this chapter or in the on-line *Reference Manual* for a detailed explanation of PRINTALL.

2 - Set Softkey Menu: 0 - SYSTEM Softkeys, 1 - User Softkey menu 1, 2 - User Softkey menu 2, 3 - User Softkey menu 3.

3 - Set keyboard typematic repeat interval. This register is not supported. Use the Windows keyboard control panel to set the repeat interval.

4 - Set delay before typematic action starts. This register is not supported. Use the Windows keyboard control panel to set the delay.

5 - This register is undefined in both HTBasic and HP BASIC.

6 - This register is undefined in both HTBasic and HP BASIC.

7 - Disable Keyboard Interrupts:

| Value | Meaning |
|---|---|
| 0 | Enable all keyboard keys. |
| 1 | Disable all keyboard keys but RESET key. |
| 2 | Disable RESET key only. |
| 3 | Disable all keyboard keys. |

8 - This register is undefined in both HTBasic and HP BASIC.

9 - This register is undefined in both HTBasic and HP BASIC.

10 - This register is undefined in both HTBasic and HP BASIC.

11 - Knob Pulse Mode is not supported by HTBasic.

12 - Set EOI flag. 1 - If CTRL-E is entered, then EOI is sent with the next character that is entered. 0 - CTRL-E has no special meaning.

13 - Katakana mode is not supported by HTBasic.

14 - Set base softkey number. 0 - Lowest softkey will be softkey 1 (default), 1 - Lowest softkey will be softkey 0. This register has no affect with KBD CMODE ON.

15 - Turn KBD CMODE ON/OFF. A non-zero value turns KBD CMODE ON for Nimitz compatibility. The Nimitz Keyboard is the 98203 keyboard used on the 9836. It has ten function keys, and the lowest function key is 0. A zero value turns KBD CMODE OFF, ITF compatibility (the default). The ITF Keyboard is the 46020 Keyboard used on Series 300 computers. It connects to the computer using the HIL interface, has eight function keys, and the lowest function key is 1. This command is equivalent to KBD CMODE {ON|OFF}.

16 - Disable scrolling keys: UP, DOWN, PREV, NEXT, BEGIN, and END. This allows a program to freeze the screen display, not allowing the user to scroll it off. A non-zero value disables scrolling and a zero value enables it.

100 - Controls the "Program Modified" dialog received when attempting to LOAD, GET, SCRATCH, or QUIT when the current program has been modified. The default is 1 or ON (QUIT ALL warning appears).

101 - Controls the auto update to new program dialog warning. Controls the "Overwrite Previous Version" dialog warning for overwriting a previous version file format with the converted file format. A value of 0 (default) does not bypass this warning message. A value of 1 bypasses the warning message.

202 - Controls performance tuning under Windows. The tradeoffs of increased HTBasic performance are decreased Windows response and decreased performance in other simultaneously executing Windows applications. The decreased Windows response is most noticeable as delayed response to mouse and keyboard input in all applications including HTBasic. Valid input is in the range of 0 to 32767, the default is 4. The performance gain with increasing value is non-linear, most improvement occurs in the bottom 10% of the parameter's range.

203 - Mouse movement interaction with ON KBD$: 0 = Allows interaction; 1 = Disallows interaction.

204 - Mouse click interaction with ON KBD$: 0 = Allows interaction; 1 = Disallows interaction.

206 - Toggle ALT key behavior: 0 = Windows default; 1 = HTBasic settings.

207 - The Graphics Speed tuning register allows dynamic thread priority setting. With the thread priority set to NORMAL, (i.e. 0) graphics will run quicker. If the user needs the ON KNOB or ON CYCLE to work properly, set KBD 207 to 3. This will give the mouse better response time, but will compromise graphic speed. To give graphics a higher priority, set KBD 207 to -3.

210 - Toggle QUIT behavior: 0 = QUIT (closes the program child window); 1 = QUIT ALL (closes HTBasic application as does the QUIT ALL command).

# KBD STATUS Registers

The following STATUS registers are supported.

0 - Get CAPS LOCK flag. Under the X Windows System, this register is undefined.

1 - Get Print All mode state.

2 - Get Softkey Menu number.

3 - This register is undefined in both HTBasic and HP BASIC.

4 - This register is undefined in both HTBasic and HP BASIC.

5 - Get the KBD$ Buffer Overflow Flag. A one means an overflow has occurred since the last time the register was read. Reading the register sets the flag to zero.

6 - Get the softkey macro expansion overflow flag. 1-overflow. Reading this register resets it to 0.

7 - Return Keyboard Interrupt Disable Mask.

8 - Return Keyboard Language. Always 0 - US ASCII.

9 - Return Keyboard Type. Always 0 - "Other Keyboard."

10 - Return State of Shift keys at the time of the last KNOB event.

| Value | Meaning |
|---|---|
| 0 | Neither key pressed |
| 1 | SHIFT key pressed |
| 2 | CTRL key pressed |
| 3 | Both keys pressed |

11 - Get Horizontal/All Pulse Mode flags. Always 0 - horizontal-pulse mode.

12 - Get EOI flag.

13 - Get Katakana flag.

14 - Get base Softkey number. 1 - base is 0, 0 - base is 1.

15 - Get keyboard compatibility flag. 0 - ITF, 1 - Nimitz.

16 - Get scrolling disable flag. 1 - disabled, 0 - enabled.

100 - Gets the "Program Modified" dialog status.

202 - Returns status of the performance tuning register.

203 - Returns status of mouse movement interaction with ON KBD$.

204 - Returns status of mouse click interaction with ON KBD$.

206 - Returns ALT key status: 0 = Windows default; 1 = HTBasic settings.

207 - Returns status of the Graphics Speed tuning register.

210 - Returns QUIT behavior status.

# Summary

This chapter discussed the general I/O statements (ENTER, OUTPUT, STATUS, and CONTROL) as they apply to CRT and KBD. The special statements for screen and printer were presented: PRINT, DISP, PRINTER IS, and PRINTALL IS. The mnemonic functions CRT, KBD, and PRT were explained. Attribute control characters were given. CONTROL and STATUS registers for CRT and KBD were listed.

# Files

This chapter explains how to perform input and output to files. File management commands are presented. The different file types are explained, random and sequential file access examples are given, and file formats are discussed. An example program showing how to convert from one file type to another is shown.

Chapter 5, "General Input and Output," discussed the general principles used for input/output (I/O). These principles apply to files as well as other I/O targets. In particular, use of ASSIGN, OUTPUT, ENTER, STATUS, and CONTROL were explained. If you have not yet read that chapter, you should do so before reading this chapter.

A file is a collection of data that is kept on disk rather than in the computer's memory. When the computer is turned off, the data in the computer memory is lost but the data in a file is not.

# File Management Commands

Several commands are available for managing files and the file system. It should be remembered that a major difference between HTBasic and workstation HP BASIC is that HTBasic is the guest of an operating system and HP BASIC *is* the operating system. As a guest of an operating system, HTBasic must live by the rules established by that system.

Please see the on-line *Reference Manual* for a more detailed description of each of these statements:

# ASSIGN

ASSIGN is the equivalent to the OPEN command in other computer languages. It was explained in Chapter 5.

```
10    ASSIGN @Io TO "C:\RMB\HP-UX.BAT";FORMAT ON,RETURN S
20    IF S THEN
30      PRINT "Oops, Error";S;"opening the file"
40      PAUSE
50    END IF
60    OUTPUT @Io;"cd \hp-ux"
70    OUTPUT @Io;"kermit take hp320.tak",END
80    ASSIGN @Io TO *
90    EXECUTE "hp-ux"
100   END
```

This example shows the ASSIGN command being used to access a file called "HP-UX.BAT". (The file must already exist. Use CREATE to create a new file.) If the ASSIGN takes place correctly, then OUTPUT is used to send some data to the file. The END in the last OUTPUT causes the file to be truncated at that point. This is useful if the old file contents were longer than the new contents. Finally, "ASSIGN ... *" is used to close the I/O path associated with the @Io variable.

# CAT

CAT displays a mass storage device directory (catalog of files), or PROG file subprogram information. Several options are available. A path-specifier may be included to show which device or part of a file system to catalog. The catalog may be displayed on the current PRINTER IS device, or sent to another device or a string array. A COUNT of the entries can be returned. Only the names of the files, or complete information about the files, can be selected. Header information about the file system can be displayed or suppressed. All or only part of the files can be selected for display. For example:

```
CAT "*.BAS"
```

# CHGRP and CHOWN

CHGRP and CHOWN are useful with an operating system like UNIX in which files are owned by individuals and groups. These commands allow a user with the appropriate privilege to change or assign ownership of files. These commands are not used in HTBasic.

# COPY

COPY is used to make a copy of a file. If a file already exists with the destination name, an error is normally returned. To suppress the error the ";PURGE" option may be specified at the end of the statement. An example of the COPY statement is

```
COPY "DATA.1" TO "A:DATA.1"
```

HTBasic does not support the copy of a full disk to another disk. You can use the EXECUTE command and the DOS "DISKCOPY" or "XCOPY" commands.

# CREATE

CREATE is used to create a new file or directory. Although SAVE and STORE will automatically create a new file to store a program in, data files must be explicitly created before they can be used in ASSIGN, DUMP DEVICE IS, PRINTALL IS, or PRINTER IS statements.

The four forms of the CREATE statement are CREATE, CREATE ASCII, CREATE BDAT, and CREATE DIR. The plain CREATE statement creates an ordinary file (DOS, NT, or UNIX). CREATE ASCII and CREATE BDAT create LIF ASCII or BDAT type files (more on these later). CREATE DIR creates directories.

The CREATE command specifies the maximum number of records to allocate for the file. However, Windows allows the maximum size of a file to be extended anytime the maximum number of records is exceeded. The number of records specified in the CREATE command is ignored and space for the file is allocated only as needed.

```
CREATE "TEMP.TXT",0
CREATE BDAT "DATA.DAT",12,34
```

# INITIALIZE

INITIALIZE is used to format a new disk. Used on an old disk, it completely erases all previous contents of the disk. HTBasic does not support INITIALIZE. To initialize a new LIF disk, use an HP BASIC workstation. To initialize a PC disk, use the File Manager. Select "Disk" and then "Format Disk...".

To programmatically FORMAT a disk, use the EXECUTE statement to call the FORMAT command:

```
EXECUTE "FORMAT A:"
```

# LINK

LINK creates a new directory entry for an existing file. This is called a hard link. After creating the link, both the old name and the new name refer to the same file. LINK is not supported by HTBasic.

# LOCK and UNLOCK

LOCK/UNLOCK are used to secure (or release) a file for exclusive use. These commands are designed for use on multitasking or network systems to prevent two users or two processes from using the same file at the same time, preventing them from making conflicting transactions.

```
10    ASSIGN @Path TO "airline.seats"
20    ASSIGN @Travelagency TO 705
30    ENTER @Travelagency;Requested
40    REPEAT
50      LOCK @Path;CONDITIONAL Notlocked
60      Locked= NOT Notlocked
70    UNTIL Locked
80    ENTER @Path;Available,Booked
90    IF Requested>Available THEN
100     UNLOCK @Path
110     OUTPUT @Travelagency;"Only ";Available;" available"
120   ELSE
130     RESET @Path
140     Available=Available-Requested
150     Booked=Booked+Requested
160     OUTPUT @Path;Available,Booked
170     UNLOCK @Path
180     OUTPUT @Travelagency;"OK"
190   END IF
200   ASSIGN @Path TO *
210   END
```

# MASS STORAGE IS

MASS STORAGE IS and the abbreviation, MSI, allow you to specify the device and path specifier to be used by default when no explicit device and path specifier are given. For example, a CAT command, without a path specifier will display files from the default path specifier. As another example, these two programs ASSIGN the same two files:

```
10  MSI "A:"                    10 ASSIGN @File1 TO "A:FILE1"
20  ASSIGN @File1 TO "FILE1"    20 ASSIGN @File2 TO "B:FILE2"
30  MSI "B:"
40  ASSIGN @File2 TO "FILE2"
```

# PERMIT

PERMIT is used under UNIX to set the permissions (mode) of a file, directory, or device. To change file attributes with HTBasic, use the PROTECT statement.

# PRINT LABEL and READ LABEL

PRINT LABEL and READ LABEL are used to set and read the volume label of a disk drive. HTBasic does not support PRINT LABEL; you must use the DOS "LABEL" command or the File Manager. This example shows the use of READ LABEL:

```
10    MASS STORAGE IS "C:"
20    READ LABEL A$
30    IF A$="No Label" THEN
40      PRINT "The volume in drive C has no label"
50    ELSE
60      PRINT "The volume in drive C is ";A$
70    END IF
80    END
```

# PROTECT

PROTECT is used to set LIF file passwords under HP BASIC and file attributes under HTBasic.

A special form of PROTECT is used by HTBasic to change file attributes. The syntax is

PROTECT *file-specifier, protect-code*

where *protect-code* is a string containing zero or more of the following characters:

| Character | Meaning |
|---|---|
| (none) | no protection |
| R | Read-only: File cannot be written or deleted. |
| S | System file: This attribute usually has no meaning. |
| H | Hidden: File will not be listed by CAT. |

If a character is not included, that attribute is cleared. If the string is blank, all attributes are cleared. For example:

```
PROTECT "FILE1","R"
```

# PURGE

PURGE is used to delete files and directories.

```
PURGE "FILE1"
```

# RENAME

RENAME is used to change the name of a file, but can also be used to move a file from one directory to another directory or disk.

```
RENAME "C:\HTB\AUTOST" TO "C:\AUTOST.BAS"
```

# RESET

When RESET is used with a file, the file position is set to the beginning of the file.

```
10     ASSIGN @I TO "TEMP.TXT";FORMAT ON
20     OUTPUT @I;"HELLO"
30     RESET @I
40     ENTER @I;A$
50     PRINT A$
60     END
```

# SYSTEM$("MSI")

SYSTEM$("MSI") allows you to read the present MASS STORAGE IS path specifier.

```
PRINT "The present MSI is ";SYSTEM$("MSI")
```

# WILDCARDS

The WILDCARDS statement enables or disables wildcard support. Wildcards are characters that can be used in a filename as a template to select a group of files to be operated upon. A filename with wildcard characters in it will be compared with existing filenames using special rules and all filenames that "match" are acted upon. Under HTBasic, wildcards are supported only in the CAT statement.

If the WILDCARDS statement is executed, it will return an error because wildcarding is always on. The question mark "?" and the asterisk "*" are the wildcard characters.

# File Types

HTBasic supports several file types. Typed files for data are LIF ASCII and BDAT. HTBasic also supports files without a file type. HTBasic calls such files ordinary files. HP BASIC calls them HP-UX files. Most files are ordinary files. The format of data written to these files with FORMAT OFF is explained in Chapter 5.

# BDAT Files

BDAT files, by default, are FORMAT OFF and are used to hold binary data. They may also be ASSIGNed with FORMAT ON and used to hold ASCII data. BDAT files may be accessed sequentially or randomly. The record size for random access is established when the file is CREATEd. If not specified, it defaults to 256 bytes.

# ASCII Files

ASCII files are LIF ASCII files and are not compatible with DOS ASCII or UNIX ASCII files. See "Ordinary Files" below to learn how to create a DOS ASCII or UNIX ASCII file. A LIF ASCII file is compatible with HP BASIC ASCII files and is most useful when exchanging data using LIF floppies. In a CAT listing, a LIF ASCII file is listed as file type "ASCII". LIF ASCII files are always written with FORMAT ON and can only be accessed sequentially.

# Ordinary Files

Ordinary files are files that do not have a file type. HP added ordinary files to HP BASIC 5.0 and called them "HP-UX" files. The name is somewhat misleading since the same file is called a "DOS" file when on a PC.

By default, ordinary files are written with FORMAT OFF, but FORMAT ON may also be used. If you wish to create or access a DOS ASCII file, use an ordinary file with FORMAT ON. A DOS ASCII file contains characters that are all in the printable ASCII range, and lines are terminated with a carriage return, line feed (CR/LF) sequence. If you wish to create or access a UNIX ASCII file, use an ordinary file with FORMAT ON and EOL CHR$(10). Lines are terminated in a UNIX ASCII file with just a line feed.

```
10   CREATE "TEMP.TXT",512    ! CREATE Ordinary file
20   ASSIGN @X TO "TEMP.TXT";FORMAT ON
! for DOS 30   INTEGER I
40   FOR I=1 TO 10
50     OUTPUT @X;"Line #";I
60   NEXT I
70   ASSIGN @X TO *
80   END
```

# File Organization

"Random" and "sequential" are not file types, but are methods of organizing and accessing the information in a file. In fact, a BDAT or ordinary file can be accessed either way, even in the same program. A file should be organized, sequentially or randomly, based on how the file will be used.

# Sequential Files

In sequential files, each data item is stored immediately following the previous one. The data in the file is stored in the order that it is produced. The data is read in the same order that it is stored. You read a sequential file from beginning to end. This results in a compact data storage structure and ease of programming.

The following example uses three files with sequential organization. The first two contain sorted data. They are merged to create the third file. Merging two files lends itself well to sequential organization.

```
10    REM Merge two sorted files
20    CREATE ASCII "merged",10
30    ASSIGN @F1 TO "file1";FORMAT ON
40    ASSIGN @F2 TO "file2";FORMAT ON
50    ASSIGN @M TO "merged";FORMAT ON
60    DIM Key1$[80],Key2$[80]
70    ON END @F1 GOTO Endf1
80    ON END @F2 GOTO Endf2
90    ENTER @F1;Key1$
100   ENTER @F2;Key2$
110   LOOP
120     IF Key1$>Key2$ THEN
130       OUTPUT @M;Key2$
140       ENTER @F2;Key2$
150     ELSE
160       OUTPUT @M;Key1$
170       ENTER @F1;Key1$
180     END IF
190   END LOOP
200 Endf1:! only file2 has any more data
210   ON END @F2 GOTO Alldone
220   LOOP
230     OUTPUT @M;Key2$
240     ENTER @F2;Key2$
250   END LOOP
260 Endf2:! only file1 has any more data
270   ON END @F1 GOTO Alldone
280   LOOP
290     OUTPUT @M;Key1$
300     ENTER @F1;Key1$
310   END LOOP
320 Alldone:! both files are out of data
330   ASSIGN @F1 TO *
340   ASSIGN @F2 TO *
350   ASSIGN @M TO *
360   END
```

# Random Access Files

As mentioned previously, "random" is not a file type, but a method of organizing and accessing the information in a file. A file should be organized based on how the file will be used. Files in which the items are accessed in a random order should be organized as a random file.

**BDAT** files contain fixed length records that can be accessed by record number. The record number is specified after the I/O path variable. The record length is specified when creating the file. The record size must be set to accommodate the largest data item.

```
10    REM Random file example (BDAT File)
20    DIM C$[56]
30    CREATE BDAT "customer.dat",100,60
40    ASSIGN @C TO "customer.dat"
50    CLEAR SCREEN
60    LOOP
70      DISP "A(dd, D(elete, S(how, Q(uit and take a vacation"
80      ON KBD GOTO Inkey
90      LOOP !endlessly until a key is pressed
100       OUTPUT CRT;TIME$(TIMEDATE);CHR$(13);
110     END LOOP
120 Inkey: K$=KBD$
130     OFF KBD
140     OUTPUT CRT
150     SELECT UPC$(K$)
160     CASE "A"
170       PRINT "Add:"
180       INPUT "Customer Number? ",C
190       INPUT "Information? ",C$
200       OUTPUT @C,C;C$
210       PRINT "Customer number #";C;"added"
220     CASE "D"
230       PRINT "Delete:"
240       INPUT "Customer Number? ",C
250       OUTPUT @C,C;"DELETED"
260       PRINT "Customer number #";C;"deleted"
270     CASE "S"
280       PRINT "Show:"
290       INPUT "Customer Number? ",C
300       ENTER @C,C;C$
310       PRINT "Customer number #";C;":",C$
320     CASE "Q"
330       PRINT "Thank you for using HTBasic!"
340       PRINT "Have a nice vacation."
350       DISP ! clear display line
360       STOP
370     CASE ELSE
380       PRINT CHR$(7);! ring the bell for a bad command
390     END SELECT
400   END LOOP
410   END
```

This example, of course, is not a complete application. But it does show the important aspects of random file use, as well as some user interface techniques. Note that the record size was declared to be 60. The length of each record can never exceed this, since each record consists of C$ (which can never be longer than 56 characters) plus the four byte length of C$ which we know will be included in the file since we are using a BDAT file with FORMAT OFF (the default).

**Ordinary files** do not have a physical record length, but you can still use a logical record length. The record number actually specifies the exact byte position in the file. The first byte is at position 1. To access a logical record, the byte position must be calculated based on the logical record length. The following example has the same capabilities as the previous program, but uses an ordinary file.

```
10    REM Random file example (Ordinary File)
```

```
20    DIM C$[58]
30    Length=60                  ! 58 Character string + CR/LF
40    CREATE "customer.dat",100
50    ASSIGN @C TO "customer.dat";FORMAT ON
60    CLEAR SCREEN
70    LOOP
80      DISP "A(dd, D(elete, S(how, Q(uit and take a vacation"
90      ON KBD GOTO Inkey
100     LOOP              !endlessly until a key is pressed
110       OUTPUT CRT;TIME$(TIMEDATE);CHR$(13);
120     END LOOP
130 Inkey: K$=KBD$
140     OFF KBD
150     OUTPUT CRT
160     SELECT UPC$(K$)
170     CASE "A"
180       PRINT "Add:"
190       INPUT "Customer Number? ",C
200       INPUT "Information? ",C$
210       OUTPUT @C,(C-1)*Length+1;C$
220       PRINT "Customer number #";C;"added"
230     CASE "D"
240       PRINT "Delete:"
250       INPUT "Customer Number? ",C
260       OUTPUT @C,(C-1)*Length+1;"DELETED"
270       PRINT "Customer number #";C;"deleted"
280     CASE "S"
290       PRINT "Show:"
300       INPUT "Customer Number? ",C
310       ENTER @C,(C-1)*Length+1;C$
320       PRINT "Customer number #";C;":",C$
330     CASE "Q"
340       PRINT "Thank you for using HTBasic!"
350       PRINT "Have a nice vacation."
360       DISP ! clear display line
370       STOP
380     CASE ELSE
390       PRINT CHR$(7);! ring the bell for a bad command
400     END SELECT
410   END LOOP
420   END
```

BDAT files give an error if a single OUTPUT is too long for the record length (unless the record length is one). However, ordinary files do not give an error. It is the programmer's job to make sure that record overflow does not occur.

# Converting LIF ASCII files to DOS ASCII

Sometimes it is advantageous to translate from a LIF ASCII file type to an ordinary file so that other programs can make use of the data.

The following program is an example showing the general principles of converting from one data type to another. It works for LIF ASCII to ordinary ASCII conversions, but may need to be modified to work in other situations.

```
10     REM ASCIIDOS.BAS
20     DIM Fi$[30],Fo$[30],L$[256]
30     INPUT "Input file?",Fi$
40     INPUT "Output file?",Fo$
50     ASSIGN @I TO Fi$;FORMAT ON
60     CREATE Fo$,1
70     ASSIGN @O TO Fo$;FORMAT ON
80     ON END @I GOTO Done
90     LOOP
100      ENTER @I;L$
110      OUTPUT @O;L$
120    END LOOP
130 Done: END
```

Knowing several general principles will help you write conversion programs to work in whatever situation you require. Line 50: Open the input file using FORMAT ON, OFF, MSB FIRST, or whatever is appropriate. Line 60: Create an output file of the type you wish to create. Line 70: Open the output file using FORMAT ON, OFF, MSB FIRST, or whatever is appropriate. Line 100: Enter the data with a statement compatible with how the data was written. For example, if integers were written in binary, then enter into integers with the file opened for binary access. Line 110: Output the data in the format you wish it to be in. For example, if you wish three integers separated by commas, make sure you know how to use the OUTPUT statement to do so. Finally, use loop constructs (FOR, LOOP, REPEAT, or WHILE) to handle groups of data that are formatted the same.

# Summary

This chapter presented statements associated with file system management. The different file types were explained, and random and sequential file access examples were given.

# IEEE-488 Interface Bus

This chapter discusses the IEEE-488 (GPIB or HP-IB) bus and the HTBasic statements used to transfer information between devices. The history of the bus is presented along with an overview of its signal lines and device addressing. The different levels of IEEE-488 bus data transfer and control statements are also presented along with the HTBasic statements that enable and control IEEE-488 interrupts. A list of CONTROL, STATUS, READIO, and WRITEIO registers for the IEEE-488 is given. A summary of the bus actions that each IEEE-488 statement generates is also included.

This chapter does not explain installation of the IEEE-488 board or device driver. The *Installing and Using* manual contains the necessary installation and configuration information for device drivers included with HTBasic. For device drivers sold separately, the documentation included with the driver explains how to load and configure the driver.

This chapter assumes that you already have some familiarity with the operation of IEEE-488 bus and does not include a detailed bus operation description. Please consult any one of the many available books about the IEEE-488 bus for more detailed information about its operation.

# IEEE-488 History

The IEEE-488 bus was developed to connect and control programmable instruments and to provide a standard interface for communication between instruments from different sources. Hewlett-Packard originally developed the interfacing technique and called it HP-IB. The interface quickly gained popularity in the computer industry. Because the interface was so versatile, the IEEE committee renamed it GPIB (General Purpose Interface Bus). All references to this interface bus in this chapter will use the name IEEE-488.

# IEEE-488 Overview

Almost any instrument can be used with the IEEE-488 specification, because it says nothing about the function of the instrument itself, or about the form of the instrument's data. Instead the specification defines a separate component, the interface, that can be added to the instrument. The signals passing into the interface from the IEEE-488 bus and from the instrument are defined in the standard. The instrument does not have complete control over the interface. Often the bus controller tells the interface what to do. The active controller performs the bus control functions for all the bus instruments.

At power-up time, the IEEE-488 card that is programmed to be the system controller becomes the active controller in charge. The system controller has several unique capabilities including the ability to send Interface Clear (IFC) and Remote Enable (REN) commands. IFC clears all device interfaces and returns control to the system controller. REN allows devices to respond to bus data once they are addressed to listen. The system controller may optionally Pass Control to another controller, which then becomes active controller.

There are 3 types of devices that can be connected to the IEEE-488 (Listeners, Talkers, and Controllers). Some devices include more than one of these functions. The standard allows a maximum of 15 devices to be connected on the same bus. A minimum system consists of one controller and one talker or listener device (i.e., a PC with a TransEra GPIB-900 board and a voltmeter).

It is possible to have several controllers on the bus but only one may be active at any given time. The active controller may pass control to another controller which in turn can pass it back or on to another controller. A listener is a device that can receive data from the bus when instructed by the controller and a talker transmits data on to the bus when instructed. The controller can set up a talker and a group of listeners so that it is possible to send data between groups of devices as well.

The IEEE-488 interface system consists of 16 signal lines and 8 ground lines. The 16 signal lines are divided into 3 groups (8 data lines, 3 handshake lines, and 5 interface management lines).

# Data Lines

The lines DIO1 through DIO8 are used to transfer addresses, control information and data. The formats for addresses and control bytes are defined by the IEEE-488 standard. Data formats are undefined and may be ASCII (with or without parity) or binary. DIO1 is the Least Significant Bit (note that this will correspond to bit 0 on most computers).

# Handshake Lines

The three handshake lines (NRFD, NDAC, DAV) control the transfer of message bytes among the devices and form the method for acknowledging the transfer of data. This handshaking process guarantees that the bytes on the data lines are sent and received without any transmission errors and is one of the unique features of the IEEE-488 bus.

The NRFD (Not Ready for Data) handshake line is asserted by a listener to indicate it is not yet ready for the next data or control byte. Note that the controller will not see NRFD released (i.e., ready for data) until all devices have released it.

The NDAC (Not Data Accepted) handshake line is asserted by a listener to indicate it has not yet accepted the data or control byte on the data lines. Note that the controller will not see NDAC released (i.e., data accepted) until all devices have released it.

The DAV (Data Valid) handshake line is asserted by the talker to indicate that a data or control byte has been placed on the data lines and has had the minimum specified stabilizing time. The byte can now be safely accepted by the devices.

The handshaking process is outlined as follows. When the controller or a talker wishes to transmit data on the bus, it sets the DAV line high (data not valid) and checks to see that the NRFD and NDAC lines are both low, then it puts the data on the data lines.

When all the devices that can receive the data are ready, each releases its NRFD (not ready for data) line. When the last receiver releases NRFD and it goes high, the controller or talker takes DAV low indicating that valid data is now on the bus.

In response each receiver takes NRFD low again to indicate it is busy and releases NDAC (not data accepted) when it has received the data. When the last receiver has accepted the data, NDAC will go high and the controller or talker can set DAV high again to transmit the next byte of data.

Note that if after setting the DAV line high, the controller or talker senses that both NRFD and NDAC are high, an error will occur. Also, if any device fails to perform its part of the handshake and releases either NDAC or NRFD, data cannot be transmitted over the bus. Eventually a timeout error will be generated.

The speed of the data transfer is controlled by the response of the slowest device on the bus; for this reason it is difficult to estimate data transfer rates on the IEEE-488 bus as they are device dependent.

# Interface Management Lines

The five interface management lines (ATN, EOI, IFC, REN, SRQ) manage the flow of control and data bytes across the interface.

The ATN (Attention) signal is asserted by the controller to indicate that it is placing an address or control byte on the data bus. ATN is released to allow the assigned talker to place status or data on the data bus. The controller regains control by reasserting ATN; this is normally done synchronously with the handshake to avoid confusion between control and data bytes.

The EOI (End or Identify) signal has two uses. A talker may assert EOI simultaneously with the last byte of data to indicate end-of-data. The controller may assert EOI along with ATN to initiate a parallel poll. Although many devices do not use parallel poll, all devices should use EOI to end transfers (many currently available devices do not).

The IFC (Interface Clear) signal is asserted only by the system controller in order to initialize all device interfaces to a known state. After releasing IFC, the system controller becomes the active controller.

The REN (Remote Enable) signal is asserted only by the system controller. Its assertion does not place devices into remote control mode; REN only enables a device to go into remote mode when addressed to listen. When in remote mode, a device should ignore its local front panel controls.

The SRQ (Service Request) line is like an interrupt: it may be asserted by any device to request the controller to take some action. The controller must determine which device is asserting SRQ by conducting a serial poll. The requesting device releases SRQ when it is polled.

# Device Addresses

The IEEE-488 standard allows up to 15 devices to be interconnected on one bus. Each device is assigned a unique primary address, ranging from 0-30, by setting the address switches on the device. A secondary address may also be specified, ranging from 0-30. See the device documentation for more information on how to set the device primary and optional secondary address.

In the HTBasic statements that access the bus, a device selector is used to specify the interface select code, the primary device address, and the optional secondary device address. The default IEEE-488 interface select code is 7. The default primary address of the system controller is 21. The following examples demonstrate how the interface and device addresses are specified.

| Device Selectors | ISC code | Pri. Add | Sec. Add |
|---|---|---|---|
| 705 | 7 | 5 | none |
| 72501 | 7 | 25 | 1 |
| 1207 | 12 | 7 | none |
| 100412 | 10 | 4 | 12 |

The primary address of the IEEE-488 board can be read using the STATUS statement and changed with the CONTROL statement. Bits 0-4 of register three specify the primary address.

```
STATUS 7,3; Pri_add     !Read Primary Address
CONTROL 7,3; Pri_add    !Set Primary Address
```

A discussion of the CONTROL and STATUS statements is given later in this chapter.

# IEEE-488 Statement Overview

HTBasic provides five levels of IEEE-488 statements: high level transfer, high level bus control, byte level transfer, low level bus control, and interface interrupt control.

The high level OUTPUT and ENTER statements allow you to easily send and receive data on the bus. All the necessary bus addressing commands are automatically generated.

The high level bus control statements allow you to abort transfers, reset the bus interface, clear specific bus devices, lockout local control of devices, return devices to their local state, pass active control to another, configure the parallel poll response, request service, perform a group execute trigger, and conduct parallel and serial polls.

The byte level SEND statement allows more detailed control over the bus. Because the user must generate all the proper bus addressing commands, use of this statement requires a more detailed knowledge about IEEE-488 bus operations.

The low level bus control statements CONTROL, STATUS, READIO, and WRITEIO allow you to directly access the IEEE-488 driver status and control registers and the controller hardware registers.

The interrupt control statements enable, control, and disable interrupts generated by the IEEE-488 interface hardware.

# High Level Transfer Statements

The ENTER and OUTPUT statements are used to transfer data between IEEE-488 devices. They automatically generate all the required bus addressing. For a description of the ENTER and OUTPUT statements, see Chapter 5, "General Input and Output." It explains how to send or suppress CR/LF line terminators and how to set the EOI signal line on the output of the last data byte. The following example demonstrates communication with an HP-GL plotter at device address five.

```
OUTPUT 705;"OP;"
ENTER 705;P1x,P1y,P2x,P2y
```

The OUTPUT statement requests the plotter to send its P points. The ENTER statement reads the P point values sent back by the plotter.

The powerful USING option gives you a high degree of control over the data format used for the transfer operations. For example:

```
OUTPUT 705 USING "#,K";Str$
ENTER 705 USING "#,K";Str$
```

Multiple listeners may also be addressed with the same command. As follows:

```
ASSIGN @Dev to 705,706,707,708
OUTPUT @Dev; "Data"
```

The OUTPUT statement listen-addresses the devices with primary addresses 5, 6, 7, and 8 and then sends the string "Data" to all of them. If the same I/O Path is used for the ENTER statement, the first device is addressed as the talker and the remaining devices, including the active controller, are addressed as listeners.

Some devices allow the selection of a particular mode of operation by the use of the secondary address. Multiple secondary addresses may be specified. This extended addressing mode is shown below.

```
ASSIGN @Dev to 7011011    !Secondary Address 10 and 11
OUTPUT @Dev; Str$
OUTPUT 70501; Str$        !Secondary Address 01
```

When the device is not the active controller, it cannot do any bus addressing. If only the interface select code is used for the ENTER and OUTPUT statements, no bus addressing will be performed. The device must make sure that it has been addressed to talk or listen before it participates in the transfer of data. If it has not been addressed, then the device will wait until it is addressed before continuing.

```
OUTPUT 7; Str$
ENTER 7; Str$
```

# High Level Bus Control Statements

HTBasic provides many high level IEEE-488 bus control statements. The actions taken on the IEEE-488 bus by each control statement are determined by three things:

1) whether the device issuing the command is the system controller,
2) whether the device is the active controller, and
3) whether the command was issued with only the interface select code or a primary address was specified.

Each statement is discussed in the following paragraphs. Also, a quick bus actions reference is provided at the end of this chapter.

# ABORT Statement

The ABORT statement stops IEEE-488 bus activity. You specify either an interface select code or an I/O Path. This statement is only supported by the IEEE-488 interface. For example:

```
ABORT 7
```

If the computer is the system controller but not the active controller, ABORT causes the computer to assume active control.

If a primary address is specified, an error is generated. If the computer is the system controller, the bus action is to issue IFC for greater than 100 micro-seconds and then to assert REN and de-assert ATN. If the computer is not the system controller but is the active controller, the bus action is: ATN, MTA, UNL, and de-assert ATN. If it is not the active controller either, no action is taken.

# CLEAR Statement

The CLEAR statement causes the active controller to send a Device Clear command to one or more devices. The effect on the device is device-dependent. You specify either an device selector or an I/O Path. This statement is only supported by the IEEE-488 interface. For example:

```
ASSIGN @Counter to 7
CLEAR @Counter
```

If primary addressing is specified, the bus action is: ATN, MTA, UNL, LAG, SDC. If only an interface select code is specified, the bus action is: ATN, DCL. If the computer is not the active controller, an error is generated.

# LOCAL Statement

The LOCAL statement returns specified IEEE-488 devices to their local (front panel) state. You specify either a device selector or an I/O Path. This statement is only supported by the IEEE-488 interface. For example:

```
LOCAL 728
```

If a primary device address is specified, a Go To Local (GTL) message is sent to all listeners and LOCAL LOCKOUT is not canceled. If only an interface select code is specified, all devices on the bus are returned to the local state and LOCAL LOCKOUT is canceled.

If a primary device address is specified and the computer is the active controller, the bus activity is: ATN, MTA, UNL, LAG, GTL.

If the computer is not the active controller but is the system controller and just an interface select code is specified, the REN line is set false. If it is also the active controller, the ATN and REN lines are both set false.

When the computer is not the system controller but is the active controller, the bus activity for an Interface Select Code is to set the ATN line and send a GTL message. When it is not the active controller, an error is generated.

# LOCAL LOCKOUT Statement

The LOCAL LOCKOUT statement sends the LLO message over the IEEE-488 bus. This prevents front panel control of IEEE-488 devices that are in the remote state. You specify either an interface select code or an I/O Path. This statement is only supported by the IEEE-488 interface. For example:

```
LOCAL LOCKOUT 7
```

If the computer is not the active controller or a primary device address is specified, an error is generated. If only an interface select code is specified, the bus action is ATN, LLO. If an I/O Path is specified, it must refer to the IEEE-488 interface.

# PASS CONTROL Statement

The PASS CONTROL statement passes active controller capability to the specified IEEE-488 device. You specify either a device selector or an I/O Path. If an I/O Path is specified, it must be assigned to an IEEE-488 device. For example:

```
ASSIGN @Dev to 705
PASS CONTROL @Dev
```

If the computer is the active controller and a primary address is specified, control is passed to the addressed device. An error is generated if the computer is not the active controller or if only an interface select code is specified.

The specified device is talk addressed, a Take-Control-Message TCT is sent, and the Attention line is set false. The computer then becomes a bus device, as opposed to a bus controller.

# PPOLL Function

The PPOLL function conducts a Parallel Poll of the IEEE-488 and the 8-bit status message from the IEEE-488 bus is returned. Each bit corresponds to the status of a device which is configured to respond to a parallel poll. You specify either an interface select code or an I/O Path as the function argument. This statement is only supported by the IEEE-488 interface. For example:

```
ASSIGN @Gpib to 7
Pstatus = PPOLL(@Gpib)
```

If an interface select code is specified, the bus action is as follows: ATN and EOI are set for greater than or equal to 25 microsec, one byte of data is read from the bus, EOI is released, and ATN is restored to its previous state. If the computer is not the active controller or a primary device address is specified, an error is generated.

# PPOLL CONFIGURE Statement

The PPOLL CONFIGURE statement configures the parallel poll response for the specified remote IEEE-488 device(s). You specify either an I/O Path or a device selector that refers to one or more IEEE-488 devices and a parallel poll configuration value from zero through 15. The three least significant bits of its binary representation select the data bus line and the fourth bit selects the logical sense of the response. For example:

```
PPOLL CONFIGURE 702;3
```

configures device number two on interface number seven to respond on data line DIO4 with a logic sense of zero when its status bit is set.

If the computer is not the active controller or if only an interface select code is specified, an error is generated. The bus action is as follows: ATN, MTA, UNL, LAG, PPC, PPE.

# PPOLL RESPONSE Statement

The PPOLL RESPONSE statement enables or disables the local IEEE-488 device parallel poll response to an active controller parallel poll request. You specify either an interface select code or an I/O Path and an enable value. An enable value of one enables the parallel poll response, whereas a zero value disables it. This statement is only supported by the IEEE-488 interface. For example:

```
ASSIGN @Gpib to 7
PPOLL RESPONSE @Gpib;1
```

The device must have been previously configured for a parallel poll response with the PARALLEL CONFIGURE statement.

# PPOLL UNCONFIGURE Statement

The PPOLL UNCONFIGURE statement disables the parallel poll response of the specified IEEE-488 device or devices. You specify either an I/O Path or a device selector that refers to one or more IEEE-488 devices. If only an interface select code is specified, all devices are deactivated from the parallel poll response. For example:

```
ASSIGN @Dev to 7
PPOLL UNCONFIGURE @Dev
```

If the computer is not the active controller, an error is generated. If a primary device address is specified, the bus action is: ATN, MTA, UNL, LAG, PPC, PPD; otherwise the bus action is: ATN, PPU.

# REMOTE Statement

The REMOTE statement sets the remote state on an IEEE-488 device by asserting the IEEE-488 bus remote line (REN). The device will switch to a remote state only after it has been addressed to listen, causing the front panel to be disabled. You specify either an I/O Path or a device selector that refers to one or more IEEE-488 devices. For example:

```
REMOTE 702
```

If the computer is the active controller and primary addresses are specified, the computer listen addresses the devices to switch them to remote mode. The bus action is:  REN, ATN, MTA, UNL, LAG. The remote line is asserted if the computer is the system controller and ISC select code is specified. If the computer is not the system controller or it is not the active controller, an error is generated.

# REQUEST Statement

The REQUEST statement sends a Service Request (SRQ) on the IEEE-488 bus. You specify either an interface select code or an I/O Path and a service value. To request service, the response value must have bit six set. The SRQ line will remain set until polled by the active controller or another REQUEST statement is executed with bit six clear. For example:

```
REQUEST 7;Bit3+Bit4+Bit6
```

If the computer is the active controller or if the device-selector or the I/O Path specifies address information, an error is generated.

# RESET Statement

The RESET statement resets the IEEE-488 interface. It asserts the IFC line for more than 100 microseconds, clears interrupts, and if the interface is the system controller, sets it to be the active controller. For example:

```
RESET 7
```

# SPOLL Function

The SPOLL function performs a serial poll of an IEEE-488 device and returns the serial poll response, specifying whether the device is requesting service. You specify either an I/O Path or a device selector. The computer must be the active controller and a primary device address must be specified, otherwise an error is generated. One secondary address may also be specified. For example:

```
Stat = SPOLL(712)
```

The IEEE-488 bus action is: ATN, UNL, MLA, TAG, SPE not-ATN, Read data byte, ATN, SPD, UNT.

# TRIGGER Statement

The TRIGGER statement allows the active controller to send a trigger message to a specified IEEE-488 device or to all listen addressed devices on the IEEE-488 bus. You specify either an I/O Path or a device selector that refers to one or more IEEE-488 devices. For example:

```
ASSIGN @Gpib to 705
TRIGGER @Gpib
```

If primary device addresses are specified, the bus action is: ATN, UNL, LAG, GET. If only an interface select code is specified, the bus action is: ATN, GET. If the computer is not the active controller, an error is generated.

# Byte Level Transfer Statements

If you need more control over the bytes transferred over the bus than the high level OUTPUT and ENTER statements allow, you can use the SEND statement as described in the following paragraphs. The OUTPUT and ENTER statements may also be used for byte level transfers under certain circumstances.

Before you can communicate with a device on the IEEE-488 bus, the talker device and the listener device(s) need to be addressed. The high level OUTPUT and ENTER transfer statements generate the necessary device addressing for you. When using the SEND statement, you must generate all the proper bus addressing commands yourself. This requires a more detailed knowledge about IEEE-488 bus operations than is presented here. Please consult any one of the many available books about the IEEE-488 bus or your IEEE-488 bus device manuals.

# SEND Statement

The SEND statement sends byte level IEEE-488 bus data and commands. Commands are sent with the ATN line asserted; whereas data bytes are sent without the ATN line asserted. You specify an I/O Path or an interface select code and a list of messages. The type of message is specified with the keywords CMD, DATA, TALK, LISTEN, SEC, MTA, MLA, UNT, and UNL. For example:

```
SEND @Gpib; UNL MLA TALK Primary CMD 24+128
```

sends the unlisten command, my listen address, the talk address specified by the value of the variable (Primary), and then the command byte 152.

The CMD message evaluates the following expression values and sends them as command bytes. If the CMD keyword is given with no expressions, it asserts the ATN line. For example:

```
SEND 7; CMD 3*5, P, A$, N
```

The DATA message evaluates the following expression values and sends them as data bytes. If the optional END keyword is added, EOI is set on the last data byte. For example:

```
SEND 7; DATA Value*4, ABS(N), Out$ END
```

The LISTEN message sends the expression values as listen address commands. The TALK message sends the expression value as a talk address command. The SEC message sends the expression values as secondary address commands. The MLA message sends the interface's listen address command. The MTA message sends the interface's talk address command. The UNL message sends the unlisten command and the UNT message sends the untalk command.

The computer must be the active controller to use the CMD, TALK, UNT, LISTEN, UNL, SEC, MTA, or MLA messages. Any talk addressed device may send DATA.

The following table lists the bus commands that can be sent with the CMD message.

| Decimal Value | Description |
|---|---|
| 1 | GTL - Go to Local |
| 4 | SDC - Selected Device Clear |
| 5 | PPC - Parallel Poll Configure |
| 8 | GET - Group Execute Trigger |
| 9 | TCT - Take Control |
| 17 | LLO - Local Lockout |
| 20 | DCL - Device Clear |
| 21 | PPU - Parallel Poll Unconfigure |
| 24 | SPE - Serial Poll Enable |
| 25 | SPD - Serial Poll Disable |
| 32-62 | LAG - Listen Address Group |
| 63 | UNL - Unlisten |
| 64-94 | TAG - Talk Address Group |
| 95 | UNT - Untalk |
| 96-111 | PPE - Parallel Poll Enable |
| 112-126 | PPD - Parallel Poll Disable |
| 96-126 | SCG - Secondary Command Group |

Note that the listen and talk address groups (LAG and TAG) consist of 31 different addresses. Each listen and talk address can be further broken down into a secondary address group (SCG). To find the appropriate listen, talk, or secondary address to send for a particular device, use the following equations:

Listen Address = Primary Address + 32
Talk Address = Primary Address + 64
Secondary Address = Primary Address + 96

The examples below show the high level transfer statement OUTPUT followed by four ways to send the exact same information across the IEEE-488 bus with the SEND statement.

```
OUTPUT 705 USING "#,K";"gt;"

SEND 7; CMD "?U%" DATA "gt;" END
SEND 7; CMD 32+31, 64+21, 32+5 DATA "gt;" END
SEND 7; UNL MTA LISTEN 5 DATA "gt;" END
SEND 7; UNL TALK 21 LISTEN 5 DATA "gt;" END
```

# OUTPUT and ENTER Statements

The OUTPUT and ENTER statements can also be used for byte level transfers. If only the interface select code is specified in the OUTPUT and ENTER statements, no bus addressing is performed.

If the device is the active controller then no addressing needs to be done, as long as the addressing has been done once. The talker and listener still need to remain addressed for transfers to take place. For example:

```
10  ENTER 705; Str$
20  FOR I=1 to 10
30    ENTER 7; Str$
40    PRINT Str$
50  NEXT I
60  END
```

This type of addressing will reduce the bus addressing overhead for each piece of data read. However, we do not recommend this practice because branching to an interrupt service routine may destroy the current talker or listener setup.

When the device is not the active controller, it cannot do any bus addressing. The device must make sure that it has been addressed to talk or listen before it participates in the transfer of data. If it has not been addressed, then the device will wait until it is addressed before continuing. For example:

```
OUTPUT 7; Str$  !Waits Until Talker Addressed
ENTER 7; Str$   !Waits Until Listener Addressed
```

A combination of the SEND and the ENTER statements can emulate any of the high level transfer or bus control statements. As an example, let's see how to conduct a serial poll (SPOLL) operation using these statements.

```
Poll_value = SPOLL(705)          !High level command

SEND 7; CMD "?5E" CMD 24         !Output SPE command
ENTER 7 USING "#,B";Poll_value   !Read Poll Value
SEND 7; CMD 25 UNT               !Output SPD command
```

You are limited in your control of the IEEE-488 bus only by your imagination and skill at combining the SEND and ENTER statements.

# Low Level Bus Control Statements

The low level bus control statements allow you to access the status and control registers in both the IEEE-488 driver and in the IEEE-488 controller hardware. The CONTROL and STATUS statements access the IEEE-488 driver registers while the READIO and WRITEIO statements access the controller hardware registers. The definitions of these registers are given near the end of this chapter.

# CONTROL and STATUS Statements

The CONTROL and STATUS statements allow you to configure the serial and parallel poll response bytes, change the primary address, set interrupt mask registers, read the status of the data and bus lines, read the interrupt status, and read the controller status and address. The following program reads the STATUS registers and prints the values out in both decimal and binary.

```
10  STATUS 7;X0,X1,X2,X3,X4,X5,X6,X7 !Read all Status Registers
20  PRINT X0,X1,X2,X3,X4,X5,X6,X67
30  PRINT IVAL$(X0,2),IVAL$(X1,2),IVAL$(X2,2),IVAL$(X3,2)
40  PRINT IVAL$(X4,2),IVAL$(X5,2),IVAL$(X6,2),IVAL(X7,2)
50  END
```

# READIO and WRITEIO Statements

The READIO and WRITEIO statements directly access the IEEE-488 controller hardware registers. Do not attempt to use the READIO and WRITEIO registers unless you are very familiar with the hardware. Use the STATUS and CONTROL registers instead.

Accessing hardware registers can cause your system to crash, data to be lost, or damage to your computer hardware. TransEra cannot be held responsible for any consequences.

# IEEE-488 Interrupts

Interrupts allow the computer to perform other tasks while you wait for some condition to occur. This eliminates the need to continually monitor the STATUS register for some event. HTBasic has the capability of monitoring up to 16 different interrupt conditions at once.

# ON INTR Statement

The ON INTR statement defines an event branch to be taken when an interface card generates an interrupt. You specify the interface select code, an optional priority and the branch type. The branch type may be either a GOTO, GOSUB, CALL, or RECOVER. For example:

```
ON INTR 7,4 GOSUB Repair
```

When an interrupt occurs a DISABLE INTR for the interface is automatically executed. Consequently, an ENABLE INTR statement must be used to explicitly re-enable interrupts.

The default priority is one. The highest priority that can be specified is 15. ON END, ON ERROR, and ON TIMEOUT have a higher priority than ON INTR. When an INTR initiated branch is taken with a GOTO, the system priority is not changed. When an ON INTR branch specifies a CALL or GOSUB, the system priority is changed to the specified priority.

RECOVER causes the program to SUBEXIT from contexts as needed to return to the defining context and resume execution at the specified program line. ON INTR statements that specify CALL or RECOVER will be serviced even if the program context has been changed to another subprogram. ON INTR statements that specify GOTO or GOSUB will be logged and then serviced when control returns to the defining program context.

ON INTR is canceled by OFF INTR, disabled by DISABLE or DISABLE INTR. The following example shows how to detect the IEEE-488 service request (SRQ).

```
10   ON INTR 7 GOSUB 80 !Where to Go When Interrupt Occurs
20   ENABLE INTR 7;2    !Enable SRQ Interrupt
30     . . .
40     . . .
50     . . .
60   STOP
70   !
80   Val = SPOLL(701)   !Clear SRQ Line
90   ENTER 701;Condition!Read Device Condition
100  PRINT Condition
110  ENABLE INTR 7      !Re-Enable SRQ Interrupt
120  RETURN
130  END
```

# OFF INTR Statement

The OFF INTR statement cancels event branches defined by ON INTR. Any INTR events that have been logged but not yet serviced are canceled. An OFF INTR statement without the optional interface select code disables event-initiated branches on all devices. If the interface select code is specified, only that interface interrupt will be disabled. For example,

```
OFF INTR 7
```

cancels event branches for the IEEE-488 interface.

# Enabling and Disabling Interrupts

The DISABLE statement disables all defined event branches except END, ERROR, and TIMEOUT. While disabled, the first event of each type that occurs is logged. When event branching is re-enabled with the ENABLE statement, all logged events are serviced in the order of their event priorities.

The DISABLE INTR statement disables interrupts from just the specified interface. For example,

```
DISABLE INTR 7
```

disables interrupts from the IEEE-488 interface.

The ENABLE INTR statement enables interrupts from a specified interface. An optional bit mask is stored in the interface interrupt-enable register. The default bit mask is the previous bit mask for that interface, or if there is no previous bit mask then a bit mask of all zeros is used. The meaning of the bit mask depends on the interface; consult the interface documentation. For example,

```
ENABLE INTR 7;Bitmask
```

enables interrupts on the IEEE-488 interface and stores the value of the variable Bitmask into the interface interrupt-enable register. The interrupt enable register bits are defined as follows:

# Interrupt Enable Register Bit Mask

| Bit | Value | Meaning |
|---|---|---|
| 15 | -32768 | Active Controller |
| 14 | 16384 | Parallel Poll Config. change |
| 13 | 8192 | My Talk address received |
| 12 | 4096 | My Listen address received |
| 11 | 2048 | EOI received |
| 10 | 1024 | SPAS |
| 9 | 512 | Remote/Local change |
| 8 | 256 | Talker/Listener Address change |
| 7 | 128 | Trigger received |
| 6 | 64 | Handshake Error |
| 5 | 32 | Unrecognized universal command |
| 4 | 16 | Secondary command while addressed |
| 3 | 8 | Clear received |
| 2 | 4 | Unrecognized addressed command |
| 1 | 2 | SRQ received |
| 0 | 1 | IFC received |

The interrupt enable register has the same bit values as STATUS registers 4 and 5. STATUS register 4 tells which condition caused the interrupt. STATUS register 5 tells which interrupts are enabled. To enable more than one interrupt, add up all the event decimal values and use this value as the ENABLE INTR bit mask.

# Handling Service Requests

HTBasic can be programmed to branch to a service routine when a device requests service. The example at the start of this chapter shows how to set up and enable an interrupt for the SRQ line. When a device sets the SRQ line, your service routine needs to perform the following steps: 1) find out which device is requesting service, 2) find out what action needs to be taken, 3) perform the needed action, and 4) re-enable the IEEE-488 interrupts.

Step 1 uses either the PPOLL command or the SPOLL command. With the SPOLL command you have to start with the first address and step through all the addresses, until you find the device requesting service. If you only have one or two devices on the IEEE-488 bus, then this method is quite fast and it eliminates step two.

Step 2 uses a serial poll (SPOLL) to read the device response byte and tells the device that it is being serviced. The device then removes the request by clearing the SRQ line.

Step 3 is dependent on the response byte value. Its interpretation is determined by the device documentation.

Step 4 re-enables the interrupts with the ENABLE INTR command. Interrupts are disabled by the Controller until the current request has been serviced. Once serviced, the interrupts need to be re-enabled.

# Parallel Polling Devices

A parallel poll is the fastest way of determining the requesting device. Each device must first be programmed to respond to a parallel poll request on a unique data line (DIO1 - DIO8). For example,

```
PPOLL CONFIGURE 705;11
```

configures device 5 to respond by placing a 1 on data line DIO4. Bit 3 determines the logic sense of the data line when the device needs service. The data line to use is determined by bits 0-2, offset by one. A value of 3 means use data line DIO4.

To disable a device from responding to a parallel poll use the following commands.

```
PPOLL UNCONFIGURE 705   !Disables Device 5
PPOLL UNCONFIGURE 7     !Disables All Devices
```

To conduct the parallel poll the PPOLL function is used as follows:

```
Pstatus = PPOLL(7)   !Parallel Poll Bus
```

After the parallel poll the variable Pstatus contains the value of the 8 data lines as set by the devices that have been configured to respond to the parallel poll.

# IEEE-488 Registers

STATUS and CONTROL registers for IEEE-488 interfaces are given below. READIO and WRITEIO registers are presented for 9914 and 7210 based IEEE-488 interfaces.

IEEE-488 hardware is not supplied standard with most computers. For HTBasic, TransEra sells the GPIB-900 IEEE-488 Controller. The board fits in any XT or AT bus slot. The board incorporates the NI TMS9914 integrated circuit, the same controller used in HP BASIC workstations. This board provides compatibility with HP BASIC at all levels, including the READIO/WRITEIO level which accesses the 9914 registers directly.

HTBasic also supports most PC IEEE-488 boards from other manufacturers. These boards most often use the NEC PD7210 Chip and consequently are not completely compatible with HP BASIC. STATUS and CONTROL registers for the 9914 and the 7210 are the same, although some bits cannot be supported by the 7210. The READIO/WRITEIO registers of the 7210 are completely different from the 9914 used by HP BASIC. Different tables are given below for the 7210 and the 9914.

READIO/WRITEIO registers allow direct access to the interface hardware. You should not attempt to use these registers unless you are familiar with how the IEEE-488 chip is programmed.

The ON INTR 7 and ENABLE INTR 7 statements are supported. The values for the enable mask in the ENABLE INTR statement are the same as those for STATUS register 5, given below. Some interrupts are not supported by the 7210.

# IEEE-488 CONTROL Registers

The following CONTROL registers are supported.

# CONTROL 0

Reset. The value must be non-zero.

# CONTROL 1

Set Serial Poll Response Byte.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Device Dependent Status |
| 6 | 64 | SRQ 1 - It's me, 0 - It's not me. |
| 5-0 | - | Device Dependent Status |

# CONTROL 2

Set Parallel Poll Response Byte.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | DIO8 |
| 6 | 64 | DIO7 |
| 5 | 32 | DIO6 |
| 4 | 16 | DIO5 |
| 3 | 8 | DIO4 |
| 2 | 4 | DIO3 |
| 1 | 2 | DIO2 |
| 0 | 1 | DIO1 |

# CONTROL 3

Set My Bus Address.

| Bit | Value | Meaning |
|---|---|---|
| 7-5 | - | Unused |
| 4-0 | - | Interface Primary Address |

# CONTROL 4

Release NDAC Holdoff. 0 - No Secondary Address, 1 - Accept Secondary.

# CONTROL 5

Set Parallel Poll Response Mask.

| Bit | Value | Meaning |
|---|---|---|
| 7-5 | - | Unused |
| 4 | 16 | Unconfigure |
| 3 | 8 | Logic Sense |
| 2-0 | - | Bits used for response |

# IEEE-488 STATUS Registers

The following STATUS registers are supported.

# STATUS 0

Return Identification. Always 1.

# STATUS 1

Return Interrupt and DMA status.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Interrupts Enabled |
| 6 | 64 | Interrupt Requested |
| 5-4 | - | Hardware Interrupt Level Switches |
| 3-2 | - | Not used |
| 1 | 2 | DMA channel 1 enabled |
| 0 | 1 | DMA channel 0 enabled |

# STATUS 2

Return Busy Bits.

| Bit | Value | Meaning |
|---|---|---|
| 7-3 | - | Unused |
| 2 | 4 | Handshake in progress |
| 1 | 2 | Interrupts Enabled |
| 0 | 1 | TRANSFER in progress |

# STATUS 3

Return Controller Status and Address.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | System Controller |
| 6 | 64 | Active Controller |
| 5 | 32 | Unused |
| 4-0 | - | Interface Primary Address |

# STATUS 4

Return Interrupt Status. Uses same bit definitions as register 5.

# STATUS 5

Return Interrupt Enable Mask. Use these values with ENABLE INTR to enable interrupts.

| Bit | Value | Meaning |
|---|---|---|
| 15 | -32768 | Active Controller |
| 14 | 16384 | Parallel Poll Config. change |
| 13 | 8192 | My Talk address received |
| 12 | 4096 | My Listen address received |
| 11 | 2048 | EOI received |
| 10 | 1024 | SPAS |
| 9 | 512 | Remote/Local change |
| 8 | 256 | Talker/Listener Address change |
| 7 | 28 | rigger received |
| 6 | 64 | Handshake Error |
| 5 | 32 | Unrecognized universal command |
| 4 | 16 | Secondary command while addressed |
| 3 | 8 | Clear received |
| 2 | 4 | Unrecognized addressed command |
| 1 | 2 | SRQ received |
| 0 | 1 | IFC received* |

*Not supported by the NEC 7210.

# STATUS 6

Return Interface Status. The REM & LOC bits are not always accurate on NEC 7210 cards.

| Bit | Value | Meaning |
|---|---|---|
| 15 | -32768 | REM |
| 14 | 16384 | LLO |
| 13 | 8192 | ATN True |
| 12 | 4096 | LPAS |
| 11 | 2048 | TPAS |
| 10 | 1024 | LADS |
| 9 | 512 | TADS |
| 8 | 256 | LSB of last address |
| 7 | 128 | System Controller |
| 6 | 64 | Active Controller |
| 5 | 32 | Unused |
| 4-0 | - | Primary Interface Address |

# STATUS 7

Return Bus Control and Data Lines.

| Bit | Value | Meaning |
|---|---|---|
| 15 | -32768 | ATN True |
| 14 | 16384 | DAV True* |
| 13 | 8192 | NDAC True* |
| 12 | 4096 | NRFD True* |
| 11 | 2048 | EOI True |
| 10 | 1024 | SRQ True |
| 9 | 512 | IFC True* |
| 8 | 256 | REN True |
| 7 | 128 | DIO8 |
| 6 | 64 | DIO7 |
| 5 | 32 | DIO6 |
| 4 | 16 | DIO5 |
| 3 | 8 | DIO4 |
| 2 | 4 | DIO3 |
| 1 | 2 | DIO2 |
| 0 | 1 | DIO1 |

* Not supported by the NEC 7210.

# 9914 READIO Registers

The following READIO registers are supported.

# 9914 READIO 1

Return Card Identification. Always 1.

# 9914 READIO 3

Return Interrupt and DMA status.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Interrupts Enabled |
| 6 | 64 | Interrupt Requested |
| 5-4 | - | Hardware Interrupt Level Switches |
| 3-2 | - | Not used |
| 1 | 2 | DMA channel 1 enabled |
| 0 | 1 | DMA channel 0 enabled |

# 9914 READIO 5

Return Controller Status and Address.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | System Controller |
| 6 | 64 | Not Active Controller |
| 5 | 32 | Unused |
| 4-0 | - | Interface Primary Address |

# 9914 READIO 17

Return Interrupt Status Register 0.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Interrupt occurred on ISR 0 |
| 6 | 64 | Interrupt occurred on ISR 1 |
| 5 | 32 | Byte Received |
| 4 | 16 | Ready for Next Byte |
| 3 | 8 | EOI detected |
| 2 | 4 | SPAS |
| 1 | 2 | Remote/Local Change |
| 0 | 1 | My Address Change |

# 9914 READIO 19

Return Interrupt Status Register 1.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Trigger Received |
| 6 | 64 | Handshake Error |
| 5 | 32 | Unrecognized Command Group |
| 4 | 16 | Secondary Command While Addressed |
| 3 | 8 | Clear Received |
| 2 | 4 | My Address Received (Listen or Talk) |
| 1 | 2 | SRQ Received |
| 0 | 1 | IFC Received |

# 9914 READIO 21

Return Interface Status.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | REM - Remote State |
| 6 | 64 | LLO - Local Lockout State |
| 5 | 32 | ATN Line True |
| 4 | 16 | LPAS - Listener Primary Addressed State |
| 3 | 8 | TPAS - Talker Primary Addressed State |
| 2 | 4 | LADS - Listener Addressed State |
| 1 | 2 | TADS - Talker Primary Addressed State |
| 0 | 1 | LSB of Last Address |

# 9914 READIO 23

Return Control-Line Status.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | ATN True |
| 6 | 64 | DAV True |
| 5 | 32 | NDAC True |
| 4 | 16 | NRFD True |
| 3 | 8 | EOI True |
| 2 | 4 | SRQ True |
| 1 | 2 | IFC True |
| 0 | 1 | REN True |

# 9914 READIO 29

Return Command Pass Through.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | DIO8 |
| 6 | 64 | DIO7 |
| 5 | 32 | DIO6 |
| 4 | 16 | DIO5 |
| 3 | 8 | DIO4 |
| 2 | 4 | DIO3 |
| 1 | 2 | DIO2 |
| 0 | 1 | DIO1 |

# 9914 READIO 31

Return Bus Data Line Status.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | DIO8 |
| 6 | 64 | DIO7 |
| 5 | 32 | DIO6 |
| 4 | 16 | DIO5 |
| 3 | 8 | DIO4 |
| 2 | 4 | DIO3 |
| 1 | 2 | DIO2 |
| 0 | 1 | DIO1 |

# 9914 WRITEIO Registers

The following WRITEIO registers are supported.

# 9914 WRITEIO 3

Set Interrupt and DMA Enable.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Enable Interrupt |
| 6-2 | - | Unused |
| 1 | 2 | Enable DMA Channel 1 |
| 0 | 1 | Enable DMA Channel 0 |

# 9914 WRITEIO 17

Set Interrupt Mask Register 0.

| Bit | Value | Meaning |
|---|---|---|
| 7-6 | | Unused |
| 5 | 32 | Byte Received |
| 4 | 16 | Ready for Next Byte |
| 3 | 8 | EOI detected |
| 2 | 4 | SPAS |
| 1 | 2 | Remote/Local Change |
| 0 | 1 | My Address Change |

# 9914 WRITEIO 19

Set Interrupt Mask Register 1.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Trigger Received |
| 6 | 64 | Handshake Error |
| 5 | 32 | Unrecognized Command Group |
| 4 | 16 | Secondary Command While Addressed |
| 3 | 8 | Clear Received |
| 2 | 4 | My Address Received (Listen or Talk) |
| 1 | 2 | SRQ Received |
| 0 | 1 | IFC Received |

# 9914 WRITEIO 23

Set Auxiliary Command Register.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | 1 - Set, 0 - Clear |
| 6-5 | - | Unused |
| 4-0 | - | Auxiliary Command |

| Auxiliary Command | | CLEAR | SET |
|---|---|---|---|
| Software Reset | 0 | 128 | |
| Release DAC Holdoff | 1 | 129 | |
| Release RFD Holdoff | 2 | xx | |
| Holdoff on all Data | 3 | 131 | |
| Holdoff on EOI only | 4 | 132 | |
| New Byte Available False | 5 | xx | |
| Force Group Execute Trigger | 6 | 134 | |
| Return to Local | 7 | 135 | |
| Send EOI with Next Byte | 8 | xx | |
| Listen Only | 9 | 137 | |
| Talk Only | 10 | 138 | |
| Goto Standby | 11 | xx | |
| Take Control Asynchronously | 12 | xx | |
| Take Control Synchronously | 13 | xx | |
| Request Parallel Poll | 14 | 142 | |
| Send Interface Clear | 15 | 143 | |
| Send Remote Enable | 16 | 144 | |
| Request Control | 17 | xx | |
| Release Control | 18 | xx | |
| Disable all Interrupts | 19 | 147 | |
| Pass Through Next Secondary | 20 | xx | |
| Short T1 Settling Time | 21 | 149 | |
| Shadow Handshake | 22 | 150 | |
| Very Short T1 Delay | 23 | 151 | |
| Request Service Bit 2 | 24 | 152 | |

# 9914 WRITEIO 25

Set Address Register.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Enable Dual Addressing |
| 6 | 64 | Disable Listener Function |
| 5 | 32 | Disable Talker Function |
| 4-0 | - | Primary Address |

# 9914 WRITEIO 27

Set Serial Poll Response.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Device Dependent Status |
| 6 | 64 | 1 - Send, 0 - Don't Send SRQ |
| 5-0 | - | Device Dependent Status |

# 9914 WRITEIO 29

Set Parallel Poll Response.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | DIO8 |
| 6 | 64 | DIO7 |
| 5 | 32 | DIO6 |
| 4 | 16 | DIO5 |
| 3 | 8 | DIO4 |
| 2 | 4 | DIO3 |
| 1 | 2 | DIO2 |
| 0 | 1 | DIO1 |

# 9914 WRITEIO 31

Set Bus Data Lines Register.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | DIO8 |
| 6 | 64 | DIO7 |
| 5 | 32 | DIO6 |
| 4 | 16 | DIO5 |
| 3 | 8 | DIO4 |
| 2 | 4 | DIO3 |
| 1 | 2 | DIO2 |
| 0 | 1 | DIO1 |

# 7210 READIO Registers

The following READIO registers are supported.

# 7210 READIO 1

Return Card Identification. Always 2.

# 7210 READIO 3

Return Interrupt and DMA status.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Interrupts Enabled |
| 6 | 64 | Interrupt Requested |
| 5-4 | - | Hardware Interrupt Level Switches |
| 3-2 | - | Not used |
| 1 | 2 | DMA channel 1 enabled |
| 0 | 1 | DMA channel 0 enabled |

# 7210 READIO 5

Return Controller Status and Address.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | System Controller |
| 6 | 64 | Active Controller |
| 5 | 32 | Unused |
| 4-0 | - | Interface Primary Address |

# 7210 READIO 18

Return Bus Data Lines.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | DIO8 |
| 6 | 64 | DIO7 |
| 5 | 32 | DIO6 |
| 4 | 16 | DIO5 |
| 3 | 8 | DIO4 |
| 2 | 4 | DIO3 |
| 1 | 2 | DIO2 |
| 0 | 1 | DIO1 |

# 7210 READIO 20

Return Interrupt Status Register 1.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Command Pass Through |
| 6 | 64 | Address Pass Through |
| 5 | 32 | Device Execute Trigger |
| 4 | 16 | End Received |
| 3 | 8 | Device Clear |
| 2 | 4 | Handshake Error |
| 1 | 2 | Data Out - Send Byte |
| 0 | 1 | Data In - Read Byte |

# 7210 READIO 22

Return Interrupt Status Register 2.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Interrupt Occurred on ISR 1 or 2 |
| 6 | 64 | SRQ Received |
| 5 | 32 | Device in Lockout State |
| 4 | 16 | Device in Remote State |
| 3 | 8 | Command Output - Send Byte |
| 2 | 4 | Lockout Change |
| 1 | 2 | Remote Change |
| 0 | 1 | Address Status Change |

# 7210 READIO 24

Return Serial Poll Status.

| Bit | Value | Meaning |
|---|---|---|
| 7-0 | - | Echoes Contents of Serial Poll Mode Reg. |

# 7210 READIO 26

Return Address Status.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Controller in Charge |
| 6 | 64 | ATN Line is High |
| 5 | 32 | Serial Poll Mode State |
| 4 | 16 | Listener Primary Addressed State |
| 3 | 8 | Talker Primary Addressed State |
| 2 | 4 | Listener Addressed |
| 1 | 2 | Talker Addressed |
| 0 | 1 | Talk or Listen Address Received |

# 7210 READIO 28

Return Command Pass Through.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | DIO8 |
| 6 | 64 | DIO7 |
| 5 | 32 | DIO6 |
| 4 | 16 | DIO5 |
| 3 | 8 | DIO4 |
| 2 | 4 | DIO3 |
| 1 | 2 | DIO2 |
| 0 | 1 | DIO1 |

# 7210 READIO 30

Return Address Register 0.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Not Used |
| 6 | 64 | Disable Talker 0. |
| 5 | 32 | Disable Listener 0. |
| 4-0 | - | Interface Major Address |

# 7210 READIO 32

Return Address Register 1.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | EOI Sent on Last Data Byte |
| 6 | 64 | Disable Talker 1. |
| 5 | 32 | Disable Listener 1. |
| 4-0 | - | Interface Minor Address |

# 7210 WRITEIO Registers

The following WRITEIO registers are supported.

# 7210 WRITEIO 3

Set Interrupt and DMA Enable.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Enable Interrupt |
| 6-2 | - | Unused |
| 1 | 2 | Enable DMA Channel 1 |
| 0 | 1 | Enable DMA Channel 0 |

# 7210 WRITEIO 18

Set Bus Command & Data Lines.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | DIO8 |
| 6 | 64 | DIO7 |
| 5 | 32 | DIO6 |
| 4 | 16 | DIO5 |
| 3 | 8 | DIO4 |
| 2 | 4 | DIO3 |
| 1 | 2 | DIO2 |
| 0 | 1 | DIO1 |

# 7210 WRITEIO 20

Set Interrupt Mask Register 1.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Command Pass Through |
| 6 | 64 | Address Pass Through |
| 5 | 32 | Device Execute Trigger |
| 4 | 16 | End Received |
| 3 | 8 | Device Clear |
| 2 | 4 | Handshake Error |
| 1 | 2 | Data Out - Send Byte |
| 0 | 1 | Data In - Read Byte |

# 7210 WRITEIO 22

Set Interrupt Mask Register 2.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Unused |
| 6 | 64 | SRQ Received |
| 5 | 32 | DMA Output - Transfer Data |
| 4 | 16 | DMA Input - Transfer Data |
| 3 | 8 | Command Output - Send Byte |
| 2 | 4 | Lockout Change |
| 1 | 2 | Remote Change |
| 0 | 1 | Address Status Change |

# 7210 WRITEIO 24

Set Serial Poll Mode Response.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Device Dependent Status |
| 6 | 64 | 1 - Send, 0 - Don't Send SRQ |
| 5-0 | - | Device Dependent Status |

# 7210 WRITEIO 26

Set Address Mode Register.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Talk Only Mode |
| 6 | 64 | Listen Only Mode |
| 5 | 32 | Transmit/Receive Mode 1 |
| 4 | 16 | Transmit/Receive Mode 0 |
| 3-2 | - | Unused |
| 1 | 2 | Address Mode 1 |
| 0 | 1 | Address Mode 0 |

# 7210 WRITEIO 28

Set Auxiliary Mode Register. Bits 7 to 5 determine which register is set. Bits 4 to 0 specify the value to write into the register. Tables giving the values for bits 4 to 0 for each auxiliary mode register follow the main table.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | CNT2 - Control Code 2 |
| 6 | 64 | CNT1 - Control Code 1 |
| 5 | 32 | CNT0 - Control Code 0 |
| 4-0 | - | COM4-COM0 - Command Codes 4-0 |

**Auxiliary Command Register.** If CNT2-CNT0 are set to 000 (binary), bits 4 to 0 are defined by the following values:

| Auxiliary Command | SET | CLEAR |
|---|---|---|
| Immediate Execute pon | 0 | xx |
| Chip Reset | 2 | xx |
| Finish Handshake | 3 | xx |
| Trigger | 4 | xx |
| Return to Local | 5 | 13 |
| Send EOI | 6 | xx |
| Non-Valid/Valid 2nd Cmd./Add. | 7 | 15 |
| Parallel Poll Flag | 9 | 1 |
| Goto Standby | 16 | xx |
| Take Control Asynchronously | 17 | xx |
| Take Control Synchronously | 18 | xx |
| Take Control Synch. on End | 26 | xx |
| Listen | 19 | xx |
| Listen in Continuous Mode | 27 | xx |
| Local Unlisten | 28 | xx |
| Execute Parallel Poll | 29 | xx |
| Send Interface Clear | 30 | 22 |
| Send Remote Enable | 31 | 23 |
| Disable System Controller | 20 | xx |

**Internal Counter Register.** If CNT2-CNT0 are set to 001 (binary), bits 4 to 0 specify the State Change Prohibit Times.

**Parallel Poll Register.** If CNT2-CNT0 are set to 011 (binary), bits 4 to 0 have the following meaning:

| Bit | Value | Meaning |
|---|---|---|
| 4 | 16 | Disables Participation in Parallel Poll |
| 3 | 8 | Logic Sense of Status |
| 2-0 | - | Data Line to Assert During Poll |

**Auxiliary Register A.** If CNT2-CNT0 are set to 100 (binary), bits 4 to 0 have the following meaning.

| Bit | Value | Meaning |
|---|---|---|
| 4 | 16 | Select 8-bit EOS length |
| 3 | 8 | Enable Transmit of EOS |
| 2 | 4 | Enable Receive of EOS |
| 1 | 2 | RFD Hold Off on End |
| 0 | 1 | RFD Hold Off on All Data |

**Auxiliary Register B.** If CNT2-CNT0 are set to 101 (binary), bits 4 to 0 have the following meaning.

| Bit | Value | Meaning |
|---|---|---|
| 4 | 16 | Indicates the Value of ist |
| 3 | 8 | Active Level of The INT Pin |
| 2 | 4 | Sets high speed as T(1) |
| 1 | 2 | Enable Transmit of END in Serial Poll |
| 0 | 1 | Enable Setting of CPT bit if Undefined Cmd. |

**Auxiliary Register E.** If CNT2-CNT0 are set to 110 (binary), bits 4 to 0 have the following meaning.

| Bit | Value | Meaning |
|---|---|---|
| 4-2 | - | Unused |
| 1 | 2 | Enable DAC Hold-Off by DCAS State |
| 0 | 1 | Enable DAC Hold-Off by DTAS State |

# 7210 WRITEIO 30

Set Address Register.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Address Register 0 or 1 |
| 6 | 64 | Disable Talk Function |
| 5 | 32 | Disable Listen Function |
| 4-0 | - | Primary Address of 0 or 1 |

# 7210 WRITEIO 32

Set End of String Register.

| Bit | Value | Meaning |
|---|---|---|
| 7-0 | - | EOS Message Byte to Send |

# Statement Bus Action Summary

The following tables show the bus actions that take place when the various IEEE-488 control statements are executed. The table is broken down into System and Non-System Controller, then into Active and Non-Active Controller, and further subdivided into interface select code only or primary address specified. The mnemonics used in the tables are listed below:

ATN - Attention Line
DCL - Device Clear
EOI - End or Identify
GET - Group Execute
GTL - Go to Local
IFC - Interface Clear
LAG - Listen Address Group
LLO - Local Lockout
MLA - My Listen Address
MTA - My Talk Address
PPC - Parallel Poll Configure
PPD - Parallel Poll Disable
PPE - Parallel Poll Enable
PPU - Parallel Poll Unconfigure
Trigger
REN - Remote Enable
SDC - Selected Device Clear
SPD - Serial Poll Disable
SPE - Serial Poll Enable
TAG - Talk Address Group
TCT - Take Control
UNL - Unlisten
UNT - Untalk

**ABORT**

| System Controller | | | | Not System Controller | | | |
|---|---|---|---|---|---|---|---|
| Active Controller | | Not Active Controller | | Active Controller | | Not Active Controller | |
| ISC Only | Primary | ISC Only | Primary | ISC Only | Primary | ISC Only | Primary |
| IFC<br>REN<br>~ATN | Error | IFC<br>REN<br>~ATN | Error | ATN<br>MTA<br>UNL<br>~ATN | Error | No Action | Error |

**CLEAR**

| System Controller | | | | Not System Controller | | | |
|---|---|---|---|---|---|---|---|
| Active Controller | | Not Active Controller | | Active Controller | | Not Active Controller | |
| ISC Only | Primary | ISC Only | Primary | ISC Only | Primary | ISC Only | Primary |
| ATN<br>DCL | ATN<br>MTA<br>UNL<br>LAG<br>SDC | Error | Error | ATN<br>DCL | ATN<br>MTA<br>UNL<br>LAG<br>SDC | Error | Error |

**LOCAL**

| System Controller | | | | Not System Controller | | | |
|---|---|---|---|---|---|---|---|
| Active Controller | | Not Active Controller | | Active Controller | | Not Active Controller | |
| ISC Only | Primary | ISC Only | Primary | ISC Only | Primary | ISC Only | Primary |
| ~REN<br>~ATN | ATN<br>MTA<br>UNL<br>LAG<br>GTL | ~REN | Error | ATN<br>GTL | ATN<br>MTA<br>UNL<br>LAG<br>GTL | Error | Error |

**LOCAL LOCKOUT**

| System Controller | | | | Not System Controller | | | |
|---|---|---|---|---|---|---|---|
| Active Controller | | Not Active Controller | | Active Controller | | Not Active Controller | |
| ISC Only | Primary | ISC Only | Primary | ISC Only | Primary | ISC Only | Primary |
| ATN<br>LLO | Error | Error | Error | ATN<br>LLO | Error | Error | Error |

**PASS CONTROL**

| System Controller | | | | Not System Controller | | | |
|---|---|---|---|---|---|---|---|
| Active Controller | | Not Active Controller | | Active Controller | | Not Active Controller | |
| ISC Only | Primary | ISC Only | Primary | ISC Only | Primary | ISC Only | Primary |
| Error | ATN<br>UNL<br>TAG | Error | Error | Error | ATN<br>UNL<br>TAG | Error | Error |

| | |
|---|---|
| TCT | TCT |
| ~ATN | ~ATN |

**PPOLL**

| System Controller | | | | Not System Controller | | | |
|---|---|---|---|---|---|---|---|
| Active Controller | | Not Active Controller | | Active Controller | | Not Active Controller | |
| ISC Only | Primary | ISC Only | Primary | ISC Only | Primary | ISC Only | Primary |
| ATN<br>EOI<br>read<br>~EOI<br>ATN* | Error | Error | Error | ATN<br>EOI<br>read<br>~EOI<br>ATN* | Error | Error | |

*Restore ATN to previous state

**PPOLL CONFIGURE**

| System Controller | | | | Not System Controller | | | |
|---|---|---|---|---|---|---|---|
| Active Controller | | Not Active Controller | | Active Controller | | Not Active Controller | |
| ISC Only | Primary | ISC Only | Primary | ISC Only | Primary | ISC Only | Primary |
| Error | ATN<br>MTA<br>UNL<br>LAG<br>PPC<br>PPE | Error | Error | Error | ATN<br>MTA<br>UNL<br>LAG<br>PPC<br>PPE | Error | Error |

**PPOLL UNCONFIGURE**

| System Controller | | | | Not System Controller | | | |
|---|---|---|---|---|---|---|---|
| Active Controller | | Not Active Controller | | Active Controller | | Not Active Controller | |
| ISC Only | Primary | ISC Only | Primary | ISC Only | Primary | ISC Only | Primary |
| ATN<br>PPU | ATN<br>MTA<br>UNL<br>LAG<br>PPC<br>PPD | Error | Error | ATN<br>PPU | ATN<br>MTA<br>UNL<br>LAG<br>PPC<br>PPD | Error | Error |

**REMOTE**

| System Controller | | | | Not System Controller | | | |
|---|---|---|---|---|---|---|---|
| Active Controller | | Not Active Controller | | Active Controller | | Not Active Controller | |
| ISC Only | Primary | ISC Only | Primary | ISC Only | Primary | ISC Only | Primary |
| REN<br>~ATN | REN<br>ATN<br>MTA<br>UNL<br>LAG | REN | Error | Error | Error | Error | Error |

**SPOLL**

| System Controller | | | | Not System Controller | | | |
|---|---|---|---|---|---|---|---|
| Active Controller | | Not Active Controller | | Active Controller | | Not Active Controller | |
| ISC Only | Primary | ISC Only | Primary | ISC Only | Primary | ISC Only | Primary |
| Error | ATN<br>UNL<br>MLA<br>TAG<br>SPE<br>~ATN<br>read<br>ATN<br>SPD<br>UNT | Error | Error | Error | ATN<br>UNL<br>MLA<br>TAG<br>SPE<br>~ATN<br>read<br>ATN<br>SPD<br>UNT | Error | Error |

**TRIGGER**

| System Controller | | | | Not System Controller | | | |
|---|---|---|---|---|---|---|---|
| Active Controller | | Not Active Controller | | Active Controller | | Not Active Controller | |
| ISC Only | Primary | ISC Only | Primary | ISC Only | Primary | ISC Only | Primary |
| ATN<br>GET | ATN<br>MTA<br>UNL<br>LAG<br>GET | Error | Error | ATN<br>GET | ATN<br>MTA<br>UNL<br>LAG<br>GET | Error | Error |

# Summary

This chapter discussed the IEEE-488 bus and how it is used to transfer information between devices. It presented the IEEE-488 signal lines and IEEE-488 device addressing. The three levels of IEEE-488 bus data transfer and control statements were also presented along with the HTBasic statements that enable and control IEEE-488 interrupts. The definitions of the registers and a summary of the bus actions that each IEEE-488 statement generates were also given.

# Serial (RS-232) I/O

This chapter describes the particulars of performing I/O on the serial (RS-232) interface. If you have not read the general discussion of I/O presented in the "General Input and Output" chapter, you should do so before reading this chapter. HTBasic requires a driver before the serial interface can be accessed. The *Installing and Using* manual explains how to load the SERIAL driver, how to prevent conflicts with a serial mouse, how to use more than two serial ports, and how to set driver switches.

This chapter describes the handshaking used for ENTER and OUTPUT, and shows how to select between hardware and software handshaking. A long discussion is presented on cabling, communication parameters, and data formats. An in-depth, technical discussion of the RS-232 standard and its omissions is given. Pin assignments for standard connectors are listed. The register definitions for CONTROL, STATUS, READIO, WRITEIO, and the ENABLE INTR are presented.

# General I/O

OUTPUT, ENTER, and TRANSFER are explained in the "General Input and Output" chapter. Before data can be exchanged, the serial driver must be loaded, the computer and device must be cabled together correctly and the RS-232 communications parameters must be set correctly on both the computer and the device.

The CONTROL and STATUS statements read or set various interface parameters, such as baud rate, character format, etc. CONTROL and STATUS examples given in this chapter assume that the serial interface has an ISC of 9. This is the default ISC for the first serial port. If you are using another port, or if you have changed the default ISC, you will use a value other than 9. The following example reads status register 3 of ISC 9.

```
PRINT "COM1 baud rate is ";STATUS(9,3)
```

The READIO and WRITEIO statements read or set hardware registers. These registers are typically quite different than the hardware registers on an HP BASIC RS-232 Interface. You should not attempt to use these registers unless you are familiar with the RS-232C hardware. You should also not mix use of READIO/WRITEIO with STATUS/CONTROL statements. Using STATUS/CONTROL statements is preferred.

# Handshaking

XON/XOFF handshaking capability, not available in HP BASIC, has been added to HTBasic. This capability is turned on by CONTROL 9,100;1 and turned off by CONTROL 9,100;0. It is **on** by default. XON is ^Q (DC1) and XOFF is ^S (DC3).

Hardware handshaking is also supported. It is turned on by

```
CONTROL 9,5;0      ! use DTR and RTS
CONTROL 9,12;0     ! read DSR, CD, and CTS
CONTROL 9,100;0    ! make sure XON/XOFF is disabled
```

It is turned off by

```
CONTROL 9,5;3      ! hold DTR and RTS active
CONTROL 9,12;176   ! ignore DSR, CD, and CTS
CONTROL 9,100;1    ! optionally enable XON/XOFF
```

By default, hardware handshaking is **off**.

# ENTER Serial

To improve performance in receiving data, a 4096 byte receive buffer is available under NT. The receive buffer provided in other versions of Windows is a 1024 byte buffer. When a character is received on the serial interface, it is placed in the receive buffer. This is true regardless of the state of the BASIC program or hardware handshaking signals. The program need not be in an ENTER statement; nor must DSR or CD be active.

**XON/XOFF Handshaking:**
1. When the buffer is about full, XOFF is sent.
2. When there is room again in the buffer, XON is sent.

**Hardware Handshaking:**
1. If the receive buffer is empty, turn on DTR and wait for a character to arrive.
2. Get a character from the receive buffer and turn off DTR.

# OUTPUT Serial

The following outlines the steps used to OUTPUT data for software and hardware handshaking.

**XON/XOFF Handshaking:**
1. If an XOFF has been received from the device,
then the computer waits for an XON to be received.
2. The data is then sent.

**Hardware Handshaking:**
1. DTR and RTS are turned on.
2. The computer waits for the device to turn on DSR and CTS.
3. The data is then sent.
4. DTR and RTS are turned off.

# Interrupt Support

The SERIAL driver supports ON INTR and ENABLE INTR. The definition of bits used in the ENABLE INTR statement is the same as for STATUS register 8, given at the end of this chapter. An example showing use of interrupts with the SERIAL driver is given in the "Interrupt" section of Chapter 5, "General Input and Output."

When an interrupt occurs, the serial interface hardware requires that it be acknowledged. To handle or acknowledge the different interrupts, you should do the following:

| Interrupt | Acknowledge by... |
|---|---|
| Error Occurred | STATUS register 10, UART Line Status |
| Data Available | ENTER or STATUS register 6, Data In |
| Tx Reg Empty | STATUS register 9, Interrupt ID |
| Modem Status | STATUS register 11, Modem Status |

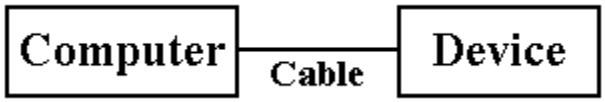
# Connecting Devices to the Serial Interface

Interfacing computers, peripherals, and instruments with a serial interface is often difficult. Once connected and configured properly, things work well, but it is not uncommon to lose a day's labor getting a new connection to work.

The following paragraphs attempt to explain a practical approach to interfacing with an RS-232 interface. An explanation is not made in every case as to why we do the things we do. For those interested, a technical discussion is given after the practical approach has been explained.

In the following discussion, we will use the following communications model:

The Practical Communications Model



# "Standard" Cables

One approach to cabling is to have several types of cables around and when an interfacing situation arises, try them all out and see if one will work. This is not such a bad approach, so we present several common cable diagrams below.

The documentation for the device you wish to communicate with may contain cable diagrams for hooking up the device to common computers. All but the "Straight Through" cable could be considered "Null Modem" cables because they attempt to make another non-modem device look like a modem. A phone call to the device manufacturer can also save a lot of time in determining what cable to use.

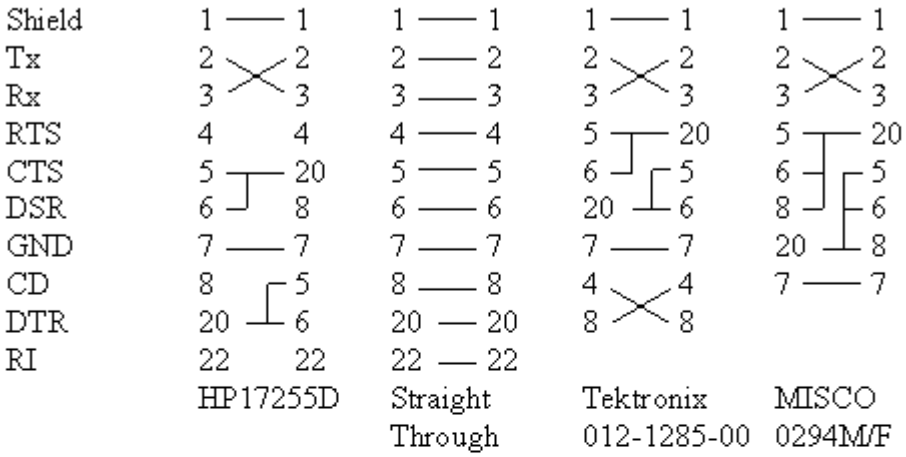
# Cabling From the Ground Up

If you wish to identify a cable which should work in a particular application, it can be done using the straight forward method described here. First, examine the connectors on the computer and the device. The most common connector is a 25 pin DB connector, but you will also commonly see 9 pin DB connectors. Less commonly, you'll see connectors of all shapes and sizes. Connectors almost always have identifying numbers near some of the pins/holes which give the pin numbers. The DB connector pins are arranged in two rows, with the first row having one more pin than the second. The pins are numbered beginning with pin 1 on the wider row. You will need to understand the pin numbering system used in the connectors only if you plan to make your own cable.

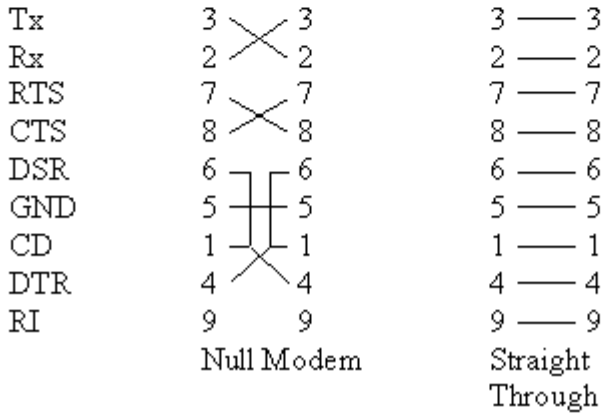
Once you have identified the type of connectors you will need, begin drawing a cable diagram using the steps in the following paragraphs. When your diagram is complete, match it with descriptions of the cables you have available. Then use, buy, or build the correct cable.

Pin 7 is usually the ground pin on a DB-25 pin connector (or pin 5 on a DB-9 connector). This can be verified by turning off and unplugging the computer, disconnecting any cables between it and other devices, and measuring with an ohm meter between the chasis and pin 7. The resistance should be zero. If pin 7 is not the ground, try pin 1, and then the other pins in turn until a ground is found. Do the same for the device and then start building your cable diagram by drawing a connection from the ground pin of the computer to the ground pin of the device.

## Some Common 25-Pin Cables




## Some Common 9-Pin Cables




The computer and the device each talk on one pin of the RS-232 connector and listen on another. Pins 2 and 3 (or pins 3 and 2 on a DB-9) are usually used for talking and listening, respectively; but this is not always the case. Turn the computer on and with a voltmeter, measure the voltage between the chasis of the computer and pin 2. Then do the same for pin 3. Ignore the polarity of the voltage, we are interested in the magnitude only. In the discussion that follows, we will refer to the larger voltage as the "transmitter"

voltage and the smaller voltage as the "receiver" voltage. The computer talks on the pin with the transmitter voltage and listens on the other. Make the same measurement on the device. Then on your diagram, hook the talk pin of the computer to the listen pin of the device and vice versa.

You have now defined a "3-wire cable" and in some instances your cabling task is done. It's probably worth trying, because if you can use a 3-wire cable, you will save yourself a lot of headaches. If data is lost when using a 3-wire cable, try enabling software handshaking (XON/XOFF), explained earlier. With software handshaking turned on, transfer data using the FORMAT ON option of ASSIGN. If software handshaking cannot be used on the device, you will need a hardware handshaking cable.

Let's continue defining a hardware handshaking cable. The three wires now hooked up are sufficient for transferring data. The goal now is to inform the computer and the device when each can and cannot send data to the other. Said another way, we wish to prevent the sending of data while the other is not ready to receive it. We must identify on which pins each presents its "I'm ready" signal(s) and on which pins each asks "Are you ready?"

Get out your voltmeter again and measure pins 5, 6, and 8 (DB-9 pins 8, 6, and 1) on the computer. Remember that we are only interested in the magnitude. Some or all of them should have the "receiver" voltage. The ones that do are asking "Are you ready?" Now measure pins 5, 6, 8, and 20 on the device. One or more of these should have the "transmitter" voltage. If only pin 20 has it, it is the "I'm ready" pin of the device. If more than one of these pins have the "transmitter" voltage, then use 5 if possible as the "I'm ready" pin and any other if 5 is not possible. Draw a connection between all the "Are you ready?" pins of the computer to a single "I'm ready" pin of the device.

All of the pins 5, 6, 8, and 20 on the device that have the "receiver" voltage are the device's "Ready?" pins. Pin 20 (DB-9 pin 4) on the computer is usually the "Ready!" pin of the computer. On your cable diagram, draw a connection between the "Ready!" pin of the computer and the "Ready?" pins of the device.

Your cable diagram is now complete. Match it with descriptions of the cables you have available. Then use, buy, or build the correct cable. If this cable does not work, you should probably call the manufacturer of the device and get their help.

# Communication Parameters

Plugging the cable between the computer and the device is not all that is required to make them talk. The devices must speak the same language in order to understand each other. It doesn't matter what it is, but it is essential they be the same! The language consists of the baud rate, data bits, parity, and stop bits. You must determine which of these parameters can be set on the device, what they can be set to, and how they are set. Typically, you would choose the highest baud rate that both the device and the computer support, one stop bit, eight data bits, and no parity.

To set these values on the computer running HTBasic, use the serial CONTROL registers 3 and 4, which are listed at the end of this chapter. For example, if you are using COM1 and the interface select code has not been changed from 9 and you wish to set 9600 baud, 1 stop bit, 8 data bits, and no parity, you would use this statement:

```
CONTROL 9,3;9600,3
```

Now plug in the cable, set the values, and give it a try. Remember to set the parameters on both the computer and the device. Depending on the device, you may have to set some switches, type a command, or do something else.

# Data Formats

Once the computer and device are communicating correctly, it is still possible to get incorrect data if the data is formatted by the device in one way and interpreted by the computer in another or vice versa. The easiest way to get things working is to instruct the device to send data in ASCII, with CR/LF terminating each data item. However, the fastest way to exchange data is to do so in binary. Read the "General Input and Output" chapter carefully to determine how data is sent and how it is interpreted when it is received. As with communications parameters, it is important to set both the computer and the device to the same data format.

# Interface Status Errors

An interface status error, Error 167, may occur when the cabling, communication parameters, or data formats are not correct. The error is not reported when it occurs but when you access the interface with ENTER or OUTPUT. To discover what the error was, execute the statement:

```
PRINT IVAL$(STATUS(9,10),2)
```

This reads the UART line status register. The meaning of the bits is

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Not used |
| 6 | 64 | Transmit Shift Register Empty |
| 5 | 32 | Transmit Holding Register Empty |
| 4 | 16 | Break Detect |
| 3 | 8 | Framing Error |
| 2 | 4 | Parity Error |
| 1 | 2 | Overrun Error |
| 0 | 1 | Data Ready |

Only bits 1 to 4 represent errors and bit 4 is not necessarily an error, if the device intended to get your attention by sending a BREAK indication. If several of the error bits are set, the meaning of the bits may not reflect the actual problem; you probably have the baud rate or character format set differently on the device and the computer.

If only a single error bit is set, the chances are good that the meaning of that bit reflects the actual error. A **framing** error indicates that the character did not end when it was supposed to. You probably have the number of data or stop bits set differently on the device and computer. A **parity** error means that the expected parity bit of the character was different than expected. You probably have the parity set differently on the device and computer. An **overrun** error means that a character was sent to the computer when the computer was not ready to receive it. You may not have handshaking set up correctly. You may not have executed the proper CONTROL statements, wired the cable correctly, or set up the device to handshake.

Keep in mind that RESET and SCRATCH will reset the communication parameters to the values specified with CONTROL registers 13 and 14. STATUS(9,3) and STATUS(9,4) may be used at any time to check the values of the baud rate and character format to see if they are what you expect.

Spurious interface status errors can be caused by turning the power to the computer or device on or off. To prevent an error on the ENTER or OUTPUT statement, the error can be cleared by STATUS(9,10) or by RESET 9.

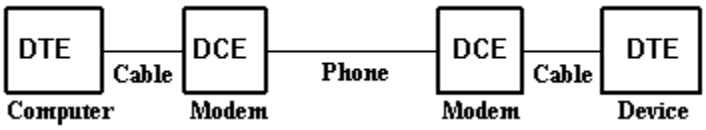
# RS-232: The Standard Non-Standard

The following is an in-depth technical discussion of the RS-232 standard and its weaknesses. You do not need to read it unless you wish a greater understanding of the standard and the issues involved, or if you have a particularly difficult cabling problem that you are attempting to solve.

# The Standard

RS-232 (Electronic Industries Association Recommended Standard #232) describes a method of connecting a computer to an instrument by way of a communication channel, most often a modem attached to a phone line. The RS-232 standard has been revised four times. The current revision is called RS-232-D and was approved on November 12, 1986. The basic communications model is shown in this figure:

The Basic Communications Model




Computers, devices, instruments, terminals, or peripherals are called Data Terminal Equipment (DTE) since they terminate each end of the communications path. Modems are called Data Communications Equipment (DCE) since they facilitate the communication across the communication channel. (In RS-232-D, the term Data Communications Equipment has been replaced with the term Data Circuit-terminating Equipment.) The RS-232 standard describes the interface between the DTE and the DCE. The interface is made using a 25 pin D shell connector. The female connector is associated with the DCE and an interface cable with a male connector is offered by the DTE. The standard optionally allows this cable to be detachable from the DTE, however, no connector type is specified. Only the connection at the DCE is specified by the standard!

# Pin Assignments for PC 25 and 9 pin connectors

| 25 Pin | 9 Pin | Common Mnemonic | Direction | Description |
|---|---|---|---|---|
| 1 | -- | Shield | -- | Shield |
| 2 | 3 | Tx | To | Transmit Data |
| 3 | 2 | Rx | From | Received Data |
| 4 | 7 | RTS | To | Request to Send |
| 5 | 8 | CTS | From | Clear to Send |
| 6 | 6 | DSR | From | DCE Ready |
| 7 | 5 | GND | Both | Signal Ground |
| 8 | 1 | CD | From | Carrier Detect |
| 20 | 4 | DTR | To | DTE Ready |
| 22 | 9 | RI | From | Ring Indicator |

This table shows the pin assignments for the most often used pins. The direction of the signal is given to or from the DCE. The following paragraphs both define the commonly used pins and give a detailed chronological example of how they are used:

Shield - The shield of the cable should be connected to pin 1 on the DTE end only. Connecting it to both can create a ground loop that allows induced noise.

GND - Signal Ground - The signal ground is common to all the other data and control signals, going both to and from the DCE.

RI - Ring Indicator - Regardless of the state of DTR, if a ring is detected on the phone line, this signal should be turned on and off with each ring. This allows the DTE to count the number of rings and decide when to turn DTR on.

DTR - Data Terminal Ready - When this signal is on, it indicates to the modem that it has permission to answer the phone and establish a connection, should it ring, and should maintain the connection until DTR is turned off. When turned off, the modem should hang up and turn DSR off. DTR should not be turned back on until after DSR has been turned off. It is permitted by the standard that DTR be on any time the DTE is ready to send or receive data.

DSR - DCE (Data Set) Ready - When the DCE is ready to operate, and off hook, and the modem is not in voice mode and the modem has finished establishing a phone call, the DCE should turn DSR on. At this point data can be transferred over the modem. If a modem error occurs or a disconnect is detected, DSR should be turned off. The DTE should interpret this as an aborted connection and the next time DSR is turned on, it is considered a new call.

CD - Carrier Detect - When the modem is receiving the proper signals from the remote modem, it turns CD on. When DTR, DSR, and CD are all on, the DTE must be capable of receiving data at any time. When CD is off, no data will be received.

Rx - Received Data - The DTE receives data from the modem on this pin.

RTS - Request to Send - While the DTE must be willing to receive data anytime CD is on, it must ask permission to send data by turning RTS on.

CTS - Clear to Send - When the modem is ready to accept data, with DSR already on, it turns CTS on. If the modem is half-duplex, it also turns CD off to indicate that no data will be received while transmitting.

Tx - Transmitted Data - With DTR, DSR, RTS, and CTS all on, the DTE can transmit data on this line. When it is done transmitting, the DTE turns RTS off and then the modem turns CTS off.

# The Non-Standard

The RS-232 standard has several major omissions. It does not specify the connector that should be used on the DTE if the required cable is detachable, and it does not specify how to connect two DTEs directly.

If the cable between the DTE and the DCE is detachable from the DTE, the standard only specifies the connector for the DCE end of the cable: a male 25 pin D shell connector. The connector for the DTE end of the cable is not specified. The RS-232 port on an IBM AT has a male 9 pin D shell connector. This is not in violation of the standard, so long as a cable is provided that makes a 25 pin D shell male connector available at the DCE end of the cable. The RS-232 port on a Hewlett-Packard 98626 Serial Interface has yet another type of connector.

It is however, a de-facto (i.e., "non-standard") standard to use a 25 pin D shell male connector on the DTE. Thus, the DTE to DCE cable has a female connector on one end, a male connector on the other, and corresponding pins in the connectors are connected "straight through:" pin 2 to pin 2, pin 3 to pin 3, etc.

While the RS-232 standard describes a straight forward method for connecting a DTE to a DCE (modem), it does not describe how to connect one DTE directly to another. Unfortunately, many people would like to connect a computer directly to an instrument — both DTEs. Individual manufacturers have addressed this problem in different ways, and have inadvertently created a lot of confusion. The general approach is to make each DTE think it is communicating with a DCE, thus preserving adherence to the RS-232 standard as much as possible. This can be done in several different ways, none of which are completely compatible with the standard.

The problem is that the standard assumes the communication channel is always slower than the DTE. Thus handshaking is present to prevent the DTE from writing too quickly to the DCE, but the standard contains no means for preventing the DCE from writing too quickly to the DTE. The only signal available for the DTE to use to tell the DCE it is not ready to accept data is DTR. But the standard specifies that "the OFF condition [of DTR] causes the DCE to be removed from the communication channel." Obviously, character handshaking with DTR is going to cause problems if the DCE hangs up between every character. Thus, although DTR is commonly used for handshaking, that use is not completely compatible with the standard.

A less common approach to hooking two DTEs together is to implement a full DCE interface in the instrument, plotter, or printer even though it is a data terminating equipment. Again, this is not completely compatible with the standard. The device must always be connected locally to the controlling computer (the DTE), but a standard DTE to DCE cable can then be used.

# Pin Assignments

The following table gives the complete pin assignments for the RS-232 standard. "Cir" is the official RS-232 name for the circuit. "CCITT" gives the CCITT standard number for the circuit. ("Dir." is the direction of the circuit relative to the DCE.)

| Pin | Circuit | CCITT | Dir. | Description |
|---|---|---|---|---|
| 1 | — | — | — | Shield |
| 2 | BA | 103 | To | Transmitted Data |
| 3 | BB | 104 | From | Received Data |
| 4 | CA | 105 | To | Request to Send |
| 5 | CB | 106 | From | Clear to Send |
| 6 | CC | 107 | From | DCE (Data Set) Ready |
| 7 | AB | 102 | Both | Signal Ground |
| 8 | CF | 109 | From | Recvd Line Signal (Carrier) Detect |
| 9 | — | — | — | Reserved for Testing |
| 10 | — | — | — | Reserved for Testing |
| 11 | — | — | — | Unassigned |
| 12 | SCF* | 122 | From | 2nd Carrier Detect |
| 13 | SCB | 121 | From | 2nd Clear to Send |
| 14 | SBA | 118 | To | 2nd Transmitted Data |
| 15 | DB | 114 | From | Transmitter Timing (DCE) |
| 16 | SBB | 119 | From | 2nd Received Data |
| 17 | DD | 115 | From | Receiver Timing (DCE) |
| 18 | LL | 141 | To | Local Loopback Test |
| 19 | SCA | 120 | To | 2nd Request to Send |
| 20 | CD | 108.2 | To | DTE Ready |
| 21 | RL | 140 | To | Remote Loopback |
| 22 | CE | 125 | From | Ring Indicator |
| 23 | CH* | 111 | To | Rate Select (DTE Source) |
| 24 | DA | 113 | To | Transmit Timing (DTE) |
| 25 | TM | 142 | From | Test Mode |

*CI, Rate Select (DCE Source), is assigned to pin 12 only if SCF is not used. Otherwise it is assigned to pin 23.

# Serial Registers

STATUS and CONTROL registers for the serial interface are given below.

READIO and WRITEIO registers for the PC serial interface are also given below. These registers are different than the hardware registers on an HP BASIC RS-232 Interface. The READIO/WRITEIO registers allow direct access to the interface hardware. You should not attempt to use these registers unless you are familiar with the PC RS-232C hardware. You should not mix use of READIO/WRITEIO with STATUS/CONTROL statements. Using STATUS/CONTROL statements is preferred.

The ON INTR and ENABLE INTR statements are supported by this interface. The values for the enable mask in the ENABLE INTR statement are the same as those for STATUS register 8, given below.

# Serial CONTROL Registers

The following CONTROL registers are supported. When a table is given to explain the meaning of each bit, to calculate the value needed in the CONTROL statement add up the values in the Value column for each of the options needed.

# CONTROL 0

Reset. The value must be non-zero.

# CONTROL 1

BREAK. The value must be non-zero. A 400-millisecond BREAK signal is sent.

# CONTROL 2

This register is undefined in both HTBasic and HP BASIC.

# CONTROL 3

Set baud. The baud rate is set to the value you specify. Available baud rates are 110, 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, and 115200.

# CONTROL 4

Set character format.

| Bits | Value | Meaning |
|---|---|---|
| 7-6 | - | Not used |
| 5,4,3 | 56 | Parity bit is always Zero (111) |
| | 40 | Parity bit is always One (101) |
| | 24 | Parity is Even (011) |
| | 8 | Parity is Odd (001) |
| | 0 | No parity is sent (000) |
| 2 | 4 | 2 stop bits (1.5 for 5 bit characters) |
| | 0 | 1 stop bits |
| 1,0 | 3 | 8 bit (11) |
| | 2 | 7 bit (10) |
| | 1 | 6 bit (01) |
| | 0 | 5 bit (00) |

# CONTROL 5

Set hardware handshaking output line state.

| Bit | Value | Meaning |
|---|---|---|
| 7-5 | - | Not used |
| 4 | 16 | Not used |
| 3 | 8 | Not used (2nd-RTS not supported) |
| 2 | 4 | Not used (DRS not supported) |
| 1 | 2 | 2=Tie RTS high, 0=Use RTS in handshaking |
| 0 | 1 | 1=Tie DTR high, 0=Use DTR in handshaking |

# CONTROL 6

Data Out. The specified character is loaded into the transmit holding register and then transmitted. Handshaking lines are not changed or read. Normally, you should use the OUTPUT statement.

# CONTROL 7

Optional Receiver/Driver Status. On this interface, this is ignored.

# CONTROL 8 to 11

These registers are undefined in both HTBasic and HP BASIC.

# CONTROL 12

Set hardware handshaking input line state.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Not used |
| 6 | 64 | Not used |
| 5 | 32 | 32=Ignore DSR, 0=Use DSR in handshaking |
| 4 | 16 | 16=Ignore CTS, 0=Use CTS in handshaking |
| 3-0 | - | Not used |

# CONTROL 13

Set default baud. Each time HTBasic is started, the default is set to 9600. This register cannot be used to change that. This register can be used to change the default set by SCRATCH or RESET.

# CONTROL 14

Set default character format. Each time HTBasic is started, the default is set to Parity disabled, 1 stop bit, 8 data bits. This register cannot be used to change that. This register can be used to change the default set by SCRATCH or RESET.

# CONTROL 100

XON/XOFF Handshaking. A non-zero value enables XON/XOFF. A zero value disables it. By default it is on.

# Serial STATUS Registers

The following STATUS registers are supported.

# STATUS 0

Card identification. Returns a 66. This is the same value returned by the HP 98644 Serial Interface. It signifies that the following differences from the HP 98626 Interface are present:

1. The optional receiver/driver lines are not present.
Register 7 does nothing.
2. Configuration switches are not present. Defaults are
9600 baud, 8 bit, no parity on a PC.
3. The physical connector is a RS-232-C 9 or 25 pin connector.

# STATUS 1

Interrupt Status. If you will be porting programs to an HP BASIC computer, you should be aware that bits 5 to 0 are defined differently under HP BASIC. Only bits 5 and 4 give the interrupt number, encoded to specify an interrupt in the range 3 to 6. This does not occur with HTBasic.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Interrupts Enabled |
| 6 | 64 | Interrupt waiting service |
| 5-2 | - | Interrupt number, 0-15 |
| 1,0 | - | Not used |

# STATUS 2

Interface Activity Status. Bit 2 is always zero in this implementation because HTBasic stops handshaking whenever a function call may occur.

| Bit | Value | Meaning |
|---|---|---|
| 7-3 | - | Not used |
| 2 | 4 | Handshake in progress (always 0) |
| 1 | 2 | Interrupts Enabled (ENABLE INTR) |
| 0 | 1 | Not used |

# STATUS 3

Baud rate.

# STATUS 4

Character format (See CONTROL above).

# STATUS 5

Read hardware handshaking output line state (See CONTROL above).

# STATUS 6

Data In. Reads next character from the receive buffer. The character is then removed from the buffer. If no characters are in the buffer, the character in the UART receive buffer is returned.

# STATUS 7

Optional Receiver/Driver Status. On this interface, this is always zero.

# STATUS 8

Interrupt Enable Mask. This register is set with the ENABLE INTR statement.

**Note:** It is recommended that bit 1, Interrupt if Tx Holding Reg. Empty, not be used because any time ENABLE INTR is executed, this register will be empty and the interrupt will immediately occur. The interrupt-driven receive buffer code will then immediately acknowledge the interrupt as a side effect of checking for data in the receiver.

| Bit | Value | Meaning |
|---|---|---|
| 7-4 | - | Not used |
| 3 | 8 | Interrupt if Modem Status (register 11) changes |
| 2 | 4 | Interrupt on error (register 10, bits 1 to 4) |
| 1 | 2 | Interrupt if Transmit Holding Reg Empty |
| 0 | 1 | Interrupt if data becomes available |

# STATUS 9

Current Interrupt ID. If bit 0 is 0, then an interrupt is pending, and bits 2 and 1 indicate the cause. The interrupts are prioritized by value. Multiple interrupts can be pending. An interrupt handler should read this register repeatedly, handling each interrupt until this register shows that no interrupt is pending. Also, if a Data Available interrupt is followed by an Error Occurred interrupt before either is serviced, only the later will be reported. This is different than HP BASIC. If you enable both interrupts, and an Error interrupt occurs, you should manually check for data available using bit 0 of register 10.

| Bit | Value | Meaning |
|---|---|---|
| 7-3 | - | Not used |
| 2,1 | 6 | Error Occurred (register 10, bits 1 to 4)(11) |
| | 4 | Data Available (10) |
| | 2 | Transmit Holding Register is empty (01) |
| | 0 | Modem Status (register 11) changed (00) |
| 0 | 1 | 1=No interrupt, 0=Interrupt pending |

To handle, or acknowledge an interrupt, you should do the following:

| Interrupt | Acknowledge by... |
|---|---|
| Error Occurred | STATUS register 10, UART Line Status |
| Data Available | ENTER or STATUS register 6, Data In |
| Tx Reg Empty | STATUS register 9, Interrupt ID |
| Modem Status | STATUS register 11, Modem Status |

# STATUS 10

UART line status.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Not used |
| 6 | 64 | Transmit Shift Register Empty |
| 5 | 32 | Transmit Holding Register Empty |
| 4 | 16 | Break Detect |
| 3 | 8 | Framing Error |
| 2 | 4 | Parity Error |
| 1 | 2 | Overrun Error |
| 0 | 1 | Data Ready |

# STATUS 11

Modem status.

| Bit | Value | Meaning |
|---|---|---|
| 7 | 128 | Carrier Detect (CD) |
| 6 | 64 | Ring Indicator (RI) |
| 5 | 32 | Data Set Ready (DSR) |
| 4 | 16 | Clear to Send (CTS) |
| 3 | 8 | Delta Carrier Detect |
| 2 | 4 | Trailing Edge Ring Indicator |
| 1 | 2 | Delta Data Set Ready |
| 0 | 1 | Delta Clear to Send |

# STATUS 12

Return the hardware handshaking (input lines) state. See CONTROL register 12, above.

# STATUS 13

Return the current default baud rate.

# STATUS 14

Return the current default character format.

# STATUS 100

Return the XON/XOFF enable state. 1 - enabled, 0 - disabled.

# STATUS 101

Return the number of characters in the receive buffer.

# Serial READIO & WRITEIO Registers

The following READIO & WRITEIO Registers are supported. For more details on any of the following registers, please see the 32-bit Windows Programmer's Reference books.

# WRITEIO 1

Baud Rate.

# WRITEIO 2

Not supported.

# WRITEIO 5

Specifies the minimum number of bytes allowed in the input buffer before the XON character is sent.

# WRITEIO 6

Specifies the maximum number of bytes allowed in the input buffer before the XOFF character is sent. The maximum number of bytes allowed is calculated by subtracting this value from the size, in bytes, of the input buffer.

# WRITEIO 7

Specifies the number of bits in the bytes transmitted and received.

# WRITEIO 8

Parity. Specifies the parity scheme to be used.

| Bit | Value | Meaning |
|---|---|---|
| 3 | 8 | Space |
| 2 | 4 | Odd |
| 1 | 2 | Even |
| 0 | 1 | Mark |
| - | 0 | No parity |

# WRITEIO 9

Stop bits. Specifies the number of stop bits to be used.

| Bit | Value | Meaning |
|---|---|---|
| 1 | 2 | 2 Stop Bits |
| 0 | 1 | 1.5 Stop Bits |
| - | 0 | 1 Stop Bit |

# WRITEIO 10

Specifies the value of the XON character for both transmission and reception.

# WRITEIO 11

Specifies the value of the XOFF character for both transmission and reception.

# WRITEIO 12

Specifies the value of the character used to replace bytes received with a parity error.

# WRITEIO 13

Specifies the value of the character used to signal the end of data.

# WRITEIO 14

Specifies the value of the character used to signal an event.

# Serial32 WRITEIO Only Registers

The following WRITEIO registers are supported. For more details on any of the following registers, please see the 32-bit Windows Programmer's Reference books.

# WRITEIO 20

Specifies the code of the extended function to perform.

| Value | Meaning |
|---|---|
| 1 | Set XOFF |
| 2 | Set XON |
| 3 | SET RTS |
| 4 | CLR RTS |
| 5 | SET DTR |
| 6 | CLR DTR |

# Serial32 READIO Only Registers

The following READIO registers are supported. For more details on any of the following registers, please see the 32-bit Windows Programmer's Reference books.

# READIO 30

Specifies the port status.

| Bit | Value | Meaning |
|---|---|---|
| 6 | 64 | TX |
| 5 | 32 | EOF |
| 4 | 16 | XOFF Sent |
| 3 | 8 | XOFF Hold |
| 2 | 4 | RLSD Hold |
| 1 | 2 | DSR Hold |
| 0 | 1 | CTS Hold |

# READIO 31

Specifies the number of bytes received by the serial provider but not yet read by a readfile operation.

# READIO 32

Specifies the number of bytes of user data remaining to be transmitted for all write operations.

# READIO 33

Errors.

| Bit | Value | Meaning | |
|---|---|---|---|
| 15 | 32768 | | Requested mode not supported |
| 8 | 256 | Output buffer full | |
| 4 | 16 | Hardware detected break | |
| 3 | 8 | Hardware detected framing error | |
| 2 | 4 | Hardware detected parity error | |
| 1 | 2 | Character buffer overrun | |
| 0 | 1 | Input buffer overflow | |

# Serial ENABLE INTR Mask

ON INTR and ENABLE INTR are supported on this interface. The definition of bits used in the ENABLE INTR statement is given above under STATUS register 8.

# Summary

This chapter discussed use of the serial (RS-232) interface. Software (XON/XOFF) and hardware handshaking sequences were given. Suggestions for cabling were made as well as instructions for setting communication parameters. An explanation was given of interface status errors. A technical explanation of the RS-232 standard was given. The registers for CONTROL, STATUS, READIO, WRITEIO, and ENABLE INTR were presented.

# Other I/O Destinations/Sources

This chapter discusses I/O (input/output) facilities for  buffers, strings, and a special interface called the "Processor Interface."

Chapter 5, "General Input and Output," discussed the general principles used for input/output. These principles apply to all I/O targets. In particular, use of ASSIGN, OUTPUT, ENTER, STATUS, CONTROL, and TRANSFER were explained. If you have not yet read that chapter, you should do so before reading this one. Chapter 6, "CRT, Keyboard, and Printer" contains information on statements that are especially useful for printer use, and should be read in connection with the Parallel Interface information in this chapter.

# I/O to Strings

The OUTPUT and ENTER statements can be used to write or read data to or from a string. This capability is convenient when you wish to capture the output for further manipulation, or when converting between different formats.

# OUTPUT to Strings

Output to strings starts at the beginning of the string with each OUTPUT statement, outputs the data into the string in FORMAT ON format, and then sets the string length. A second OUTPUT statement will overwrite the information from the first.

```
10 OUTPUT A$;"1"
20 OUTPUT A$;"2"
30 PRINT A$="2"&CHR$(13)&CHR$(10)
40 END
```

This program shows that the second output overwrites the data from the first. It also illustrates that the normal item and line terminators are output to the string unless suppressed.

OUTPUT to a string with the "W" image specifier writes binary data to a string. The data is always written using the native byte ordering of the computer system. Intel processors use LSB FIRST ordering. Motorola 68xxx and HP PA-RISC processors use MSB FIRST ordering.

# ENTER from Strings

ENTER from strings starts at the beginning of the string with each ENTER statement, reads the data from the string with FORMAT ON, and returns an EOI signal with the last character of the string.

```
10 A$="12"&CHR$(10)&"34"
20 ENTER A$;B$
30 ENTER A$;C$
40 PRINT B$,C$
```

In this program, both B$ and C$ will have the value "12", even though the first ENTER did not read all the data from the string. With strings, each ENTER re-starts from the beginning of the string.

ENTER from a string with the "W" image specifier reads binary data from a string. The data is always read using the native byte ordering of the computer system, as explained previously in the "OUTPUT to Strings" section.

# Buffers

I/O to strings, as explained above, has limited application. Buffer I/O is more powerful. HTBasic implements circular buffers. Fill and empty pointers remember where the last OUTPUT, ENTER, or TRANSFER ended, allowing the next statement to pick up where the last one left off. The ASSIGN statement sets up a buffer for I/O. While buffers can be created in strings or arrays (named buffers), unnamed buffers are recommended. The statement

```
ASSIGN @Iopath TO BUFFER [300]
```

creates an unnamed buffer and assigns it an I/O path name. The

```
ASSIGN @Another TO BUFFER X(*)
```

statement assigns an I/O path name to a variable previously declared as a buffer in a COM, DIM, INTEGER, LONG, STATIC or REAL statement.

The buffer specified in ASSIGN may now be used in ENTER, OUTPUT, and TRANSFER statements. Information kept about a BUFFER includes the current number of bytes in the buffer (initially set to 0), the empty and the fill pointers (initially set to 1), the buffer capacity, and TRANSFER information.

If somehow a BUFFER ceased to exist while the I/O path used to write to it still existed, fatal errors could result. For this reason, the BUFFER lifetime must equal or exceed the I/O path lifetime. The following table shows the legal and illegal combinations of BUFFERs and I/O paths. A value of 0 means the combination is legal. A non-zero value gives the error returned if this combination is used.

| **Type of BUFFER** | **Type of I/O Path** | | | |
|---|---|---|---|---|
| | **Local** | | **COM** | **Parameter** |
| Local | 0 | 602 | 602 | |
| Same COM | - | 0 | - | |
| Different COM | - | 602 | - | |
| Parameter | 0 | 602 | 0* | |
| Un-named | 0 | 602 | 602 | |
| ALLOCATE | 603 | 603 | 603 | |
| Not a BUFFER | 603 | 603 | 603 | |

*If the I/O path and BUFFER parameters have been passed through multiple CALL levels, the BUFFER must outlive the I/O path. Also, if a parameter originated as a COM variable, the rules for COM variables apply.

Unnamed buffers can only be accessed through their I/O path. Named buffers can be directly accessed through their variable's name, but this procedure is not recommended since the data in the buffer is unformatted, the data may have the wrong byte order, and direct access does not automatically update the buffer registers. The data in the buffer and the string's current length can be changed, but the buffer registers (empty and fill pointers, current-number-of-bytes register) are not automatically updated. To automatically update the buffer registers use ENTER, OUTPUT, and TRANSFER statements.

# BUFFER STATUS/CONTROL Registers

The STATUS and CONTROL registers of an I/O path assigned to a BUFFER were presented in full in Chapter 5. The following example shows use of registers 2 through 5. All of these registers can be read, and all but the first can be set.

```
10   ASSIGN @Io TO BUFFER [1000];FORMAT OFF
20   Info("Start",@Io)
30   OUTPUT @Io;PI,1/3
40   Info("After OUTPUT",@Io)
50   ENTER @Io;X
60   Info("After ENTER",@Io)
70   END
80   SUB Info(When$,@Io)
90     PRINT When$
100    PRINT "  Total buffer size is    ";STATUS(@Io,2)
110    PRINT "  Fill pointer is         ";STATUS(@Io,3)
120    PRINT "  # of bytes in buffer is ";STATUS(@Io,4)
130    PRINT "  Empty pointer is        ";STATUS(@Io,5)
140  SUBEND
```

This program produces the following output:

```
Start
 Total buffer size is     1000
 Fill pointer is          1
 # of bytes in buffer is  0
 Empty pointer is         1
After OUTPUT
 Total buffer size is     1000
 Fill pointer is          17
 # of bytes in buffer is  16
 Empty pointer is         1
After ENTER
 Total buffer size is     1000
 Fill pointer is          17
 # of bytes in buffer is  8
 Empty pointer is         9
```

# The Processor Interface (32)

Interface select code 32 has a special usage and cannot be changed. It allows some system attributes associated with the computer processor to be set or read. CONTROL and STATUS are the only operations allowed on this interface. To ease porting of HP BASIC programs, CONTROL operations are supported, but are ignored except as noted. The following STATUS registers are supported in this version.

# STATUS 0

Parity Checking. Always 1 (enabled), regardless of the presence or absence of and state of parity checking.

# STATUS 1

External Cache. Always 0 (disabled), regardless of the presence or absence and state of an external cache.

# STATUS 2

Floating-Point Math Hardware. If a math coprocessor is present, and enabled, a one is returned. If no math coprocessor is present, or if one is present and disabled, then zero is returned.

# STATUS 3

Internal (Inside the Processor) Cache. Always 0 (disabled).

# STATUS 4

Battery-Backed-Up Clock Type. Presently, always 1.

0 = Battery-backed-up clock is NOT present.
1 = Battery-backed-up clock is present.

# STATUS 5

Reports faster timing resolution.

1 = indicates the faster timing resolution is in effect.

0 = indicates the faster timing resolution is not effect.

# Accessing Other Interfaces and Devices

If available, an HTBasic device driver is the best way to access a device, interface, or plug-in board. An HTBasic device driver is loaded only while HTBasic is running and so does not consume any memory when not needed. An HTBasic driver also has STATUS and CONTROL registers that facilitate the set up and use of the board.

# Summary

This chapter discussed I/O to buffers, strings and the processor interface (interface 32).

# DLL Toolkit

The Dynamic Link Library (DLL) Toolkit is provided to permit the use of exterior DLLs in HTBasic programs.

# DLL GET

The first command keyword is DLL GET, which sets up a Dynamic Link Library (DLL) function to use in the program. The syntax for this command is:

```
DLL GET "returntype dllname:functionname" AS "alias"
```

where:

***returntype*** is one of the following: VOID, SHORT, LONG, DOUBLE, CHAR, CHARPTR, VARIABLE.

***dllname*** must be the name of a loaded DLL.

***functionname*** is the name of the function in the DLL you wish to call, or a variable exported from the DLL. All Function/Variable names must use valid HTBasic function name conventions or an alias using HTBasic function name conventions must be provided. The DLL loader will allow you to load two functions with the same name as long as they are in different DLL's. However, without an alias specified, there is no way to differentiate which DLL you are trying to call and the DLL loader will always call the first function by that name. You cannot have an HTBasic function with the same name as a DLL function.

***alias*** is an optional function/variable name to use within HTBasic.

Examples of this command include:

```
DLL GET "VOID Pipecalc:Xsection" AS "Cross"
DLL GET "SHORT Pipecalc:Xsection" AS "Cross"
DLL GET "LONG Pipecalc:Xsection" AS "Cross"
DLL GET "DOUBLE Pipecalc:Xsection" AS "Cross"
DLL GET "CHAR Pipecalc:Xsection" AS "Cross"
DLL GET "CHARPTR Pipecalc:Xsection" AS "Cross"
DLL GET "VARIABLE Pipecalc:Xsection" AS "Cross"
```

Both _cdecl type DLLs and _stdCall DLLs may be called with the DLL Toolkit. They syntax is:

```
DLL GET "STDCALL VOID DLL:FUNCTION" AS "ALIAS"
DLL GET "CDECL VOID DLL:FUNCTION" AS "ALIAS"
DLL GET "VOID DLL:FUNCTION" AS "ALIAS" ! defaults to cdecl
```

# DLL LOAD

The DLL LOAD command keyword specifies the Dynamic Link Library (DLL) to LOAD into the program. The keyword syntax is:

```
DLL LOAD "dllname"
```

where:

***dllname*** must be the name of a DLL to load.

A couple of examples of the command keyword in use would be:

```
DLL LOAD "Pipecalc"
DLL LOAD "Flowtrak"
```

# DLL READ

The DLL READ command retrieves a Dynamic Link Library (DLL) variable value to use in the HTBasic program and reads it into a BASIC variable. Its syntax is:

```
DLL READ "varname";basic variable
```

where:

***varname*** is any variable name within the DLL.

***basic variable*** is any legal variable name to use within HTBasic.

Samples of this command could include:

```
DLL READ "Xsection";Crosec
DLL READ "Flowrate";Torrant
```

# DLL UNLOAD

The DLL UNLOAD command specifies the Dynamic Link Library (DLL) to UNLOAD from the HTBasic program. You may specify a specific DLL to remove or simply remove all of them by using the syntax:

```
DLL UNLOAD "dllname"  or ALL
```

where:

***dllname*** must be the name of a DLL to load.

Samples of this command could include:

```
DLL UNLOAD ALL    !Removes all loaded DLLs
DLL UNLOAD "Flowtrak"    !Removes only Flowtrack.dll
```

# DLL WRITE

The DLL WRITE command sets a Dynamic Link Library (DLL) variable to use in the HTBasic program and writes the value of a BASIC variable into a DLL variable. Its syntax is:

```
DLL WRITE "varname";value
```

where:

***varname*** is any variable name within the DLL.

***value*** is any numeric value.

Samples of this command in operation could include:

```
DLL WRITE "Xsection";3.559
DLL WRITE "Flowrate";20.9
```

# LIST DLL

The LIST DLL command lists the name of each Dynamic Link Library (DLL) function and variable currently in memory. The syntax of the LIST DLL command is simply:

```
LIST DLL
```

Samples of the LIST DLL command could include:

```
LIST DLL    !lists the DLLs to the CRT
LIST DLL #PRT    !lists the DLLs to a printer
```

# Export.h

**Export.h** is a C++ file header for C++ functions and variables exported from HTBasic. The user can use these functions from a DLL.

**NOTE**: This is a C++ file. Knowledge of C++ will be helpful in understanding and making use of these functions.

The exported  C++ variables include:

# g_hBasicCursor

This is the handle to an alternate cursor for the Basic Window. To use an alternate cursor, set this handle to the desired cursor or call the SetBasicCursor function.

# g_hBasicWindow

This is the handle to the main HTBasic window.

# Exported Functions

The exported C++ functions include:

# Disp

Prints String directly on HTBasic's display line. The C++ function declaration (or prototype) is:

```
void Disp(char * Msg,BOOL opt=1);

The example of the command in operation is:

Disp("This is the HTBasic DISP line",(1))
Disp("This is the HTBasic message line",(0))
```

Where a non zero value sends output to the DISP line. A zero value sneds the output to the message line.

# Signal

Sets an HTBasic signal (0-15). The C++ function declaration is:

```
int Signal(int num)
```

An example of the function is:

```
int success = Signal(15);
```

# CheckInt

Check to see if a clear I/O or Basic Reset occurred. The C++ function declaration is:

```
int CheckInt()
```

The sample of the function is:

```
if (CheckInt())
{    // an interrupt occurred
}
```

**Note**: This does not work reliably on spawned threads.

# SetBasicCursor

Sets one of the standard Windows cursors for ht Htbasic Window to use. The C++ function declaration is:

Void SetBasicCursor(char * cursor);

A sample of the command is:

SetBasicCursor("hourglass");

Valid parameter strings are: arrow, crosshair, help, hourglass, ibeam, no, sizeall, size nesw, size ns, size nwse, size we, up arrow, wait arrow. Passing any other sting can and will refresh the cursor.

# PutBuffer

Writes string information into an HTBasic Buffer. The C++ function declaration is:

```
int PutBuffer(void * IOPath,char* data,short option)
```

You must have a pointer to a buffer that you have already passed into the DLL. The option parameter if true will cause a CR/LF to be added to the data. An example of this command would be:

```
PutBuffer(buffer,"hello",1)
```

# GetBuffer

Reads string information out of an HTBasic Buffer. The C++ function declaration is:

```
int GetBuffer(void * IOPath,char * data, long len)
```

A sample of the command is:

```
char data[20];
GetBuffer(buffer,data,15)          // read 15 bytes from buffer
```

# Interactive

See HTBasic Suspend Interactive and resume interactive command. The C++ function declaration is:

```
void Interactive(short option)
```

If option is false then interactive is suspended.  If option is true interaction is resumed. A sample of the command is:

```
Interactive(0);
```

# GetBasicEvents

Requests handles to events that can be monitored for reset and shutdown. The C++ function declaration is:

```
void GetBasicEvents(LPHANDLE ResetEvent, LPHANDLE ShutdownEvent);
```

ResetEvent and ShutdownEvent are Event handles that will be signaled if the corresponding event happens on the HTBasic Thread. A sample of the function is:

```
HANDLE hResetEvent,hShutdownEvent;
GetBasicEvents(&ResetEvent,&ShutdownEvent);
```

# Registerthread

Hands HTBasic an event handle that it will wait on before shutting down. The C++ function declaration is:

```
void Registerthread(LPHANDLE phEvent);
```

where, **phEvent** is an Event handle that is created in the DLL and handed to HTBasic. A sample of this is:

```
HANDLE hEvent;
RegisterThread(&hEvent);
```

# Unregisterthread

Tells HTBasic that it no longer needs to wait on the previously registered event. The C++ function declaration is:

```
void Unregisterthread(LPHANDLE phEvent);
```

A sample of the command is:

```
HANDLE hEvent;
Unregistered(&hEvent);
```

where, **phEvent** is an Event handle that is created in the DLL and previously used to call RegisterThread.

# Summary

The DLL Toolkit commands put powerful tools into the programmer's hands to better utilize HTBasic in many new applications or improve its usefulness in existing applications.

# International Language Support

This chapter describes HTBasic international language support. HP BASIC is fairly tightly tied to the Roman-8 character set. HTBasic is not tied to any given character set. HTBasic depends on the operating system for keyboard, display, and printer support of different character sets. HTBasic provides support for collation or lexical ordering and upper- and lowercase conversions. Users are encouraged to use the character sets supported by the operating system. The use of Roman-8 is discouraged when it is not supported by the operating system.

This chapter describes handling attribute/color character conflicts, LEXICAL ORDER (collating sequence), upper/lowercase conversions, LABEL characters, and user-defined lexical order rules. Limited Roman-8 character set support is explained. Lexical order and character set tables are given at the end of the chapter.

# Character Sets

Character sets are supported through printer and screen fonts and keyboard input methods. The correct combinations of fonts and input methods are usually installed as part of the system installation.

ASCII is one of the most widely used character sets, but unfortunately, defines characters only up to CHR$(127) and excludes many characters necessary in languages throughout the world. Other character sets define characters up to CHR$(255) and include other necessary characters. The following paragraphs describe several of these character sets. HTBasic has built-in support for one character set, but the capability is present for users to add support for other character sets.

Japanese-enabled versions of HTBasic, when run in Japanese mode, allow use of two-byte characters using the ISO-932 and Shift-JIS character sets. These are explained later in this chapter.

Character set tables for code pages 850, 437, Roman-8, Latin-1, ISO-932 and an overview of the Shift-JIS character set are given at the end of this chapter.

# Latin-1

The character set used by Microsoft Windows is called Latin-1 or ISO 8859-1.

HTBasic for Windows expects the Latin-1 character set to be in use. If it is not, the keyboard may not match the display and upper- and lowercase conversions and collating may not function as expected.

# ISO-932 and Shift-JIS

All Japanese-enabled versions of HTBasic use the ISO-932 single-byte and Shift-JIS character double-byte sets for character representation when run in Japanese mode. These are the same character sets used by HP BASIC. The ISO-932 character set is the same as the ASCII set for characters whose values are less than 128. Characters whose values are between CHR$(129) and CHR$(159) or between CHR$(224) and CHR$(252) are used as leading bytes for two-byte characters using the Shift-JIS character mapping. Characters whose values are between CHR$(161) and CHR$(223) are half-width katakana characters from the ISO-932 character set. CHR$(160) is a space. CHR$(128) and CHR$(253) - CHR$(255) are undefined.

Non-ASCII characters are entered into HTBasic programs using the operating system's *Input method*. ASCII, katakana, and hiragana characters are entered using the normal keyboard keys together with special shift keys. Kanji are usually entered by allowing the user to type a phonetic (ASCII, katakana, or hiragana) representation of the desired character on the keyboard and pressing a convert key, which displays a list of possible characters at the bottom of the screen or in a separate window. The user then chooses the desired character from the list.

# Variable Names

HP BASIC limits the international characters in variable names to CHR$(161) to CHR$(254). In HTBasic, this range is expanded to CHR$(128) to CHR$(254) since many commonly used characters are in the range excluded by HP BASIC. In Japanese mode, HTBasic allows characters in the range CHR$(161) - CHR$(223) (single-width katakana characters) in variable names.

# Attribute Character Conflict

HP BASIC uses the range CHR$(128) to CHR$(143) for attribute and color control characters. This range is used by some character sets for various international characters. To allow use of characters in this range, HTBasic will move attribute and color control characters from this range to CHR$(16) to CHR$(31) with the statement:.

```
CONTROL CRT,100;1
```

To restore the normal range, use

```
CONTROL CRT,100;0
```

When HTBasic is run in Japanese mode, CONTROL CRT,100;1 is executed automatically.

CONTROL CRT,100 does not affect values used with CONTROL registers, only values PRINTed or OUTPUT to the CRT. This statement is an enhancement to HTBASIC and will return an error if executed on a Series 200/300 computer.

The following table shows the attribute and color control characters for both the normal and alternate ranges. Remember that not all attributes are supported on every display.

| Attribute | Normal | Alternate |
|---|---|---|
| None | 128 | 16 |
| Inverse | 129 | 17 |
| Blinking | 130 | 18 |
| Inverse & Blinking | 131 | 19 |
| Underline | 132 | 20 |
| Underline & Inverse | 133 | 21 |
| Underline & Blinking | 134 | 22 |
| Underline, Inverse, & Blinking | 135 | 23 |

| Attribute | Normal | Alternate |
|---|---|---|
| White | 136 | 24 |
| Red | 137 | 25 |
| Yellow | 138 | 26 |
| Green | 139 | 27 |
| Cyan | 140 | 28 |
| Blue | 141 | 29 |
| Magenta | 142 | 30 |
| Black | 143 | 31 |

# Lexical Order

"Lexical order" is another term for "alphabetical order". A "lexical order" defines an ordering of each character in a character set. By assigning an order number to each character, strings can be compared in a meaningful way with "<", ">", and MAT SORT. Different languages have different lexical orders.

The statement LEXICAL ORDER IS can be used to specify lexical order rules. The current LEXICAL ORDER is returned by the SYSTEM$("LEXICAL ORDER IS") function.

Rules for five languages are built into HTBasic: ASCII, FRENCH, GERMAN, SPANISH, and SWEDISH. (In HTBasic, LEXICAL ORDER IS STANDARD is equivalent to LEXICAL ORDER IS ASCII). These languages are inclusive enough to support most ordering conventions. If the language you are using is not listed, check the LEXICAL ORDER tables near the end of this chapter to see which most nearly matches your language. You may define your own ordering rules as explained later in this chapter.

In Japanese mode, HTBasic defaults to LEXICAL ORDER IS STANDARD.

You must have the correct character set active for the built-in rules to function correctly. Limited support for Roman-8 on operating systems that don't support it is explained later in this chapter.

Execute one of the following statements to specify lexical ordering rules:

```
LEXICAL ORDER IS ASCII
LEXICAL ORDER IS FRENCH
LEXICAL ORDER IS GERMAN
LEXICAL ORDER IS SPANISH
LEXICAL ORDER IS SWEDISH
```

# Upper and Lowercase Conversions

The LEXICAL ORDER IS statement also determines upper/lowercase conversions in addition to ordering rules. Rules for the built-in languages are given in the table below. Note that Ÿ (uppercase Y umlaut) does not exist in codepage 437, 850 or the Latin-1 character sets. In these cases, Y is used for UPC$("ÿ").

You may define rules for other languages using "LEXICAL ORDER IS *Array(*)*", which has been enhanced to allow case conversion rules to be stored in the array along with order rules. HP BASIC does not support these enhancements but does not return an error if they are present. These enhancements are explained later under "User-Defined UPC$/LWC$ Rules".

In Japanese mode, uppercase and lowercase conversion is limited to ASCII characters; Japanese characters are not converted.

```
Uppercase Table
          a-z àáâãäåæçèéêëìíîïðñòóôõöøùúûüýÿþ
-----------------------------------------------
ASCII   : A-Z ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝŸÞ
FRENCH  : A-Z AAAÃAAÆCEEEEIIIIÐÑOOOÕOØUUUUYYÞ
GERMAN  : A-Z ÀÁÂÃÄÅÆCÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝYÞ
SPANISH: A-Z AAAÃÄÅÆCEEEEIIIIÐÑOOOÕÖØUUUÜYYÞ
SWEDISH: A-Z ÀÁÂÃÄÅÆCÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝYÞ

Lowercase Table
          A-Z ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝŸÞ
-----------------------------------------------
ASCII   : a-z àáâãäåæçèéêëìíîïðñòóôõöøùúûüýÿþ
FRENCH  : a-z àáâãäåæçèéêëìíîïðñoóôõöøùúûüýÿþ
GERMAN  : a-z àáâãäåæçèéêëìíîïðñoóôõöøùúûüýÿþ
SPANISH: a-z àáâãäåæçèéêëìíîïðñoóôõöøùúûüýÿþ
SWEDISH: a-z àáâãäåæçèéêëìíîïðñòóôõöøùúûüýÿþ
```

# Japanese Character Conversions

In Japanese mode, HTBasic supports converting between hirigana and double-width katakana and between single- and double-width katakana and single- and double-width Roman characters. This support is accessed through the CVT$ command.

# LABEL Character Set

One limitation of most operating system character set support is that it does not contain vector definitions of the characters for the LABEL statement. Also, like HP BASIC, the HTBasic LABEL statement does not support all the international language characters above CHR$(127). But unlike HP BASIC, HTBasic has been enhanced to allow the user to define his own characters, or delete existing characters. The characters that are defined by default are:

| Char | Latin-1 | PC-850 | Roman-8 |
|---|---|---|---|
| ü | 252 | 129 | 207 |
| é | 233 | 130 | 197 |
| â | 226 | 131 | 192 |
| ä | 228 | 132 | 204 |
| à | 224 | 133 | 200 |
| å | 229 | 134 | 212 |
| ç | 231 | 135 | 181 |
| ê | 234 | 136 | 193 |
| ë | 235 | 137 | 205 |
| è | 232 | 138 | 201 |
| ï | 239 | 139 | 221 |
| î | 238 | 140 | 209 |
| ì | 236 | 141 | 217 |
| Ä | 196 | 142 | 216 |
| Å | 197 | 143 | 208 |
| É | 201 | 144 | 220 |
| æ | 230 | 145 | 215 |
| Æ | 198 | 146 | 211 |
| ô | 244 | 147 | 194 |
| ö | 246 | 148 | 206 |
| ò | 242 | 149 | 202 |
| û | 251 | 150 | 195 |
| ù | 249 | 151 | 203 |
| Ö | 214 | 153 | 218 |
| Ü | 220 | 154 | 219 |
| ø | 248 | 155 | 214 |
| £ | 163 | 156 | 187 |
| Ø | 216 | 157 | 210 |
| á | 225 | 160 | 196 |
| í | 237 | 161 | 213 |
| ó | 243 | 162 | 198 |
| ú | 250 | 163 | 199 |
| ñ | 241 | 164 | 183 |
| Ñ | 209 | 165 | 182 |
| ¿ | 191 | 168 | 185 |
| ¡ | 161 | 173 | 184 |
| ¤ | 164 | 207 | 186 |
| ß | 223 | 225 | 222 |
| ` | 175 | 238 | 176 |
| ¢ | 180 | 239 | 168 |
| § | 167 | 245 | 189 |
| ° | 176 | 248 | 179 |
| Umlaut | 168 | 249 | 171 |

When run in Japanese mode, HTBasic allows the user to load a Japanese character set for use with the LABEL command. This is not done by default because of its memory requirements; the Japanese character set contains several thousand characters.

# Defining Your Own LABEL Characters

To define your own characters, use one of these two syntaxes:

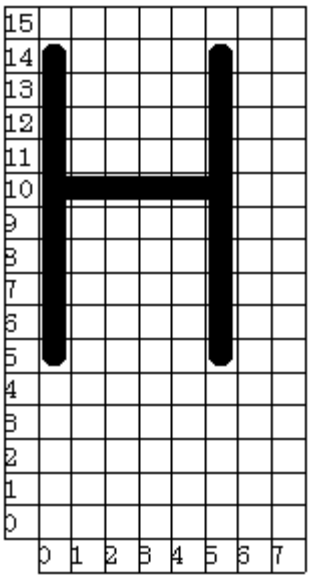
```
CONFIGURE LABEL First_char TO String$
CONFIGURE LABEL First_char TO Array$(*)
```

where *First_char* is a numeric expression, rounded to an integer, which gives the LABEL character to be defined, and *String$* is a string expression that contains the new definition. If a string array is specified, then one definition is stored in each element and additional characters following *First_char* are also defined. Characters in the range 33 to 255 may be defined.

HTBasic provides 8 kilobytes of definition space. If you run out of space, you can free up space by deleting unused definitions. To delete the definition of a character, specify a zero length string for the definition.

The LABEL font is defined in a character cell that is 8 units wide and 16 units high. The x units are numbered 0 to 7; the y units are numbered 0 to 15. The baseline is y=5. The normal descender, such as that for the lower-case "g", goes down to y=1. Characters are left justified in the character cell. The top of an H is at y=14. The right side of the H is at x=6.

Each character in the definition gives an x,y coordinate and a flag indicating whether to move or draw to that coordinate. The flag is stored in bit 7 of the character. If set, MOVE to x,y; otherwise DRAW to x,y. The x coordinate is stored in bits 6, 5, and 4. The y coordinate is stored in bits 3, 2, 1, and 0.




The following example shows the definition of the character "H":

```
Val = Move+ x*16 + y
133 = 128 + 0*16 + 5
14  =       0*16 + 14
238 = 128 + 6*16 + 14
101 =       6*16 + 5
138 = 128 + 0*16 + 10
106 =       6*16 + 10

CONFIGURE LABEL 72 TO CHR$(133)&CHR$(14)&CHR$(238)&
 CHR$(101)&CHR$(138)&CHR$(106)
```

# Using LABELCHR.BAS

An example program, LABELCHR.BAS, is distributed with HTBasic that can be used to examine the definitions of characters, or develop new definitions. LABELCHR.BAS will not fully automate the task of adding characters, but is presented as an aid in developing character definitions.

When you run LABELCHR.BAS the screen will be painted with four grids. Grid #1 will be blank and is the current grid. Grid #2 displays an ASCII table of existing LABEL definitions for characters in the range 128 to 255. (The grid is actually too small to enclose all the characters. It is normal for the last row and last column to be displayed outside of the grid.) Grid #3 displays the letter "g", a good example of where a descender is located in the character cell. Grid #4 displays the letter "H", a good example of where an uppercase letter is located in the character cell.

The softkey menu displays the available choices. Softkey 1 is "Digitize Char", 2 is "Display Char", 3 is "Which Grid?", 4 is "Erase ON/OFF", 5 is "Show Chars", and 8 is "EXIT". Pressing "EXIT" will cause the program to end; control is returned to the BASIC system. Pressing "Which Grid?" will specify one of the four grids as the current grid. Pressing "Show Chars" will display an ASCII table in the current grid. The table is just like the one displayed in grid 2 at start up, unless you have changed some character definitions.

Pressing "Display Char" will display a character in the current grid. You will be asked to input the character you wish to display. Type the character and hit ENTER, or type the NUM of the character and hit ENTER. Depending on the Erase Flag, the character will be displayed on a fresh grid, or it will be displayed on top of the old contents of the grid. This capability is useful when building a new definition based on existing characters.

Pressing softkey 4, the "Erase ON/OFF" flag is toggled between ON and OFF. When ON, the current grid is cleared before each character is displayed. When OFF, the current grid is overlaid with each displayed character. An asterisk, "*", is displayed next to the ON or OFF to show the current state of Erase.

Pressing "Digitize Char" allows you to construct a new character definition. Typically, you would display in one or more of the grids whatever characters will assist you in creating a new definition, and then you would display in the current grid a character that is most like the new one you wish to define. Then you would press "Digitize Char" and begin digitizing the new character. You may use a mouse, or the arrow keys to move the cursor in the grid.

When using a mouse, move the cursor to the desired coordinate and then press the left button to Draw or the right button to Move there. Click either mouse button with the cursor outside the grid to end the definition. If using the arrow keys, move to the location you wish to Move/Draw to and press ENTER. The softkey menu will change, displaying "Draw", "Move", and "Digitize Done". Press "Move" or "Draw". When you are done, press ENTER and then "Digitize Done".

As each point is digitized, the correct value to use in CHR$ is printed on the display line. You should write these values down and use them when constructing your definitions. When you are done digitizing a character you are given the option of immediately assigning it to a character. If you wish to do so, enter the NUM of the character; if you do not wish to do so, enter -1 for the NUM. If you immediately define the character, you can then see it using "Display Char" or "Show Chars". The definition lasts until you QUIT HTBasic.

# User-Defined Lexical Orders

The lexical order rules provided with HTBasic are sufficient for most uses. But if needed, you may define your own rules. The lexical order of each character may be specified. Also, because the lexical order of some languages treat some letters as if they were two, treat some two-letter combinations as if they were one, and ignore some letters, you may define certain special cases to handle these situations. User-defined lexical rules are stored in an array and are activated with the statement:

```
LEXICAL ORDER IS Array(*)
```

When user-defined rules are in effect, SYSTEM$("LEXICAL ORDER IS") returns "USER DEFINED".

# Order Table

The main part of user-defined lexical order rules is stored in an order table, which is the first 256 elements of the array. These elements specify the lexical order for each CHR$ from 0 to 255. The order number is stored in the upper byte of each element. For example, to assign lexical order number 0 to the letter "A", and lexical order 1 to the letter "B":

```
10  INTEGER A(0:256)
20  A(NUM("A"))=SHIFT(0,-8)
30  A(NUM("B"))=SHIFT(1,-8)
```

When "A" is compared to "B", it will be smaller, since 0 is smaller than 1. "A"<B" returns 1.

Any special cases (2-to-1, 1-to-2, ignore) are noted in the lower byte of each element, and if additional information is needed it is stored in a "Special Case" table that follows after the first 256 elements of the array.

The lower byte of each element in the order table may have a value from 0 to 255. The meaning of each value is given in the following table:

| Value | Meaning |
|---|---|
| 0 | No special case |
| 1 | Ignore this character |
| 64+index | 2-to-1 translation might be needed on this character |
| 128+index | Perform 1-to-2 translation |
| 192-255 | Sub-order number exists for this character |

In the above table, "index" is a value from 0 to 63 and specifies an index into the special case table.

# Special Case Table

The length of the special case table is stored in the 257th element of the array and can be from 0 to 64 elements. The length must be stated, even if it is zero. Thus, the smallest the array specified in the LEXICAL ORDER IS statement can be is 257 elements.

The special case table starts with the 258th element of the array, which is the element immediately following the length. An index of 0 specifies the 258th element of the array.

**Note:** In the example, the BASE of the array was 0, and so the 258th element of the array is A(257). Thus, index 0 in the special case table is A(257). If the BASE of the array had been another value, the element number for the start of the special case table would have been different, but still at the 258th element.

# Ignore Characters

To expand on our previous example, let's specify that the letter "C" be ignored:

```
40   A(NUM("C"))=1
100 A(256)=0
```

Now "ABC" will be equal to "AB" and "C" will be equal to "". You can see from line 100 that we have also specified a zero length special case table. As we add to our example, we will add to line 100. Because the array A(*) has been declared with a length of 257 characters, the array declaration in line 10 will also have to be changed as we add special cases.

# 2-to-1 Translation

A 2-to-1 translation takes a two character combination, and translates it to one character.

**Note:** The strings involved are not actually changed. The change occurs internally for the string comparison and is then discarded.

To define 2-to-1 translations starting with a certain character, the order table entry for the starting character is used to store three things: 1) the order number is stored in the upper byte for use when the character occurs, but not as part of a two character combination, 2) the value 64 is stored in the lower byte to indicate that this character is the first character of one or more 2-to-1 translations, 3) an index into the special case table is stored in the lower byte.

The index into the special case table points to a list of two character combinations that all start with the same first character. The first entry in the list gives the number of two character combinations in the list. The remaining entries give the second character of each two character combination and the order number to use in place of the combination. The second character is given in the upper byte and the order number is given in the lower byte.

**Note:** The first character was given in the order table and need not be repeated in the special case table. Only the second character of each combination is given in the special case table.

For example, we might want to consider "DX" to be a single character with order number 4 and "DY" to be a single character with order number 3. For all other occurrences of the letter "D" we want "D" to have order number 2. For our example,

```
10  INTEGER A(0:259)
50  A(NUM("D"))=SHIFT(2,-8)+64+0
100 A(256)=0+3
110 A(257)=2
120 A(258)=SHIFT(NUM("X"),-8) + 4
130 A(259)=SHIFT(NUM("Y"),-8) + 3
```

Line 50 is the order table entry for the letter "D". Order number 2 will be used for "D" unless it is "DX" or "DY". The value 64 indicates one or more 2-to-1 translations exist that start with the letter "D". The value 0 is the index into the special case table.

Line 100 is the length of the special case table. Previously in our example, we had set it to zero, but we are now adding three entries to the special case table.

In line 110, A(257) is at index 0 in the special case table. This is the start of our list of two character combinations beginning with "D". Since we have two, "DX" and "DY", we set A(257) to two.

Lines 120 and 130 define "DX" and "DY" to have order numbers 4 and 3. Now the following will both be true:

"DY" <"DX" - because 3 <4
"DZ" <"DX" - because 2 < 4 ("D" < "DX").

# 1-to-2 Translation

A 1-to-2 translation takes a single character and translates it into two characters. This capability also includes intelligent handling of upper and lowercase. For example, if "E" is to be translated to "FG" then "Exyz" should be translated to "Fgxyz", while "EXYZ" should be translated to "FGXYZ".

**Note:** The strings involved are not actually changed. The change occurs internally for the string comparison and is then discarded.

To define 1-to-2 translations for a certain character, the order table entry for the character is used to store three things: 1) the first order number is stored in the upper byte, 2) the value 128 is stored in the lower byte to indicate a 1-to-2 translation, 3) an index into the special case table is stored in the lower byte.

The special case table entry contains the second order numbers for both upper and lowercase. The lower byte contains the lowercase order number, while the upper byte contains the uppercase order number. The uppercase order number is used if the initial character is uppercase and is not followed by a lowercase character.

For our example, if we want to use an order number of 5 for "F", 6 for "G", and 37 for "g", then:

```
10  INTEGER A(0:260)
60  A(NUM("E"))=SHIFT(5,-8)+128+3
100 A(256)=0+3+1
140 A(260)=SHIFT(6,-8) + 37
```

Line 100 is the length of the special case table. Previously in our example, we had set it to three, but we have now added one more entry to the special case table.

Line 60 is the order table entry for the letter "E" and line 140 is the special case entry for "E". In place of "E", two order numbers will be used, 5 and 37 for "Fg", or 5 and 6 for "FG". Now the following will all be true:

"E" > "FD"
"E" = "FG"
"E" < "FH"
"EXYZ" = "FGXYZ"
"Exyz" = "Fgxyz"

# Sub-Order Numbers

Sometimes it is useful to assign several characters the same order number, yet still collate them in a specific order. For example, it might make sense to assign all occurrences of "E", regardless of the accent, the same order number, but still allow them to be collated in a specific order. This can be accomplished using sub-order numbers. Sub-order numbers can range from 0 to 63. To assign a sub-order number to a character, set the lower byte of the order table entry to the sub-order value plus 192.

When strings are compared, if the order numbers of two characters are the same, the sub-order numbers are used to determine the lexical order. If a sub-order number has not been explicitly assigned to a character, 0 is used. A sub-order number can not be assigned to characters that are used in Ignore, 2-to-1, or 1-to-2 translations since the lower byte of the order table entry is already used.

As an example of sub-order number usage, say we wish to give "H" and "I" the same order number, 7, but wish "H" to collate before "I" using sub-order numbers. We can give "H" a sub-order number of 0 and "I" a sub-order number of 1:

```
80 A(NUM("H"))=SHIFT(7,-8)+192+0
90 A(NUM("I"))=SHIFT(7,-8)+192+1
```

# Putting User-Defined Rules Into Effect

An order number must be assigned to each value, 0 through 255, in the order table. Once this has been done, as well as assigning all special cases, the array may be specified in a LEXICAL ORDER IS statement to make it take effect. All of the user-defined order rules explained above are compatible between HTBasic and HP BASIC. HTBasic extensions to LEXICAL ORDER IS that allow user-defined upper and lowercase conversions are explained in the following paragraphs.

But first, let's complete our example:

```
5     REM USERDEF.BAS
10    INTEGER A(0:260)
20    A(65)=0
30    A(66)=256
40    A(67)=1
50    A(68)=576
60    A(69)=1411
65    A(70)=1280
75    A(71)=1536
80    A(72)=1984
90    A(73)=1985
100   A(256)=4
110   A(257)=2
120   A(258)=22532
130   A(259)=22787
140   A(260)=1573
150   FOR I=74 TO 255
160     A(I)=SHIFT(I-66,-8)
170   NEXT I
180   FOR I=0 TO 64
190     A(I)=SHIFT(I+190,-8)
200   NEXT I
210   LEXICAL ORDER IS A(*)
220   END
```

We have added lines 65, 75, and 150 to 200 to assign the characters that were not yet assigned. We have also sped up the program by pre-evaluating functions like NUM("A") and SHIFT(0,-8) wherever possible. Finally, line 210 causes all the changes to take effect.

# User-Defined UPC$/LWC$ Rules

In addition to specifying order rules, HTBasic has been enhanced to let you specify upper and lowercase conversion rules as well. This capability is an extension to HTBasic and will not work if used with HP BASIC. However, HP BASIC will not return an error; the UPC$/LWC$ rules will simply be ignored.

**Note:** There is some danger in specifying meaningless upper/lowercase rules because HTBasic uses these rules in checking the syntax of a command or program line. For example, when you type "RUN", UPC$("RUN") is compared against the list of known statements. As long as you take reasonable care in defining your rules, you shouldn't have any problems.

Upper/lowercase rules are stored in the LEXICAL ORDER IS array immediately following the special case table. The UPC$/LWC$ table consists of 257 elements. The first element must have the value 21576 to indicate that the UPC$/LWC$ table is present. The rules themselves are stored in the next 256 elements, one for each possible character. All 256 characters must be defined. In each element, the upper byte contains the UPC$ value and the lower byte contains the LWC$ value.

For example, in the USERDEF.BAS example above, the special case table ends at element A(260). Element A(261) should be assigned a value of 21576 if upper/lowercase rules are also being specified. Elements A(262) through A(517) would contain the rules for CHR$(0) through CHR$(255). To set the UPC$/LWC$ values for "A" and "a", the following statements would be used:

```
A(257+A(256))=21576
A(258+A(256)+NUM("A"))=SHIFT( NUM("A"),-8)+NUM("a")
A(258+A(256)+NUM("a"))=SHIFT( NUM("A"),-8)+NUM("a")
```

The subscript calculation in this example deserves some explanation. If the array BASE is zero then A(256) is the length of the special case table, 257+A(256) is the first element after the special case table, and 258+A(256)+NUM("*x*") is the UPC$/LWC$ definition for "*x*".

Of course, it is best to simplify these statements to:

```
A(261)=21576
A(327)=16737
A(359)=16737
```

# Example Data Files

If you installed the optional LEXICAL ORDER files during installation, in the LEXICAL subdirectory in the HTBasic directory there are several examples of user-defined LEXICAL ORDER IS tables that change both the order rules and the upper/lowercase rules. The files that are included depend on the version of HTBasic. Files for code page 850 have filenames of PC*.LEX. Files for Roman-8 have filenames of R8*.LEX. Files for the out-dated version of Roman-8 used by HP BASIC are stored in files with names HP*.LEX. Files for Latin-1 have filenames of L1*.LEX.

The file LEXICAL.BAS contains a SUB named "Lexical" that can be used to load the tables stored in these files. (This SUB is listed earlier in this chapter.) Line 50 specifies PC*.LEX files stored in the C:\HTB directory, but you may change this line to fit your needs.

# Roman-8 Character Set Support

Although Roman-8 is a fairly popular character set, especially among users of the European-language versions of HP BASIC, it is not available under Windows. Since HTBasic depends upon the operating system for character set support, you should convert from Roman-8 to the native character set of your computer. A conversion program is presented in the next section.

If you **must** use the Roman-8 character set, the sections following the translation program section describe solutions that give most of the capabilities needed. To use a different character set you must 1) change the character set used by the display, 2) change the character set produced by the keyboard, 3) change the lexical order rules, and 4) change the LABEL character definitions.

# Roman-8 Translation Program

An example program, HP2PC.BAS, is distributed with HTBasic that can be used to translate ASCII files (including program files saved in ASCII) from the Roman-8 character set to code page 850 or Latin-1. The program only translates characters that appear literally or in CHR$(xxx) statements, where "xxx" is a constant above 127. If a character is specified in any other way (for example, "CHR$(X+3)"), it is not translated. You will have to make those translations manually.

If any attribute control characters in the range CHR$(128) to CHR$(143) are seen, they are translated to the alternate range at CHR$(16) to CHR$(31) and you must add the following statement to make attribute characters be recognized in this new range:

```
CONTROL CRT,100;1
```

Several characters that exist in the Roman-8 character set are not found in code page 850 or Latin-1. When translating to code page 850, the characters in the range CHR$(144) to CHR$(160) are translated to CHR$(219), a rectangular block, to make them easy to spot and hand translate. When translating to Latin-1, the characters in the range CHR$(144) to CHR$(160) are unchanged and the Dutch guilder symbol "ƒ", CHR$(190), is translated to "*", CHR$(42), to make it easy to spot and hand translate. Other characters are translated to similar characters:

| **From...** | | | **To...** | | |
|---|---|---|---|---|---|
| **Character** | **Roman-8** | | **Character** | **PC-850** | **Latin-1** |
| Grave accent | 169 | ‘ | 96 | 96 | |
| Circumflex | 170 | ^ | 94 | 94 | |
| Tilde | 172 | ~ | | 126 | |
| Lira | 175 | £ | | 156 | 163 |
| Š | 235 | S | 83 | 83 | |
| š | 236 | s | | 115 | 115 |
| Ÿ | 238 | Y | 89 | 89 | |

To translate to code page 437, specify code page 850. The only difference is the translation for CHR$(191), the "¢" symbol. It is translated to CHR$(189) which is correct for code page 850, but should be CHR$(155) for code page 437. This minor correction can then be done by hand.

# Display Font

A Windows font containing the Roman-8 character set, HTBGROM8.FON, is supplied with HTBasic for Windows. To install the Roman-8 font, select Control Panel, Fonts, and Add.... Then change the drive and directory to LEXICAL subdirectory of the HTBasic directory (HTBWIN, default). Then select Roman8 and OK. Roman8 should then be accessible to any Windows program that uses fixed width fonts.

To select Roman8 for use with HTBasic, use the -fn command line switch, explained in the *Installing and Using* manual. Select Program Manager, TransEra HTBasic, File, Properties, and add "-fn Roman8" to the command line. For example,

```
C:\HTBWIN\HTBWIN.EXE -fn Roman8
```

**Note**: Changing the display font without changing the keyboard font causes a mismatch for characters above CHR$(128). CONFIGURE KBD, explained below, can correct this situation for the most part.

# Keyboard

The statement CONFIGURE KBD has been added to HTBasic to allow simple keyboard character set re-mapping. (This is different than CONFIGURE KEY, which re-maps function and editor keys.) CONFIGURE KBD is not a complete keyboard driver. It uses a look-up table to translate characters from one character set to another.

The files PCTOR8.KBD (formerly HP200.KBD) and L1TOR8.KBD contain the necessary keyboard re-mappings from code page 850 and Latin-1 to Roman-8. The following program will set up the re-mapping. Use either PCTOR8 or L1TOR8 in line 60, depending on the character set in use by the operating system keyboard driver. Where no translation exists for a character, CHR$(252) is returned.

```
10   !SETKBD.BAS
20   DIM Pc2hp$[256]
30   CLEAR SCREEN
40   PRINT "Set up translation string to Roman-8"
60   ASSIGN @Io TO "PCTOR8.KBD" !Use L1TOR8 for Latin-1
70   ENTER @Io;Pc2hp$
80   ASSIGN @Io TO *
90   CONFIGURE KBD 0 TO Pc2hp$
100  END
```

To enter a character without re-mapping, use the ANY CHAR function.

The syntax of the CONFIGURE KBD statement is:

CONFIGURE KBD *First_char* TO *String*$

where *First_char* is a numeric expression, rounded to an integer, that gives the first keyboard character to be re-mapped, and the first character in *String*$ gives the display character that it is re-mapped to. If the length of *String*$ is longer than one, then additional characters following *First_char* are also re-mapped.

# LEXICAL ORDER

When using Roman-8 with other versions, you must load LEXICAL ORDER rules from data files. The data files are named:

| Language | File |
|---|---|
| ASCII | R8ASCII.LEX |
| FRENCH | R8FRENCH.LEX |
| GERMAN | R8GERMAN.LEX |
| SPANISH | R8SPANIS.LEX |
| SWEDISH | R8SWEDIS.LEX |

The SUB "Lexical", listed previously in this chapter and stored on the distribution disks in the file LEXICAL.BAS, can be used to set the LEXICAL ORDER using these files. Change line 50 to specify "R8" instead of "PC" and change the directory as necessary:

```
50   A$="C:\HTB\R8"&L$[1;6]&".LEX"
```

# LABEL

The CONFIGURE LABEL statement must be used to define characters above 127 so that they match the Roman-8 character set. CONFIGURE LABEL is explained earlier in this chapter.

# LEXICAL ORDER Tables

The following pages contain LEXICAL ORDER tables for FRENCH, GERMAN, SPANISH and SWEDISH. Different tables are presented for code page 850, Roman-8 and Latin-1 character sets. No tables are given for ASCII because when the LEXICAL ORDER IS ASCII, regardless of the character set the order number is the same as the NUM of each character.

For code page 850 and the Latin-1 character sets, the order number for each character was chosen according to the following guidelines: If the character existed in the Roman-8 character set, it is given the same order number it had under HP BASIC. New alphabetic characters were added in alphabetical order. New symbol characters were added after CHR$(127) and given sequentially increasing order numbers.

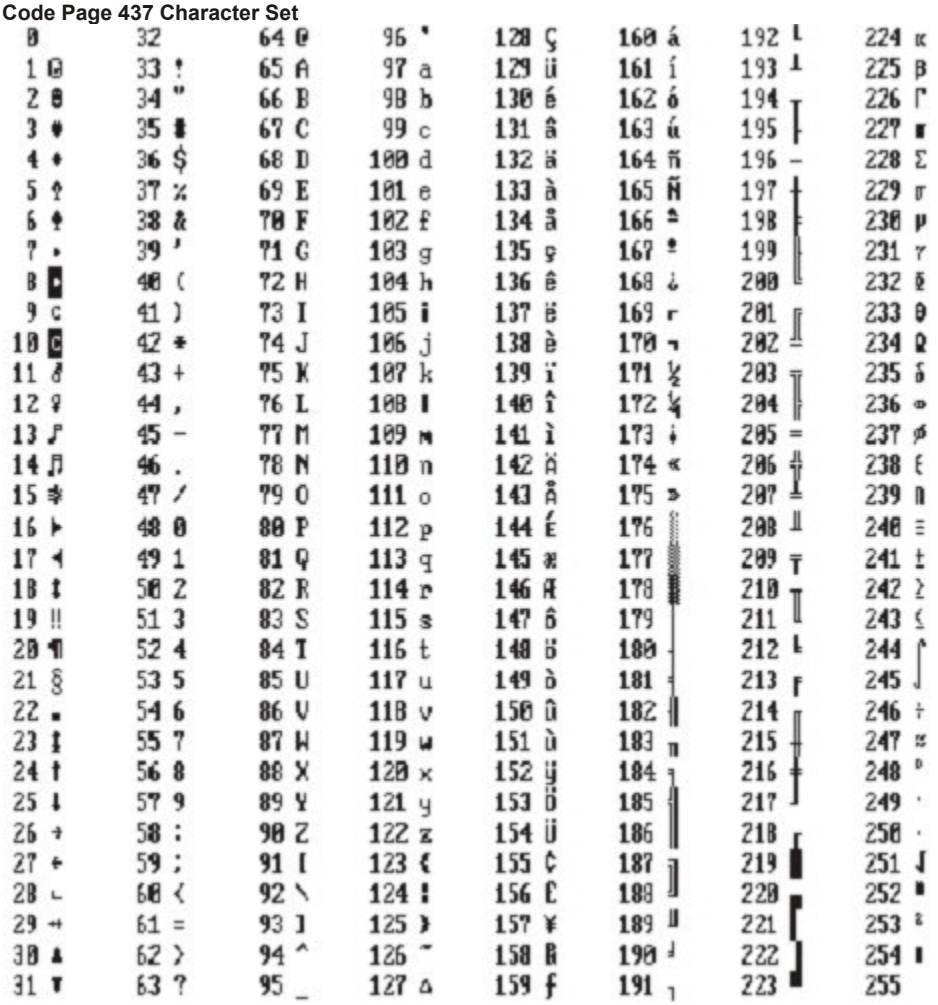
The order numbers for Roman-8 differ slightly than those in HP BASIC. This is because several characters have been added to Roman-8 since HP BASIC was created. The new characters are 177, 178 and 242 to 245.

Each table contains the Order number, the NUM, and the CHR$ for each character. If the character is ignored during comparisons, the order will be blank. For two-character combinations that have a single order number, the two characters are given in the Chr$ column, but no Num is listed. For characters that are expanded into two characters, the two characters are listed in the Order column. If the original character is uppercase, two expansions are listed. Remember that two order numbers are produced, not just one. If the character has a sub-order number, it is given following a decimal point in the Order column.

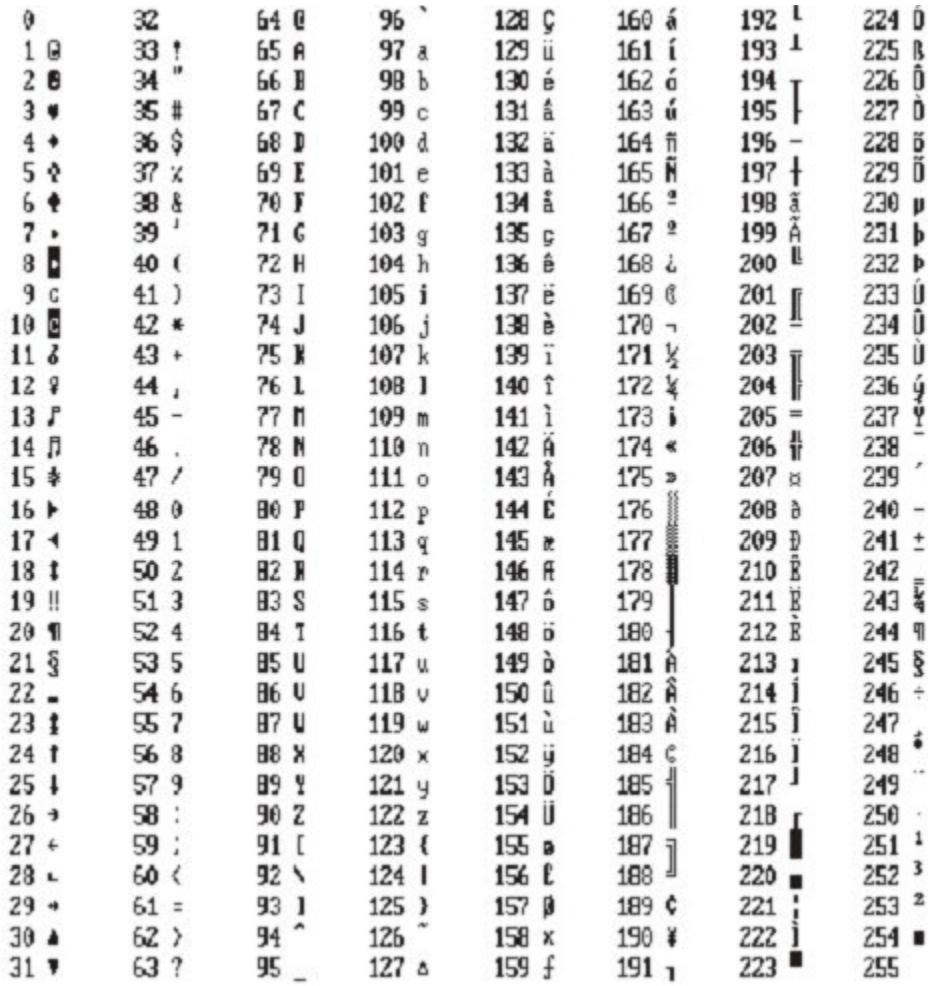
# Character Set Tables

The following pages contain character set and LEXICAL ORDER tables for code pages 850, 437, Roman-8 and Latin-1 character sets.

**Code Page 437 Character Set**



**Code Page 850 Character Set**



**LEXICAL ORDER IS FRENCH (Code Page 850)**

| Order | Num | Chr |
|---|---|---|
| | 45 | - |
| 0 | 0 | |
| 1 | 1 | ☺ |
| 2 | 2 | ☻ |
| 3 | 3 | ♥ |
| 4 | 4 | ♦ |
| 5 | 5 | ♣ |
| 6 | 6 | ♠ |
| 7 | 7 | • |
| 8 | 8 | ◘ |
| 9 | 9 | ○ |
| 10 | 10 | ◙ |
| 11 | 11 | ♂ |
| 12 | 12 | ♀ |
| 13 | 13 | ♪ |
| 14 | 14 | ♫ |
| 15 | 15 | ☼ |
| 16 | 16 | ► |
| 17 | 17 | ◄ |
| 18 | 18 | ↕ |
| 19 | 19 | ‼ |
| 20 | 20 | ¶ |
| 21 | 21 | § |
| 22 | 22 | ▬ |
| 23 | 23 | ↨ |
| 24 | 24 | ↑ |
| 25 | 25 | ↓ |
| 26 | 26 | → |
| 27 | 27 | ← |
| 28 | 28 | ∟ |
| 29 | 29 | ↔ |
| 30 | 30 | ▲ |
| 31 | 31 | ▼ |
| 32 | 32 | |
| 33 | 33 | ! |
| 34 | 34 | " |
| 35 | 35 | # |
| 36 | 36 | $ |
| 37 | 37 | % |
| 38 | 38 | & |
| 39 | 39 | ' |
| 40 | 40 | ( |
| 41 | 41 | ) |
| 42 | 42 | * |
| 43 | 43 | + |
| 44 | 44 | , |
| 45 | 46 | . |
| 46 | 47 | / |
| 47 | 48 | 0 |
| 48 | 49 | 1 |
| 49 | 50 | 2 |
| 50 | 51 | 3 |
| 51 | 52 | 4 |
| 52 | 53 | 5 |
| 53 | 54 | 6 |
| 54 | 55 | 7 |
| 55 | 56 | 8 |
| 56 | 57 | 9 |
| 57 | 58 | : |
| 58 | 59 | ; |
| 59 | 60 | < |
| 60 | 61 | = |
| 61 | 62 | > |
| 62 | 63 | ? |
| 63 | 64 | @ |
| 64 | 65 | A |
| 64 | 142 | Ä |
| 64 | 143 | Å |
| 64 | 146 | Æ |
| 64 | 181 | Á |
| 64 | 182 | Â |
| 64 | 183 | À |
| 64 | 199 | ã |
| 65 | 66 | B |
| 66 | 67 | C |
| 66 | 128 | Ç |
| 67 | 68 | D |
| 68 | 209 | Ð |
| 69 | 69 | E |
| 69 | 144 | É |
| 69 | 210 | Ê |
| 69 | 211 | Ë |
| 69 | 212 | È |
| 70 | 70 | F |
| 71 | 71 | G |
| 72 | 72 | H |
| 73 | 73 | I |
| 73 | 214 | Í |
| 73 | 215 | Î |
| 73 | 216 | Ï |
| 73 | 222 | Ì |
| 74 | 74 | J |
| 75 | 75 | K |
| 76 | 76 | L |
| 77 | 77 | M |
| 78 | 78 | N |
| 79 | 165 | Ñ |
| 80 | 79 | O |
| 80 | 153 | Ö |
| 80 | 157 | Ø |
| 80 | 224 | Ó |
| 80 | 226 | Ô |
| 80 | 227 | Ò |
| 80 | 229 | Õ |
| 81 | 80 | P |
| 82 | 81 | Q |
| 83 | 82 | R |
| 84 | 83 | S |
| 85 | 84 | T |
| 86 | 85 | U |
| 87 | 154 | Ü |
| 87 | 233 | Ú |
| 87 | 234 | Û |
| 87 | 235 | Ù |
| 88 | 86 | V |
| 89 | 87 | W |
| 90 | 88 | X |
| 91 | 89 | Y |
| 91 | 237 | Ý |
| 92 | 90 | Z |
| 93 | 232 | Þ |
| 94 | 91 | [ |
| 95 | 92 | \ |
| 96 | 93 | ] |
| 97 | 94 | ^ |
| 98 | 95 | _ |
| 99 | 96 | ` |
| 100 | 97 | a |
| 100 | 131 | â |
| 100 | 132 | ä |
| 100 | 133 | à |
| 100 | 134 | ã |
| 100 | 145 | æ |
| 100 | 160 | á |
| 100 | 198 | ã |
| 101 | 98 | b |
| 102 | 99 | c |
| 103 | 135 | ç |
| 104 | 100 | d |
| 105 | 208 | ð |
| 106 | 101 | e |
| 106 | 130 | é |
| 106 | 136 | ê |
| 106 | 137 | ë |
| 106 | 138 | è |
| 107 | 102 | f |
| 108 | 103 | g |
| 109 | 104 | h |
| 110 | 105 | i |
| 110 | 139 | ï |
| 110 | 140 | î |
| 110 | 141 | ì |
| 110 | 161 | í |
| 111 | 106 | j |
| 112 | 107 | k |
| 113 | 108 | l |
| 114 | 109 | m |
| 115 | 110 | n |
| 116 | 164 | ñ |
| 117 | 111 | o |
| 117 | 147 | ô |
| 117 | 148 | ö |
| 117 | 149 | ò |
| 117 | 155 | œ |
| 117 | 162 | ó |
| 117 | 228 | õ |
| 118 | 112 | p |
| 119 | 113 | q |
| 120 | 114 | r |
| 121 | 115 | s |
| ss | 225 | ß |
| 122 | 116 | t |
| 123 | 117 | u |
| 124 | 129 | ü |
| 124 | 150 | û |
| 124 | 151 | ù |
| 124 | 163 | ú |
| 125 | 118 | v |
| 126 | 119 | w |
| 127 | 120 | x |
| 128 | 121 | y |
| 128 | 152 | ÿ |
| 128 | 236 | ý |
| 129 | 122 | z |
| 130 | 231 | þ |
| 131 | 123 | { |
| 132 | 124 | \| |
| 133 | 125 | } |
| 134 | 126 | ~ |
| 135 | 239 | ´ |
| 138 | 249 | ¨ |
| 141 | 238 | ¯ |
| 142 | 248 | ° |
| 143 | 173 | ¡ |
| 144 | 168 | ¿ |
| 145 | 207 | ¤ |
| 146 | 156 | £ |
| 147 | 190 | ¥ |
| 148 | 245 | § |
| 149 | 159 | ƒ |
| 150 | 189 | ¢ |
| 151 | 240 | - |
| 152 | 172 | ¼ |
| 153 | 171 | ½ |
| 154 | 166 | ª |
| 155 | 167 | º |
| 156 | 174 | « |
| 157 | 254 | ■ |
| 158 | 175 | » |
| 159 | 241 | ± |
| 160 | 127 | ∆ |
| 161 | 158 | x |
| 162 | 169 | ® |
| 163 | 170 | ¬ |
| 164 | 176 | ▒ |
| 165 | 177 | ▓ |
| 166 | 178 | │ |
| 167 | 179 | ┤ |
| 168 | 180 | Á |
| 169 | 184 | © |
| 170 | 185 | ╣ |
| 171 | 186 | ║ |
| 172 | 187 | ╗ |
| 173 | 188 | ╝ |
| 174 | 191 | ┐ |
| 175 | 192 | └ |
| 176 | 193 | ┴ |
| 177 | 194 | ┬ |
| 178 | 195 | ├ |
| 179 | 196 | ─ |
| 180 | 197 | ┼ |
| 181 | 200 | ╚ |
| 182 | 201 | ╔ |
| 183 | 202 | ╩ |
| 184 | 203 | ╦ |
| 185 | 204 | ╠ |
| 186 | 205 | ═ |
| 187 | 206 | ╬ |
| 188 | 213 | ¦ |
| 189 | 217 | ┘ |
| 190 | 210 | ┌ |
| 191 | 219 | ▌ |
| 192 | 220 | ▄ |
| 193 | 221 | ¦ |
| 194 | 223 | ▀ |
| 195 | 230 | µ |
| 196 | 242 | ‗ |
| 197 | 243 | ¾ |
| 198 | 244 | ¶ |
| 199 | 246 | ÷ |
| 200 | 247 | ¸ |
| 201 | 250 | · |
| 202 | 251 | ¹ |
| 203 | 252 | ³ |
| 204 | 253 | ² |
| 205 | 255 | |

**LEXICAL ORDER IS GERMAN (Code Page 850)**

| Order | Num | Chr |
|---|---|---|
| 0 | 0 | |
| 1 | 1 | ☺ |
| 2 | 2 | ☻ |
| 3 | 3 | ♥ |
| 4 | 4 | ♦ |
| 5 | 5 | ♣ |
| 6 | 6 | ♠ |
| 7 | 7 | • |
| 8 | 8 | ◘ |
| 9 | 9 | ○ |
| 10 | 10 | ◙ |
| 11 | 11 | ♂ |
| 12 | 12 | ♀ |
| 13 | 13 | ♪ |
| 14 | 14 | ♫ |
| 15 | 15 | ☼ |
| 16 | 16 | ► |
| 17 | 17 | ◄ |
| 18 | 18 | ↕ |
| 19 | 19 | ‼ |
| 20 | 20 | ¶ |
| 21 | 21 | § |
| 22 | 22 | ▬ |
| 23 | 23 | ↨ |
| 24 | 24 | ↑ |
| 25 | 25 | ↓ |
| 26 | 26 | → |
| 27 | 27 | ← |
| 28 | 28 | ∟ |
| 29 | 29 | ↔ |
| 30 | 30 | ▲ |
| 31 | 31 | ▼ |
| 32 | 32 | |
| 33 | 33 | ! |
| 34 | 34 | " |
| 35 | 35 | # |
| 36 | 36 | $ |
| 37 | 37 | % |
| 38 | 38 | & |
| 39 | 39 | ' |
| 40 | 40 | ( |
| 41 | 41 | ) |
| 42 | 42 | * |
| 43 | 43 | + |
| 44 | 44 | , |
| 45 | 45 | - |
| 46 | 46 | . |
| 47 | 47 | / |
| 48 | 48 | 0 |
| 49 | 49 | 1 |
| 50 | 50 | 2 |
| 51 | 51 | 3 |
| 52 | 52 | 4 |
| 53 | 53 | 5 |
| 54 | 54 | 6 |
| 55 | 55 | 7 |
| 56 | 56 | 8 |
| 57 | 57 | 9 |
| 58 | 58 | : |
| 59 | 59 | ; |
| 60 | 60 | < |
| 61 | 61 | = |
| 62 | 62 | > |
| 63 | 63 | ? |
| 64 | 64 | @ |
| 65 | 65 | A |
| AE/Ae | 142 | Ä |
| 66 | 146 | Æ |
| 67 | 143 | Å |
| 68 | 181 | Á |
| 69 | 183 | À |
| 70 | 182 | Â |
| 71 | 199 | Ã |
| 72 | 66 | B |
| 73 | 67 | C |
| 74 | 128 | Ç |
| 75 | 68 | D |
| 76 | 209 | Ð |
| 77 | 69 | E |
| 78 | 144 | É |
| 79 | 212 | È |
| 80 | 210 | Ê |
| 81 | 211 | Ë |
| 82 | 70 | F |
| 83 | 71 | G |
| 84 | 72 | H |
| 85 | 73 | I |
| 86 | 214 | Í |
| 87 | 222 | Ì |
| 88 | 215 | Î |
| 89 | 216 | Ï |
| 90 | 74 | J |
| 91 | 75 | K |
| 92 | 76 | L |
| 93 | 77 | M |
| 94 | 78 | N |
| 95 | 165 | Ñ |
| 96 | 79 | O |
| OE/Oe | 153 | Ö |
| 97 | 224 | Ó |
| 98 | 227 | Ò |
| 99 | 226 | Ô |
| 100 | 229 | Õ |
| 101 | 157 | Ø |
| 102 | 80 | P |
| 103 | 81 | Q |
| 104 | 82 | R |
| 105 | 83 | S |
| 107 | 84 | T |
| 108 | 85 | U |
| UE/Ue | 154 | Ü |
| 109 | 233 | Ú |
| 110 | 235 | Ù |
| 111 | 234 | Û |
| 112 | 86 | V |
| 113 | 87 | W |
| 114 | 88 | X |
| 115 | 89 | Y |
| 116 | 237 | Ý |
| 117 | 90 | Z |
| 118 | 232 | Þ |
| 119 | 91 | [ |
| 120 | 92 | \ |
| 121 | 93 | ] |
| 122 | 94 | ^ |
| 123 | 95 | _ |
| 124 | 96 | ` |
| 125 | 97 | a |
| ae | 132 | ä |
| 126 | 145 | æ |
| 127 | 134 | å |
| 128 | 160 | á |
| 129 | 133 | à |
| 130 | 131 | â |
| 131 | 198 | ã |
| 132 | 98 | b |
| 133 | 99 | c |
| 134 | 135 | ç |
| 135 | 100 | d |
| 136 | 208 | ð |
| 137 | 101 | e |
| 138 | 130 | é |
| 139 | 138 | è |
| 140 | 136 | ê |
| 141 | 137 | ë |
| 142 | 102 | f |
| 143 | 103 | g |
| 144 | 104 | h |
| 145 | 105 | i |
| 146 | 161 | í |
| 147 | 141 | ì |
| 148 | 140 | î |
| 149 | 139 | ï |
| 150 | 106 | j |
| 151 | 107 | k |
| 152 | 108 | l |
| 153 | 109 | m |
| 154 | 110 | n |
| 155 | 164 | ñ |
| 156 | 111 | o |
| oe | 148 | ö |
| 157 | 162 | ó |
| 158 | 149 | ò |
| 159 | 147 | ô |
| 160 | 228 | õ |
| 161 | 155 | œ |
| 162 | 112 | p |
| 163 | 113 | q |
| 164 | 114 | r |
| 165 | 115 | s |
| ss | 225 | ß |
| 167 | 116 | t |
| 168 | 117 | u |
| ue | 129 | ü |
| 169 | 163 | ú |
| 170 | 151 | ù |
| 171 | 150 | û |
| 172 | 118 | v |
| 173 | 119 | w |
| 174 | 120 | x |
| 175 | 121 | y |
| 176 | 152 | ÿ |
| 176.1 | 236 | ý |
| 177 | 122 | z |
| 178 | 231 | þ |
| 179 | 123 | { |
| 180 | 124 | \| |
| 181 | 125 | } |
| 182 | 126 | ~ |
| 183 | 239 | ´ |
| 186 | 249 | ¨ |
| 189 | 238 | ¯ |
| 190 | 248 | ° |
| 191 | 173 | ¡ |
| 192 | 168 | ¿ |
| 193 | 207 | ⌐ |
| 194 | 156 | ¬ |
| 195 | 190 | ½ |
| 196 | 245 | ¼ |
| 197 | 159 | « |
| 198 | 189 | » |
| 199 | 240 | ± |
| 200 | 172 | ª |
| 201 | 171 | º |
| 202 | 166 | ª |
| 203 | 167 | º |
| 204 | 174 | « |
| 205 | 254 | » |
| 206 | 175 | » |
| 207 | 241 | ± |
| 208 | 127 | ¤ |
| 209 | 158 | x |
| 210 | 169 | ® |
| 211 | 170 | ¬ |
| 212 | 176 | ▒ |
| 213 | 177 | ▒ |
| 214 | 178 | ▓ |
| 215 | 179 | │ |
| 216 | 180 | ┤ |
| 217 | 184 | © |
| 218 | 185 | ╣ |
| 219 | 186 | ║ |
| 220 | 187 | ╗ |
| 221 | 188 | ╝ |
| 222 | 191 | ┐ |
| 223 | 192 | └ |
| 224 | 193 | ┴ |
| 225 | 194 | ┬ |
| 226 | 195 | ├ |
| 227 | 196 | ─ |
| 228 | 197 | ┼ |
| 229 | 200 | ╚ |
| 230 | 201 | ╔ |
| 231 | 202 | ╩ |
| 232 | 203 | ╦ |
| 233 | 204 | ╠ |
| 234 | 205 | ═ |
| 235 | 206 | ╬ |
| 236 | 213 | ╒ |
| 237 | 217 | ┘ |
| 238 | 218 | ┌ |
| 239 | 219 | ▒ |
| 240 | 220 | ▄ |
| 241 | 221 | ¦ |
| 242 | 223 | ▀ |
| 243 | 230 | µ |
| 244 | 242 | ≡ |
| 245 | 243 | ± |
| 246 | 244 | ¶ |
| 247 | 246 | ÷ |
| 248 | 247 | ° |
| 249 | 250 | · |
| 250 | 251 | ¹ |
| 251 | 252 | ³ |
| 252 | 253 | ² |
| 253 | 255 | |

**LEXICAL ORDER IS SPANISH (Code Page 850)**

| Order | Num | Chr | Order | Num | Chr | Order | Num | Chr | Order | Num | Chr | Order | Num | Chr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | | 53 | 53 | 5 | 83 | 227 | Ô | 114 | 106 | j | 158 | 166 | ª |
| 1 | 1 | ☺ | 54 | 54 | 6 | 83 | 229 | Õ | 115 | 107 | k | 159 | 167 | º |
| 2 | 2 | ☻ | 55 | 55 | 7 | 84 | 80 | P | 116 | 108 | l | 160 | 171 | « |
| 3 | 3 | ♥ | 56 | 56 | 8 | 85 | 81 | Q | 117 | | ll | 161 | 254 | ■ |
| 4 | 4 | ♦ | 57 | 57 | 9 | 86 | 82 | R | 117 | | lL | 162 | 175 | » |
| 5 | 5 | ♣ | 58 | 58 | : | 87 | 83 | S | 118 | 109 | m | 163 | 241 | ± |
| 6 | 6 | ♠ | 59 | 59 | ; | 89 | 84 | T | 119 | 110 | n | 164 | 127 | ∩ |
| 7 | 7 | • | 60 | 60 | < | 90 | 85 | U | 120 | 164 | ñ | 165 | 158 | × |
| 8 | 8 | ◘ | 61 | 61 | = | 90 | 154 | Ü | 121 | 111 | o | 166 | 169 | ® |
| 9 | 9 | ○ | 62 | 62 | > | 90 | 233 | Ù | 121 | 147 | ô | 167 | 170 | ¬ |
| 10 | 10 | ◙ | 63 | 63 | ? | 90 | 234 | Û | 121 | 148 | ö | 168 | 176 | ▒ |
| 11 | 11 | ♂ | 64 | 64 | @ | 90 | 235 | Ù | 121 | 149 | ò | 169 | 177 | ▒ |
| 12 | 12 | ♀ | 65 | 65 | A | 91 | 86 | V | 121 | 155 | ø | 170 | 178 | ▓ |
| 13 | 13 | ♪ | 65 | 142 | Ä | 92 | 87 | W | 121 | 162 | ó | 171 | 179 | │ |
| 14 | 14 | ♫ | 65 | 143 | Å | 93 | 88 | X | 121 | 220 | ö | 172 | 180 | ┤ |
| 15 | 15 | ☼ | 65 | 146 | Æ | 94 | 89 | Y | 122 | 112 | p | 173 | 184 | © |
| 16 | 16 | ► | 65 | 181 | Á | 94 | 237 | Ý | 123 | 113 | q | 174 | 185 | ╣ |
| 17 | 17 | ◄ | 65 | 182 | Â | 95 | 90 | Z | 124 | 114 | r | 175 | 186 | ║ |
| 18 | 18 | ↕ | 65 | 183 | À | 96 | 232 | Þ | 125 | 115 | s | 176 | 187 | ╗ |
| 19 | 19 | ‼ | 65 | 199 | Ã | 97 | 91 | [ | ss | | 225 | 177 | 188 | ╝ |
| 20 | 20 | ¶ | 66 | 66 | B | 98 | 92 | \ | 127 | 116 | t | 178 | 191 | ┐ |
| 21 | 21 | § | 67 | 67 | C | 99 | 93 | ] | 128 | 117 | u | 179 | 192 | └ |
| 22 | 22 | ▬ | 67 | 128 | Ç | 100 | 94 | ^ | 128 | 129 | ü | 180 | 193 | ┴ |
| 23 | 23 | ↨ | 68 | | Ch | 101 | 95 | _ | 128 | 150 | û | 181 | 194 | ┬ |
| 24 | 24 | ↑ | 60 | | CH | 102 | 96 | ` | 120 | 151 | ù | 102 | 195 | ├ |
| 25 | 25 | ↓ | 69 | 68 | D | 103 | 97 | a | 128 | 163 | ú | 183 | 196 | ─ |
| 26 | 26 | → | 70 | 209 | Ð | 103 | 131 | â | 129 | 118 | v | 184 | 197 | ┼ |
| 27 | 27 | ← | 71 | 69 | E | 103 | 132 | ä | 130 | 119 | w | 185 | 200 | ╚ |
| 28 | 28 | ∟ | 71 | 144 | É | 103 | 133 | à | 131 | 120 | x | 186 | 201 | ╔ |
| 29 | 29 | ↔ | 71 | 210 | Ê | 103 | 134 | å | 132 | 121 | y | 187 | 202 | ╩ |
| 30 | 30 | ▲ | 71 | 211 | Ë | 103 | 145 | æ | 132 | 152 | ÿ | 188 | 203 | ╦ |
| 31 | 31 | ▼ | 71 | 212 | È | 103 | 160 | á | 132 | 236 | ý | 189 | 204 | ╠ |
| 32 | 32 | | 72 | 70 | F | 103 | 198 | ã | 133 | 122 | z | 190 | 205 | ═ |
| 33 | 33 | ! | 73 | 71 | G | 104 | 98 | b | 134 | 231 | þ | 191 | 206 | ╬ |
| 34 | 34 | " | 74 | 72 | H | 105 | 99 | c | 135 | 123 | { | 192 | 210 | ╥ |
| 35 | 35 | # | 75 | 73 | I | 105 | 135 | ç | 136 | 124 | \| | 193 | 217 | ┘ |
| 36 | 36 | $ | 75 | 214 | Í | 106 | | ch | 137 | 125 | } | 194 | 218 | ┌ |
| 37 | 37 | % | 75 | 215 | Î | 106 | | cH | 138 | 126 | ~ | 195 | 219 | ▌ |
| 38 | 38 | & | 75 | 216 | Ì | 107 | 100 | d | 139 | 239 | ´ | 196 | 220 | ▄ |
| 39 | 39 | ' | 75 | 222 | İ | 108 | 208 | à | 142 | 249 | ¯ | 197 | 221 | ¦ |
| 40 | 40 | ( | 76 | 74 | J | 109 | 101 | e | 145 | 238 | ¯ | 198 | 223 | ▀ |
| 41 | 41 | ) | 77 | 75 | K | 109 | 130 | é | 146 | 248 | ° | 199 | 230 | µ |
| 42 | 42 | * | 78 | 76 | L | 109 | 136 | ê | 147 | 173 | ¡ | 200 | 242 | ‗ |
| 43 | 43 | + | 79 | | LI | 109 | 137 | ë | 148 | 168 | ¿ | 201 | 243 | ¾ |
| 44 | 44 | , | 79 | | LL | 109 | 138 | è | 149 | 207 | ¤ | 202 | 244 | ¶ |
| 45 | 45 | - | 80 | 77 | M | 110 | 102 | f | 150 | 156 | £ | 203 | 240 | ‐ |
| 46 | 46 | . | 81 | 78 | N | 111 | 103 | g | 151 | 190 | ¥ | 204 | 247 | ¸ |
| 47 | 47 | / | 82 | 165 | Ñ | 112 | 104 | h | 152 | 245 | ğ | 205 | 250 | · |
| 48 | 48 | 0 | 83 | 79 | O | 113 | 105 | i | 153 | 159 | ƒ | 206 | 251 | ¹ |
| 49 | 49 | 1 | 83 | 153 | Ö | 113 | 139 | ï | 154 | 189 | ¢ | 207 | 252 | ³ |
| 50 | 50 | 2 | 83 | 157 | Ø | 113 | 140 | î | 155 | 240 | – | 208 | 253 | ² |
| 51 | 51 | 3 | 83 | 224 | Ó | 113 | 141 | ì | 156 | 172 | ¼ | 209 | 255 | |
| 52 | 52 | 4 | 83 | 226 | Ô | 113 | 161 | í | 157 | 171 | ½ | | | |

**LEXICAL ORDER IS SWEDISH (Code Page 850)**

| Order | Num | Chr | Order | Num | Chr | Order | Num | Chr | Order | Num | Chr | Order | Num | Chr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | | 52 | 52 | 4 | 104 | 214 | Ì | 155 | 134 | ä | 211 | 175 | » |
| 1 | 1 | Ξ | 53 | 53 | 5 | 105 | 222 | Ì | 156 | 160 | á | 212 | 241 | ± |
| 2 | 2 | ▣ | 54 | 54 | 6 | 106 | 215 | Î | 157 | 133 | à | 213 | 127 | ▫ |
| 3 | 3 | ▮ | 55 | 55 | 7 | 107 | 216 | Ï | 158 | 131 | â | 214 | 158 | x |
| 4 | 4 | ♦ | 56 | 56 | 8 | 108 | 165 | Ñ | 159 | 132 | ä | 215 | 169 | ⌀ |
| 5 | 5 | ♣ | 57 | 57 | 9 | 109 | 224 | Ò | 160 | 198 | ã | 216 | 170 | ¬ |
| 6 | 6 | ♠ | 58 | 58 | : | 110 | 227 | Ô | 161 | 135 | ç | 217 | 176 | ▒ |
| 7 | 7 | • | 59 | 59 | ; | 111 | 226 | Ô | 162 | 208 | ð | 218 | 177 | ▒ |
| 8 | 8 | ◘ | 60 | 60 | < | 112 | 153 | Ö | 163 | 138 | è | 219 | 178 | ▒ |
| 9 | 9 | ○ | 61 | 61 | - | 113 | 229 | Õ | 164 | 136 | ê | 220 | 179 | │ |
| 10 | 10 | ◙ | 62 | 62 | > | 114 | 157 | Œ | 165 | 137 | ë | 221 | 180 | ┤ |
| 11 | 11 | ♂ | 63 | 63 | ? | 116 | 233 | Ú | 166 | 161 | í | 222 | 184 | © |
| 12 | 12 | ♀ | 64 | 64 | @ | 117 | 235 | Ù | 167 | 141 | ì | 223 | 185 | ╣ |
| 13 | 13 | ♪ | 65 | 65 | A | 118 | 234 | Û | 168 | 140 | î | 224 | 186 | ▌ |
| 14 | 14 | ♫ | 66 | 66 | B | 119 | 154 | Ü | 169 | 139 | ï | 225 | 187 | ▐ |
| 15 | 15 | ☼ | 67 | 67 | C | 120 | 237 | Ÿ | 170 | 164 | ñ | 226 | 188 | ▄ |
| 16 | 16 | ► | 68 | 68 | D | 121 | 232 | Þ | 171 | 162 | ó | 227 | 191 | ┐ |
| 17 | 17 | ◄ | 69 | 69 | E | 122 | 91 | [ | 172 | 149 | ò | 228 | 192 | └ |
| 18 | 18 | ↕ | 70 | 70 | F | 123 | 92 | \ | 173 | 147 | ô | 229 | 193 | ┴ |
| 19 | 19 | ‼ | 71 | 71 | G | 124 | 93 | ] | 174 | 148 | ö | 230 | 194 | ┬ |
| 20 | 20 | ¶ | 72 | 72 | H | 125 | 94 | ^ | 175 | 228 | õ | 231 | 195 | ├ |
| 21 | 21 | § | 73 | 73 | I | 126 | 95 | _ | 176 | 155 | œ | 232 | 196 | ─ |
| 22 | 22 | ▬ | 74 | 74 | J | 127 | 96 | ` | 178 | 163 | ú | 233 | 197 | ┼ |
| 23 | 23 | ↨ | 75 | 75 | K | 128 | 97 | a | 179 | 151 | ù | 234 | 200 | ╚ |
| 24 | 24 | ↑ | 76 | 76 | L | 129 | 98 | b | 180 | 150 | û | 235 | 201 | ▌ |
| 25 | 25 | ↓ | 77 | 77 | M | 130 | 99 | c | 181 | 129 | ü | 236 | 202 | ▌ |
| 26 | 26 | → | 78 | 78 | N | 131 | 100 | d | 182 | 152 | ÿ | 237 | 203 | ▀ |
| 27 | 27 | ← | 79 | 79 | O | 132 | 101 | e | 182.1 | 236 | ý | 238 | 204 | ▀ |
| 28 | 28 | ∟ | 80 | 80 | P | 132 | 130 | é | 183 | 231 | þ | 239 | 205 | ═ |
| 29 | 29 | ↔ | 81 | 81 | Q | 133 | 102 | f | 184 | 123 | ¢ | 240 | 206 | ▀ |
| 30 | 30 | ▲ | 82 | 82 | R | 134 | 103 | g | 185 | 124 | ¡ | 241 | 213 | ╒ |
| 31 | 31 | ▼ | 83 | 83 | S | 135 | 104 | h | 186 | 125 | ¤ | 242 | 217 | ┘ |
| 32 | 32 | | 84 | 84 | T | 136 | 105 | i | 187 | 126 | ~ | 243 | 218 | ┌ |
| 33 | 33 | ! | 85 | 85 | U | 137 | 106 | j | 188 | 239 | ´ | 244 | 219 | ▉ |
| 34 | 34 | " | 86 | 86 | V | 138 | 107 | k | 191 | 249 | ¨ | 245 | 220 | ▄ |
| 35 | 35 | # | 87 | 87 | W | 139 | 108 | l | 194 | 238 | ¯ | 246 | 221 | ▌ |
| 36 | 36 | $ | 88 | 88 | X | 140 | 109 | m | 195 | 248 | ° | 247 | 223 | ▀ |
| 37 | 37 | % | 89 | 89 | Y | 141 | 110 | n | 196 | 173 | ¿ | 248 | 230 | µ |
| 38 | 38 | & | 90 | 90 | Z | 142 | 111 | o | 197 | 168 | ¿ | 249 | 242 | ≥ |
| 39 | 39 | ' | 91 | 146 | Æ | 143 | 112 | p | 198 | 207 | ¤ | 250 | 243 | ≤ |
| 40 | 40 | ( | 92 | 143 | Å | 144 | 113 | q | 199 | 156 | £ | 251 | 244 | ¶ |
| 41 | 41 | ) | 93 | 181 | Á | 145 | 114 | r | 200 | 190 | ¥ | 252 | 246 | ÷ |
| 42 | 42 | * | 94 | 183 | À | 146 | 115 | s | 201 | 245 | § | 253 | 247 | ≈ |
| 43 | 43 | + | 95 | 182 | Â | ss | 225 | ß | 202 | 159 | f | 254 | 250 | · |
| 44 | 44 | , | 96 | 142 | Ä | 147 | 116 | t | 203 | 189 | ¢ | 255.0 | 251 | ¹ |
| 45 | 45 | - | 97 | 199 | Ã | 148 | 117 | u | 204 | 240 | - | 255.1 | 252 | ³ |
| 46 | 46 | . | 98 | 128 | Ç | 149 | 118 | v | 205 | 172 | ¼ | 255.2 | 253 | ² |
| 47 | 47 | / | 99 | 209 | Ð | 150 | 119 | w | 206 | 171 | ½ | 255.3 | 255 | |
| 48 | 48 | 0 | 100 | 144 | É | 151 | 120 | x | 207 | 166 | ª | | | |
| 49 | 49 | 1 | 101 | 212 | È | 152 | 121 | y | 208 | 167 | º | | | |
| 50 | 50 | 2 | 102 | 210 | Ê | 153 | 122 | z | 209 | 174 | « | | | |
| 51 | 51 | 3 | 103 | 211 | Ë | 154 | 145 | æ | 210 | 254 | ▮ | | | |

**Roman-8 Character Set**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 NU | 32 | 64 @ | 96 ` | 128 Ç | 160 á | 192 À | 224 Á |
| 1 SH | 33 ! | 65 A | 97 a | 129 ü | 161 í | 193 È | 225 ã |
| 2 SX | 34 " | 66 B | 98 b | 130 é | 162 ó | 194 Ò | 226 ã |
| 3 EX | 35 # | 67 C | 99 c | 131 â | 163 ú | 195 Ù | 227 Þ |
| 4 ET | 36 $ | 68 D | 100 d | 132 ä | 164 ñ | 196 á | 228 ð |
| 5 EQ | 37 % | 69 E | 101 e | 133 à | 165 Ñ | 197 é | 229 Í |
| 6 AK | 38 & | 70 F | 102 f | 134 å | 166 Î | 198 ó | 230 Ì |
| 7 BL | 39 ' | 71 G | 103 g | 135 ç | 167 Ï | 199 ú | 231 Ó |
| 8 BS | 40 ( | 72 H | 104 h | 136 ê | 168 ¿ | 200 à | 232 Ò |
| 9 HT | 41 ) | 73 I | 105 i | 137 ë | 169 ® | 201 è | 233 Õ |
| 10 LF | 42 * | 74 J | 106 j | 138 è | 170 ¬ | 202 ò | 234 õ |
| 11 VT | 43 + | 75 K | 107 k | 139 ï | 171 ½ | 203 ù | 235 Š |
| 12 FF | 44 , | 76 L | 108 l | 140 î | 172 ¼ | 204 ä | 236 š |
| 13 CR | 45 - | 77 M | 109 m | 141 ì | 173 ¡ | 205 ë | 237 Ú |
| 14 SO | 46 . | 78 N | 110 n | 142 Ä | 174 « | 206 ö | 238 Ÿ |
| 15 SI | 47 / | 79 O | 111 o | 143 Å | 175 » | 207 ü | 239 ÿ |
| 16 DL | 48 0 | 80 P | 112 p | 144 É | 176 ã | 208 Å | 240 Þ |
| 17 D1 | 49 1 | 81 Q | 113 q | 145 æ | 177 Ý | 209 î | 241 þ |
| 18 D2 | 50 2 | 82 R | 114 r | 146 Æ | 178 ý | 210 Ø | 242 · |
| 19 D3 | 51 3 | 83 S | 115 s | 147 ô | 179 ° | 211 Æ | 243 µ |
| 20 DL | 52 4 | 84 T | 116 t | 148 ö | 180 Ç | 212 å | 244 ¶ |
| 21 NK | 53 5 | 85 U | 117 u | 149 ò | 181 ç | 213 í | 245 § |
| 22 SY | 54 6 | 86 V | 118 v | 150 û | 182 Ñ | 214 ø | 246 - |
| 23 EB | 55 7 | 87 W | 119 w | 151 ù | 183 ñ | 215 æ | 247 ¸ |
| 24 CN | 56 8 | 88 X | 120 x | 152 ÿ | 184 ¤ | 216 Ä | 248 ° |
| 25 EM | 57 9 | 89 Y | 121 y | 153 Ö | 185 ¿ | 217 ì | 249 ¨ |
| 26 SB | 58 : | 90 Z | 122 z | 154 Ü | 186 ¤ | 218 Ö | 250 ¿ |
| 27 EC | 59 ; | 91 [ | 123 { | 155 ø | 187 £ | 219 Ü | 251 « |
| 28 FS | 60 < | 92 \ | 124 \| | 156 £ | 188 ¥ | 220 É | 252 ¨ |
| 29 GS | 61 = | 93 ] | 125 } | 157 Ø | 189 § | 221 Ï | 253 » |
| 30 RS | 62 > | 94 ^ | 126 ~ | 158 × | 190 ƒ | 222 Ì | 254 ± |
| 31 US | 63 ? | 95 _ | 127 ⌘ | 159 ƒ | 191 ç | 223 Ô | 255 ¯ |

**LEXICAL ORDER IS FRENCH (Roman-8)**

| Order | Num | Chr | Order | Num | Chr | Order | Num | Chr | Order | Num | Chr | Order | Num | Chr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 45 | - | | 51 | 52 | 4 | | 81 | 80 | P | | 112 | 107 | k | | 151 | 242 | · |
| 0 | 0 | ᴺ | | 52 | 53 | 5 | | 82 | 81 | Q | | 113 | 108 | l | | 152 | 243 | µ |
| 1 | 1 | ˢ₁ | | 53 | 54 | 6 | | 83 | 82 | R | | 114 | 109 | m | | 153 | 244 | ¶ |
| 2 | 2 | ˢ₂ | | 54 | 55 | 7 | | 84 | 83 | S | | 115 | 110 | n | | 154 | 245 | ¾ |
| 3 | 3 | ˢ₃ | | 55 | 56 | 8 | | 85 | 235 | Š | | 116 | 183 | ñ | | 155 | 246 | - |
| 4 | 4 | ⁴ | | 56 | 57 | 9 | | 86 | 84 | T | | 117 | 111 | o | | 156 | 247 | ¼ |
| 5 | 5 | ᴱ | | 57 | 58 | : | | 87 | 85 | U | | 117 | 194 | ó | | 157 | 248 | ½ |
| 6 | 6 | ᴬ | | 58 | 59 | ; | | 87 | 173 | Ù | | 117 | 198 | ó | | 158 | 249 | ª |
| 7 | 7 | Ⱥ | | 59 | 60 | < | | 87 | 174 | Ú | | 117 | 202 | ò | | 159 | 250 | º |
| 8 | 8 | ᴮ | | 60 | 61 | = | | 87 | 219 | Û | | 117 | 206 | ö | | 160 | 251 | « |
| 9 | 9 | ᴴ | | 61 | 62 | > | | 87 | 237 | Ú | | 117 | 214 | ø | | 161 | 252 | ■ |
| 10 | 10 | ᴸ | | 62 | 63 | ? | | 88 | 86 | V | | 117 | 234 | õ | | 162 | 253 | » |
| 11 | 11 | ᵁ | | 63 | 64 | @ | | 89 | 87 | W | | 118 | 112 | p | | 163 | 254 | ± |
| 12 | 12 | ᴱ | | 64 | 65 | A | | 90 | 88 | X | | 119 | 113 | q | | 164 | 127 | ▒ |
| 13 | 13 | ᶜ | | 64 | 161 | À | | 91 | 89 | Y | | 120 | 114 | r | | 165 | 168 | ¨ |
| 14 | 14 | ᴳ | | 64 | 162 | Â | | 91 | 177 | Ý | | 121 | 115 | s | | 166 | 128 | Ç |
| 15 | 15 | ᴵ | | 64 | 208 | Å | | 91 | 238 | Ÿ | | ss | 222 | ß | | 167 | 129 | ü |
| 16 | 16 | ᴰ | | 64 | 211 | Æ | | 92 | 90 | Z | | 122 | 236 | š | | 168 | 130 | é |
| 17 | 17 | ᴰ | | 64 | 216 | Ä | | 93 | 240 | Þ | | 123 | 116 | t | | 169 | 131 | â |
| 18 | 18 | ᴰ | | 64 | 224 | Á | | 94 | 91 | [ | | 124 | 117 | u | | 170 | 132 | ä |
| 19 | 19 | ᴱ | | 64 | 225 | Ã | | 95 | 92 | \ | | 124 | 195 | û | | 171 | 133 | à |
| 20 | 20 | ᴺ | | 65 | 66 | B | | 96 | 93 | ] | | 124 | 199 | ú | | 172 | 134 | å |
| 21 | 21 | ᴺ | | 66 | 67 | C | | 97 | 94 | ^ | | 124 | 203 | ù | | 173 | 135 | ç |
| 22 | 22 | ˢ | | 66 | 180 | Ç | | 98 | 95 | _ | | 124 | 207 | ü | | 174 | 136 | ê |
| 23 | 23 | ᴱ | | 67 | 68 | D | | 99 | 96 | ` | | 125 | 118 | v | | 175 | 137 | ë |
| 24 | 24 | ˢ | | 68 | 227 | Ð | | 100 | 97 | a | | 126 | 119 | w | | 176 | 138 | è |
| 25 | 25 | ᴱ | | 69 | 69 | E | | 100 | 192 | ã | | 127 | 120 | x | | 177 | 139 | ï |
| 26 | 26 | ˢ | | 69 | 163 | È | | 100 | 196 | á | | 120 | 121 | y | | 179 | 140 | î |
| 27 | 27 | ᴱ | | 69 | 164 | Ê | | 100 | 200 | à | | 128 | 178 | ý | | 179 | 141 | ì |
| 28 | 28 | ᴱ | | 69 | 165 | Ë | | 100 | 204 | ä | | 128 | 239 | ÿ | | 180 | 142 | Ä |
| 29 | 29 | ᴳ | | 69 | 220 | É | | 100 | 212 | å | | 129 | 122 | z | | 181 | 143 | Å |
| 30 | 30 | ᴮ | | 70 | 70 | F | | 100 | 215 | æ | | 130 | 241 | þ | | 182 | 144 | É |
| 31 | 31 | ᵁ | | 71 | 71 | G | | 100 | 226 | ã | | 131 | 123 | { | | 183 | 145 | æ |
| 32 | 32 | | | 72 | 72 | H | | 101 | 98 | b | | 132 | 124 | \| | | 184 | 146 | Æ |
| 33 | 33 | ! | | 73 | 73 | I | | 102 | 99 | c | | 133 | 125 | } | | 185 | 147 | ô |
| 34 | 34 | " | | 73 | 166 | Î | | 103 | 181 | ç | | 134 | 126 | ~ | | 186 | 148 | ö |
| 35 | 35 | # | | 73 | 167 | Ï | | 104 | 100 | d | | 135 | 168 | ´ | | 187 | 149 | ò |
| 36 | 36 | $ | | 73 | 229 | Í | | 105 | 228 | ð | | 136 | 169 | ` | | 188 | 150 | û |
| 37 | 37 | % | | 73 | 230 | Ì | | 106 | 101 | e | | 137 | 170 | ^ | | 189 | 151 | ù |
| 38 | 38 | & | | 74 | 74 | J | | 106 | 193 | ê | | 138 | 171 | ¨ | | 190 | 152 | ÿ |
| 39 | 39 | ' | | 75 | 75 | K | | 106 | 197 | é | | 139 | 172 | ~ | | 191 | 153 | Ö |
| 40 | 40 | ( | | 76 | 76 | L | | 106 | 201 | è | | 140 | 175 | £ | | 192 | 154 | Ü |
| 41 | 41 | ) | | 77 | 77 | M | | 106 | 205 | ë | | 141 | 176 | ¯ | | 193 | 155 | ø |
| 42 | 42 | * | | 78 | 78 | N | | 107 | 102 | f | | 142 | 179 | ° | | 194 | 156 | £ |
| 43 | 43 | + | | 79 | 182 | Ñ | | 108 | 103 | g | | 143 | 184 | ¡ | | 195 | 157 | Ø |
| 44 | 44 | , | | 80 | 79 | O | | 109 | 104 | h | | 144 | 185 | ¿ | | 196 | 158 | × |
| 45 | 46 | . | | 80 | 210 | Ø | | 110 | 105 | i | | 145 | 186 | ¤ | | 197 | 159 | ƒ |
| 46 | 47 | / | | 80 | 210 | Ö | | 110 | 209 | î | | 146 | 107 | £ | | 198 | 255 | ■ |
| 47 | 48 | 0 | | 80 | 223 | Ô | | 110 | 213 | í | | 147 | 188 | ¥ | | | | |
| 48 | 49 | 1 | | 80 | 231 | Ó | | 110 | 217 | ì | | 148 | 189 | ₧ | | | | |
| 49 | 50 | 2 | | 80 | 232 | Ò | | 110 | 221 | ï | | 149 | 190 | ƒ | | | | |
| 50 | 51 | 3 | | 80 | 233 | Õ | | 111 | 106 | j | | 150 | 191 | ¢ | | | | |

**LEXICAL ORDER IS GERMAN (Roman-8)**

| Order | Num | Chr |
|---|---|---|
| 0 | 0 | ␀ |
| 1 | 1 | ☺ |
| 2 | 2 | ☻ |
| 3 | 3 | ♥ |
| 4 | 4 | ♦ |
| 5 | 5 | ♣ |
| 6 | 6 | ♠ |
| 7 | 7 | • |
| 8 | 8 | ◘ |
| 9 | 9 | ○ |
| 10 | 10 | ◙ |
| 11 | 11 | ♂ |
| 12 | 12 | ♀ |
| 13 | 13 | ♪ |
| 14 | 14 | ♫ |
| 15 | 15 | ☼ |
| 16 | 16 | ► |
| 17 | 17 | ◄ |
| 18 | 18 | ↕ |
| 19 | 19 | ‼ |
| 20 | 20 | ¶ |
| 21 | 21 | § |
| 22 | 22 | ▬ |
| 23 | 23 | ↨ |
| 24 | 24 | ↑ |
| 25 | 25 | ↓ |
| 26 | 26 | → |
| 27 | 27 | ← |
| 28 | 28 | ∟ |
| 29 | 29 | ↔ |
| 30 | 30 | ▲ |
| 31 | 31 | ▼ |
| 32 | 32 | |
| 33 | 33 | ! |
| 34 | 34 | " |
| 35 | 35 | # |
| 36 | 36 | $ |
| 37 | 37 | % |
| 38 | 38 | & |
| 39 | 39 | ' |
| 40 | 40 | ( |
| 41 | 41 | ) |
| 42 | 42 | * |
| 43 | 43 | + |
| 44 | 44 | , |
| 45 | 45 | - |
| 46 | 46 | . |
| 47 | 47 | / |
| 48 | 48 | 0 |
| 49 | 49 | 1 |
| 50 | 50 | 2 |
| 51 | 51 | 3 |
| 52 | 52 | 4 |
| 53 | 53 | 5 |
| 54 | 54 | 6 |
| 55 | 55 | 7 |
| 56 | 56 | 8 |
| 57 | 57 | 9 |
| 58 | 58 | : |
| 59 | 59 | ; |
| 60 | 60 | < |
| 61 | 61 | = |
| 62 | 62 | > |
| 63 | 63 | ? |
| 64 | 64 | @ |
| 65 | 65 | A |
| AE/Ae | 216 | Ä |
| 66 | 211 | Æ |
| 67 | 208 | Å |
| 68 | 224 | Á |
| 69 | 161 | À |
| 70 | 162 | Â |
| 71 | 225 | Ã |
| 72 | 66 | B |
| 73 | 67 | C |
| 74 | 180 | Ç |
| 75 | 68 | D |
| 76 | 227 | Đ |
| 77 | 69 | E |
| 78 | 220 | É |
| 79 | 163 | È |
| 80 | 164 | Ë |
| 81 | 165 | Ê |
| 82 | 70 | F |
| 83 | 71 | G |
| 84 | 72 | H |
| 85 | 73 | I |
| 86 | 222 | Í |
| 87 | 230 | Ì |
| 88 | 166 | Ï |
| 89 | 167 | Î |
| 90 | 74 | J |
| 91 | 75 | K |
| 92 | 76 | L |
| 93 | 77 | M |
| 94 | 78 | N |
| 95 | 182 | Ñ |
| 96 | 79 | O |
| OE/Oe | 218 | Ö |
| 97 | 231 | Ó |
| 98 | 232 | Ò |
| 99 | 223 | Ô |
| 100 | 233 | Õ |
| 101 | 210 | Ø |
| 102 | 80 | P |
| 103 | 81 | Q |
| 104 | 82 | R |
| 105 | 83 | S |
| 106 | 235 | Š |
| 107 | 84 | T |
| 108 | 85 | U |
| UE/Ue | 219 | Ü |
| 109 | 237 | Ú |
| 110 | 173 | Ù |
| 111 | 174 | Û |
| 112 | 86 | V |
| 113 | 87 | W |
| 114 | 88 | X |
| 115 | 89 | Y |
| 116 | 238 | Ý |
| 117 | 177 | Ÿ |
| 118 | 90 | Z |
| 119 | 240 | Þ |
| 120 | 91 | [ |
| 121 | 92 | \ |
| 122 | 93 | ] |
| 123 | 94 | ^ |
| 124 | 95 | _ |
| 125 | 96 | ` |
| 126 | 97 | a |
| ae | 204 | ä |
| 127 | 215 | æ |
| 128 | 212 | å |
| 129 | 196 | á |
| 130 | 200 | à |
| 131 | 192 | â |
| 132 | 226 | ã |
| 133 | 90 | ð |
| 134 | 99 | c |
| 135 | 181 | ç |
| 136 | 100 | d |
| 137 | 228 | ð |
| 138 | 101 | e |
| 139 | 197 | é |
| 140 | 201 | è |
| 141 | 202 | ë |
| 142 | 205 | ê |
| 143 | 102 | f |
| 144 | 103 | g |
| 145 | 104 | h |
| 146 | 105 | i |
| 147 | 213 | í |
| 148 | 217 | ì |
| 149 | 209 | ï |
| 150 | 221 | î |
| 151 | 106 | j |
| 152 | 107 | k |
| 153 | 108 | l |
| 154 | 109 | m |
| 155 | 110 | n |
| 156 | 183 | ñ |
| 157 | 111 | o |
| oe | 206 | ö |
| 158 | 198 | ó |
| 159 | 202 | ò |
| 160 | 194 | ô |
| 161 | 204 | õ |
| 162 | 214 | ø |
| 163 | 112 | p |
| 164 | 113 | q |
| 165 | 114 | r |
| 166 | 115 | s |
| ss | 222 | ß |
| 167 | 206 | š |
| 168 | 116 | t |
| 169 | 117 | u |
| ue | 207 | ü |
| 170 | 199 | ú |
| 171 | 203 | ù |
| 172 | 195 | û |
| 173 | 118 | v |
| 174 | 119 | w |
| 175 | 120 | x |
| 176 | 121 | y |
| 177 | 208 | ÿ |
| 178 | 178 | ý |
| 179 | 122 | z |
| 180 | 241 | þ |
| 181 | 123 | { |
| 182 | 124 | \| |
| 183 | 125 | } |
| 184 | 126 | ~ |
| 185 | 168 | ´ |
| 186 | 169 | ¯ |
| 187 | 170 | ¨ |
| 188 | 171 | ¸ |
| 189 | 172 | ˜ |
| 190 | 175 | ¿ |
| 191 | 176 | ¯ |
| 192 | 179 | ° |
| 193 | 184 | ¡ |
| 194 | 185 | ¿ |
| 195 | 186 | ¤ |
| 196 | 187 | £ |
| 197 | 188 | ¥ |
| 198 | 189 | § |
| 199 | 190 | ƒ |
| 200 | 191 | ¢ |
| 201 | 242 | - |
| 202 | 243 | µ |
| 203 | 244 | ¶ |
| 204 | 245 | · |
| 205 | 246 | - |
| 206 | 247 | ¸ |
| 207 | 248 | ¹ |
| 208 | 249 | º |
| 209 | 250 | » |
| 210 | 251 | ¼ |
| 211 | 252 | ½ |
| 212 | 253 | ¾ |
| 213 | 254 | ¿ |
| 214 | 127 | ⌂ |
| 215 | 160 | Ç |
| 216 | 128 | ü |
| 217 | 129 | é |
| 218 | 130 | â |
| 219 | 131 | ä |
| 220 | 132 | à |
| 221 | 133 | å |
| 222 | 134 | ç |
| 223 | 135 | ê |
| 224 | 136 | ë |
| 225 | 137 | è |
| 226 | 138 | ï |
| 227 | 139 | î |
| 228 | 140 | ì |
| 229 | 141 | Ä |
| 230 | 142 | Å |
| 231 | 143 | É |
| 232 | 144 | æ |
| 233 | 145 | Æ |
| 234 | 146 | ô |
| 235 | 147 | ö |
| 236 | 148 | ò |
| 237 | 149 | û |
| 238 | 150 | ù |
| 239 | 151 | ÿ |
| 240 | 152 | Ö |
| 241 | 153 | Ü |
| 242 | 154 | ¢ |
| 243 | 155 | £ |
| 244 | 156 | ¥ |
| 245 | 157 | ₧ |
| 246 | 158 | ƒ |
| 247 | 159 | á |
| 248 | 255 | í |

**LEXICAL ORDER IS SPANISH (Roman-8)**

| Order | Num | Chr | Order | Num | Chr | Order | Num | Chr | Order | Num | Chr | Order | Num | Chr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | ' | 52 | 52 | 4 | 83 | 223 | Ô | 113 | 209 | f | 149 | 186 | ª |
| 1 | 1 | ' | 53 | 53 | 5 | 83 | 231 | ó | 113 | 213 | f | 150 | 187 | ' |
| 2 | 2 | ' | 54 | 54 | 6 | 83 | 232 | ò | 113 | 217 | i | 151 | 188 | ¥ |
| 3 | 3 | ' | 55 | 55 | 7 | 83 | 233 | Õ | 113 | 221 | Y | 152 | 189 | § |
| 4 | 4 | ' | 56 | 56 | 8 | 84 | 88 | P | 114 | 106 | j | 153 | 190 | ƒ |
| 5 | 5 | ' | 57 | 57 | 9 | 85 | 81 | Q | 115 | 107 | k | 154 | 191 | ç |
| 6 | 6 | ' | 58 | 58 | : | 86 | 82 | R | 116 | 108 | l | 155 | 246 | - |
| 7 | 7 | ' | 59 | 59 | ; | 87 | 83 | S | 117 | | ll | 156 | 247 | ' |
| 8 | 8 | ' | 60 | 60 | < | 88 | 235 | Š | 117 | | lL | 157 | 248 | ½ |
| 9 | 9 | ' | 61 | 61 | = | 89 | 84 | T | 118 | 109 | m | 158 | 249 | ª |
| 10 | 10 | ' | 62 | 62 | > | 90 | 85 | U | 119 | 110 | n | 159 | 250 | º |
| 11 | 11 | ' | 63 | 63 | ? | 90 | 173 | Ù | 120 | 183 | ñ | 160 | 251 | « |
| 12 | 12 | ' | 64 | 64 | @ | 90 | 174 | Ü | 121 | 111 | o | 161 | 252 | ■ |
| 13 | 13 | ' | 65 | 65 | A | 90 | 219 | Ü | 121 | 194 | ô | 162 | 253 | » |
| 14 | 14 | ' | 65 | 161 | À | 90 | 237 | Ú | 121 | 198 | ó | 163 | 254 | ± |
| 15 | 15 | ' | 65 | 162 | Â | 91 | 86 | V | 121 | 202 | ò | 164 | 127 | ' |
| 16 | 16 | ' | 65 | 208 | Å | 92 | 87 | W | 121 | 206 | õ | 165 | 160 | ' |
| 17 | 17 | ' | 65 | 211 | Æ | 93 | 88 | X | 121 | 214 | ø | 166 | 177 | ' |
| 18 | 18 | ' | 65 | 216 | ñ | 94 | 89 | Y | 121 | 234 | õ | 167 | 178 | ' |
| 19 | 19 | ' | 65 | 224 | á | 94 | 200 | Ÿ | 122 | 112 | p | 168 | 242 | · |
| 20 | 20 | ' | 65 | 225 | Ã | 95 | 90 | Z | 123 | 113 | q | 169 | 243 | µ |
| 21 | 21 | ' | 66 | 66 | B | 96 | 240 | Þ | 124 | 114 | r | 170 | 244 | ¶ |
| 22 | 22 | ' | 67 | 67 | C | 97 | 91 | [ | 125 | 115 | s | 171 | 245 | ' |
| 23 | 23 | ' | 67 | 180 | Ç | 98 | 92 | \ | ss | ZZZ | ß | 172 | 128 | Ç |
| 24 | 24 | ' | 68 | | Ch | 99 | 93 | ] | 126 | 236 | š | 173 | 129 | ' |
| 25 | 25 | ' | 68 | | CH | 100 | 94 | ^ | 127 | 116 | t | 174 | 130 | ' |
| 26 | 26 | ' | 69 | 68 | D | 101 | 95 | _ | 128 | 117 | u | 175 | 131 | ' |
| 27 | 27 | ' | 70 | 227 | Ð | 102 | 96 | ' | 128 | 195 | û | 176 | 132 | ' |
| 28 | 28 | ' | 71 | 69 | E | 103 | 97 | a | 128 | 199 | ú | 177 | 133 | ' |
| 29 | 29 | ' | 71 | 163 | È | 103 | 192 | à | 128 | 203 | ù | 178 | 134 | ' |
| 30 | 30 | ' | 71 | 164 | Ê | 103 | 196 | á | 128 | 207 | ũ | 179 | 135 | ' |
| 31 | 31 | ' | 71 | 165 | Ë | 103 | 200 | à | 129 | 118 | v | 180 | 136 | ' |
| 32 | 32 | | 71 | 220 | É | 103 | 201 | ã | 130 | 119 | w | 181 | 137 | ' |
| 33 | 33 | ! | 72 | 70 | F | 103 | 212 | å | 131 | 120 | x | 182 | 130 | ' |
| 34 | 34 | " | 73 | 71 | G | 103 | 215 | æ | 132 | 121 | y | 183 | 139 | ' |
| 35 | 35 | ■ | 74 | 72 | H | 103 | 226 | ã | 132 | 239 | ÿ | 184 | 140 | ' |
| 36 | 36 | ' | 75 | 73 | I | 104 | 98 | b | 133 | 122 | z | 185 | 141 | ' |
| 37 | 37 | ' | 75 | 166 | Î | 105 | 99 | c | 134 | 241 | þ | 186 | 142 | ' |
| 38 | 38 | ' | 75 | 167 | Ï | 105 | 181 | ç | 135 | 123 | { | 187 | 143 | ' |
| 39 | 39 | ' | 75 | 229 | Í | 106 | | ch | 136 | 124 | \| | 188 | 144 | ' |
| 40 | 40 | ( | 75 | 230 | Ì | 106 | | cH | 137 | 125 | } | 189 | 145 | ' |
| 41 | 41 | ) | 76 | 74 | J | 107 | 100 | d | 138 | 126 | ~ | 190 | 146 | ' |
| 42 | 42 | * | 77 | 75 | K | 108 | 228 | ð | 139 | 168 | ´ | 191 | 147 | ' |
| 43 | 43 | + | 78 | 76 | L | 109 | 101 | e | 140 | 169 | ` | 192 | 148 | ' |
| 44 | 44 | , | 79 | | Ll | 109 | 193 | é | 141 | 170 | ^ | 193 | 149 | ' |
| 45 | 45 | - | 79 | | LL | 109 | 197 | é | 142 | 171 | ¯ | 194 | 150 | ' |
| 46 | 46 | . | 80 | 77 | M | 109 | 201 | è | 143 | 172 | ~ | 195 | 151 | ' |
| 47 | 47 | / | 81 | 70 | N | 109 | 205 | ë | 144 | 175 | £ | 196 | 152 | ' |
| 48 | 48 | 0 | 82 | 182 | Ñ | 110 | 102 | f | 145 | 176 | ¯ | 197 | 153 | ' |
| 49 | 49 | 1 | 83 | 79 | O | 111 | 103 | g | 146 | 179 | ª | 198 | 154 | ' |
| 50 | 50 | 2 | 83 | 210 | Ø | 112 | 104 | h | 147 | 180 | ¡ | 199 | 155 | ' |
| 51 | 51 | 3 | 83 | 218 | Õ | 113 | 105 | i | 148 | 185 | ¿ | 200 | 156 | ' |

**LEXICAL ORDER IS SWEDISH (Roman-8)**

| Order | Num | Chr | Order | Num | Chr | Order | Num | Chr | Order | Num | Chr | Order | Num | Chr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | ␀ | 52 | 52 | 4 | 101 | 229 | í | 154 | 122 | z | 206 | 242 | ' |
| 1 | 1 | ␁ | 53 | 53 | 5 | 105 | 238 | ì | 155 | 215 | × | 207 | 243 | μ |
| 2 | 2 | ␂ | 54 | 54 | 6 | 106 | 166 | ĵ | 156 | 212 | Ô | 208 | 244 | ¶ |
| 3 | 3 | ␃ | 55 | 55 | 7 | 107 | 167 | ĵ | 157 | 196 | Ä | 209 | 245 | · |
| 4 | 4 | ␄ | 56 | 56 | 8 | 108 | 182 | Ĥ | 158 | 208 | Ö | 210 | 246 | - |
| 5 | 5 | ␅ | 57 | 57 | 9 | 109 | 231 | ĝ | 159 | 192 | Ü | 211 | 247 | ' |
| 6 | 6 | ␆ | 58 | 58 | : | 110 | 232 | ù | 160 | 201 | ß | 212 | 248 | ¸ |
| 7 | 7 | ␇ | 59 | 59 | ; | 111 | 223 | ĝ | 161 | 226 | à | 213 | 249 | ą |
| 8 | 8 | ␈ | 60 | 60 | < | 112 | 218 | ŭ | 162 | 181 | ç | 214 | 250 | ş |
| 9 | 9 | ␉ | 61 | 61 | = | 113 | 233 | ŭ | 163 | 228 | ê | 215 | 251 | Ć |
| 10 | 10 | ␊ | 62 | 62 | > | 114 | 218 | ŝ | 164 | 201 | è | 216 | 252 | ■ |
| 11 | 11 | ␋ | 63 | 63 | ? | 115 | 235 | ŝ | 165 | 193 | é | 217 | 253 | ˝ |
| 12 | 12 | ␌ | 64 | 64 | @ | 116 | 237 | ŭ | 166 | 205 | ĉ | 218 | 254 | ţ |
| 13 | 13 | ␍ | 65 | 65 | A | 117 | 173 | ŭ | 167 | 213 | ĩ | 219 | 127 | ˇ |
| 14 | 14 | ␎ | 66 | 66 | B | 118 | 174 | ŭ | 168 | 217 | ì | 220 | 160 | Ŕ |
| 15 | 15 | ␏ | 67 | 67 | C | 119 | 219 | ŭ | 169 | 209 | ĵ | 221 | 128 | Ł |
| 16 | 16 | ␐ | 68 | 68 | D | 120 | 177 | ŷ | 170 | 221 | ì | 222 | 129 | ś |
| 17 | 17 | ␑ | 69 | 69 | E | 121 | 238 | ŷ | 171 | 183 | ĥ | 223 | 130 | Ś |
| 18 | 18 | ␒ | 70 | 70 | F | 122 | 248 | Ĵ | 172 | 198 | ó | 224 | 131 | Ś |
| 19 | 19 | ␓ | 71 | 71 | G | 123 | 91 | [ | 173 | 202 | ù | 225 | 132 | Ł |
| 20 | 20 | ␔ | 72 | 72 | H | 124 | 92 | \ | 174 | 191 | ó | 226 | 133 | ś |
| 21 | 21 | ␕ | 73 | 73 | I | 125 | 93 | ] | 175 | 206 | ú | 227 | 134 | Ś |
| 22 | 22 | ␖ | 74 | 74 | J | 126 | 94 | ^ | 176 | 231 | ô | 228 | 135 | ś |
| 23 | 23 | ␗ | 75 | 75 | K | 127 | 95 | _ | 177 | 214 | ø | 229 | 136 | Ń |
| 24 | 24 | ␘ | 76 | 76 | L | 128 | 96 | ` | 178 | 236 | ă | 230 | 137 | Ń |
| 25 | 25 | ␙ | 77 | 77 | M | 129 | 97 | a | 179 | 199 | ô | 231 | 138 | ł |
| 26 | 26 | ␚ | 78 | 78 | N | 130 | 98 | b | 180 | 210 | ñ | 232 | 139 | ś |
| 27 | 27 | ␛ | 79 | 79 | O | 131 | 99 | c | 181 | 195 | ô | 233 | 140 | Ś |
| 28 | 28 | ␜ | 80 | 80 | P | 132 | 100 | d | 182 | 227 | ñ | 234 | 141 | Ť |
| 29 | 29 | ␝ | 81 | 81 | Q | 133 | 101 | e | 183 | 178 | ģ | 235 | 142 | Ž |
| 30 | 30 | ␞ | 82 | 82 | R | 134 | 102 | f | 184 | 200 | ģ | 236 | 143 | Ť |
| 31 | 31 | ␟ | 83 | 83 | S | 135 | 103 | g | 185 | 241 | þ | 237 | 144 | Ź |
| 32 | 32 |  | 84 | 84 | T | 136 | 104 | h | 186 | 120 | ł | 238 | 145 | ' |
| 33 | 33 | ! | 85 | 85 | U | 137 | 105 | i | 187 | 124 | ł | 239 | 146 | Ź |
| 34 | 34 | " | 86 | 86 | V | 138 | 106 | j | 188 | 126 | ~ | 240 | 147 | ˝ |
| 35 | 35 | # | 87 | 87 | W | 139 | 107 | k | 189 | 161 | ˝ | 241 | 148 | Ż |
| 36 | 36 | $ | 88 | 88 | X | 140 | 108 | l | 190 | 169 | ˛ | 242 | 149 | ˝ |
| 37 | 37 | % | 89 | 89 | Y | 141 | 109 | m | 191 | 169 | ˛ | 243 | 150 | Ż |
| 38 | 38 | & | 90 | 90 | Z | 142 | 110 | n | 192 | 170 | ˇ | 244 | 151 | ˝ |
| 39 | 39 | ' | 91 | 91 | Æ | 143 | 111 | o | 193 | 171 | ¯ | 245 | 152 | Ż |
| 40 | 40 | ( | 92 | 92 | Å | 144 | 112 | p | 194 | 172 | ˘ | 246 | 153 | ˛ |
| 41 | 41 | ) | 93 | 224 | Á | 145 | 113 | q | 195 | 175 | £ | 247 | 154 | Ż |
| 42 | 42 | * | 94 | 161 | À | 146 | 114 | r | 196 | 176 |  | 248 | 155 | ˛ |
| 43 | 43 | + | 95 | 162 | Ã | 147 | 115 | s | 197 | 179 | ˝ | 249 | 156 | Ż |
| 44 | 44 | , | 96 | 163 | Ã | 148 | 116 | t | 198 | 180 | ˙ | 250 | 157 | ˛ |
| 45 | 45 | - | 97 | 225 | Ñ | 149 | 117 | u | 199 | 185 | ą | 251 | 158 | Ż |
| 46 | 46 | . | 98 | 164 | Ç | 150 | 118 | v | 200 | 186 | ł | 252 | 159 | ˛ |
| 47 | 47 | / | 99 | 227 | Ð | 151 | 119 | w | 201 | 187 | Ľ | 253 | 255 | ˙ |
| 48 | 48 | 0 | 100 | 228 | Ê | 152 | 120 | x | 202 | 188 | ľ |  |  |  |
| 49 | 49 | 1 | 101 | 163 | Ë | 153 | 121 | y | 203 | 189 | Ś |  |  |  |
| 50 | 50 | 2 | 102 | 164 | È | 154 | 122 | z | 204 | 190 | ś |  |  |  |
| 51 | 51 | 3 | 103 | 165 | Ë | 155 | 123 | { | 205 | 191 | ŕ |  |  |  |

**Latin-1 Character Set**

Note in the table below, CHR$(145) and CHR$(146) are extensions to Latin-1 found in Windows fonts and may not be present in other implementations of Latin-1.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 ■ | 32 | 64 @ | 96 ` | 128 ■ | 160 | 192 À | 224 à |
| 1 ■ | 33 ! | 65 A | 97 a | 129 ■ | 161 ¡ | 193 Á | 225 á |
| 2 ■ | 34 " | 66 B | 98 b | 130 ■ | 162 ¢ | 194 Â | 226 â |
| 3 ■ | 35 # | 67 C | 99 c | 131 ■ | 163 £ | 195 Ã | 227 ã |
| 4 ■ | 36 $ | 68 D | 100 d | 132 ■ | 164 ¤ | 196 Ä | 228 ä |
| 5 ■ | 37 % | 69 E | 101 e | 133 ■ | 165 ¥ | 197 Å | 229 å |
| 6 ■ | 38 & | 70 F | 102 f | 134 ■ | 166 ¦ | 198 Æ | 230 æ |
| 7 ■ | 39 ' | 71 G | 103 g | 135 ■ | 167 § | 199 Ç | 231 ç |
| 8 ■ | 40 ( | 72 H | 104 h | 136 ■ | 168 ¨ | 200 È | 232 è |
| 9 ■ | 41 ) | 73 I | 105 i | 137 ■ | 169 © | 201 É | 233 é |
| 10 ■ | 42 * | 74 J | 106 j | 138 ■ | 170 ª | 202 Ê | 234 ê |
| 11 ■ | 43 + | 75 K | 107 k | 139 ■ | 171 « | 203 Ë | 235 ë |
| 12 ■ | 44 , | 76 L | 108 l | 140 ■ | 172 ¬ | 204 Ì | 236 ì |
| 13 ■ | 45 - | 77 M | 109 m | 141 ■ | 173 - | 205 Í | 237 í |
| 14 ■ | 46 . | 78 N | 110 n | 142 ■ | 174 ® | 206 Î | 238 î |
| 15 ■ | 47 / | 79 O | 111 o | 143 ■ | 175 ¯ | 207 Ï | 239 ï |
| 16 ■ | 48 0 | 80 P | 112 p | 144 ■ | 176 ° | 208 Ð | 240 ð |
| 17 ■ | 49 1 | 81 Q | 113 q | 145 ‘ | 177 ± | 209 Ñ | 241 ñ |
| 18 ■ | 50 2 | 82 R | 114 r | 146 ’ | 178 ² | 210 Ò | 242 ò |
| 19 ■ | 51 3 | 83 S | 115 s | 147 ■ | 179 ³ | 211 Ó | 243 ó |
| 20 ■ | 52 4 | 84 T | 116 t | 148 ■ | 180 ´ | 212 Ô | 244 ô |
| 21 ■ | 53 5 | 85 U | 117 u | 149 ■ | 181 µ | 213 Õ | 245 õ |
| 22 ■ | 54 6 | 86 V | 118 v | 150 ■ | 182 ¶ | 214 Ö | 246 ö |
| 23 ■ | 55 7 | 87 W | 119 w | 151 ■ | 183 · | 215 × | 247 ÷ |
| 24 ■ | 56 8 | 88 X | 120 x | 152 ■ | 184 ¸ | 216 Ø | 248 ø |
| 25 ■ | 57 9 | 89 Y | 121 y | 153 ■ | 185 ¹ | 217 Ù | 249 ù |
| 26 ■ | 58 : | 90 Z | 122 z | 154 ■ | 186 º | 218 Ú | 250 ú |
| 27 ■ | 59 ; | 91 [ | 123 { | 155 ■ | 187 » | 219 Û | 251 û |
| 28 ■ | 60 < | 92 \ | 124 \| | 156 ■ | 188 ¼ | 220 Ü | 252 ü |
| 29 ■ | 61 = | 93 ] | 125 } | 157 ■ | 189 ½ | 221 Ý | 253 ý |
| 30 ■ | 62 > | 94 ^ | 126 ~ | 158 ■ | 190 ¾ | 222 Þ | 254 þ |
| 31 ■ | 63 ? | 95 _ | 127 ■ | 159 ■ | 191 ¿ | 223 ß | 255 ÿ |

**LEXICAL ORDER IS FRENCH (Latin-1)**

| Order | Num | Chr | Order | Num | Chr | Order | Num | Chr | Order | Num | Chr | Order | Num | Chr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 45 | - | | 51 | 52 | 4 | | 81 | 80 | P | | 113 | 109 | m | | 152 | 171 | « |
| | | | | | | | | | | | | | | | | | | |
| 1 | 1 | | | 53 | 54 | 6 | | 83 | 82 | R | | 115 | 241 | ñ | | 154 | 177 | ± |
| 2 | 2 | | | 54 | 55 | 7 | | 84 | 83 | S | | 116 | 111 | o | | 155 | 127 | |
| 3 | 3 | | | 55 | 56 | 8 | | 85 | 84 | T | | 116 | 242 | ò | | 156 | 178 | |
| 4 | 4 | | | 56 | 57 | 9 | | 86 | 85 | U | | 116 | 243 | ó | | 157 | 129 | |
| 5 | 5 | | | 57 | 58 | : | | 86 | 217 | Ù | | 116 | 244 | ô | | 158 | 130 | |
| 6 | 6 | | | 58 | 59 | ; | | 86 | 218 | Ú | | 116 | 245 | õ | | 159 | 131 | |
| 7 | 7 | | | | | | | 86 | 219 | Û | | 116 | 246 | ö | | 160 | 132 | |
| 8 | 8 | | | 60 | 61 | = | | 86 | 220 | Ü | | 116 | 248 | ø | | 161 | 133 | |
| 9 | 9 | | | 61 | 62 | > | | 87 | 86 | V | | 117 | 112 | p | | 162 | 134 | |
| 10 | 10 | | | 62 | 63 | ? | | 88 | 87 | W | | 118 | 113 | q | | 163 | 135 | |
| 11 | 11 | | | 63 | 64 | @ | | 89 | 88 | X | | 119 | 114 | r | | 164 | 136 | |
| 12 | 12 | | | 64 | 65 | A | | 90 | 89 | Y | | 120 | 115 | s | | 165 | 137 | |
| 13 | 13 | | | 64 | 192 | À | | 90 | 221 | Ý | | ss | 223 | ß | | 166 | 138 | |
| 14 | 14 | | | 64 | 193 | Á | | 91 | 90 | Z | | 121 | 116 | t | | 167 | 139 | |
| 15 | 15 | | | 64 | 194 | Â | | 92 | 222 | Þ | | 122 | 117 | u | | 168 | 140 | |
| 16 | 16 | | | 64 | 195 | Ã | | 93 | 91 | [ | | 122 | 249 | ù | | 169 | 141 | |
| 17 | 17 | | | 64 | 196 | Ä | | 94 | 92 | \ | | 122 | 250 | ú | | 170 | 142 | |
| 18 | 18 | | | 64 | 197 | Å | | 95 | 93 | ] | | 122 | 251 | û | | 171 | 143 | |
| 19 | 19 | | | 64 | 198 | Æ | | 96 | 94 | ^ | | 122 | 252 | ü | | 172 | 144 | |
| 20 | 20 | | | 65 | 66 | B | | 97 | 95 | _ | | 123 | 118 | v | | 173 | 145 | ' |
| 21 | 21 | | | 66 | 67 | C | | 98 | 96 | ` | | 124 | 119 | w | | 174 | 146 | ' |
| 22 | 22 | | | 66 | 199 | Ç | | 99 | 97 | a | | 125 | 120 | x | | 175 | 147 | |
| 23 | 23 | | | 67 | 68 | D | | 99 | 224 | à | | 126 | 121 | y | | 176 | 148 | |
| 24 | 24 | | | 68 | 208 | Ð | | 99 | 225 | á | | 126 | 253 | ý | | 177 | 149 | |
| 25 | 25 | | | 69 | 69 | E | | 99 | 226 | â | | 126 | 255 | ÿ | | 178 | 150 | |
| 26 | 26 | | | 69 | 200 | È | | 99 | 227 | ã | | 127 | 122 | z | | 179 | 151 | |
| 27 | 27 | | | 69 | 201 | É | | 99 | 228 | ä | | 128 | 254 | þ | | 180 | 152 | |
| 28 | 28 | | | 69 | 202 | Ê | | 99 | 229 | å | | 129 | 123 | { | | 181 | 153 | |
| 29 | 29 | | | 69 | 203 | Ë | | 99 | 230 | æ | | 130 | 124 | \| | | 182 | 154 | |
| 30 | 30 | | | 70 | 70 | F | | 100 | 98 | b | | 131 | 125 | } | | 183 | 155 | |
| 31 | 31 | | | 71 | 71 | G | | 101 | 99 | c | | 132 | 126 | ~ | | 184 | 156 | |
| 32 | 32 | | | 72 | 72 | H | | 102 | 231 | ç | | 133 | 180 | ´ | | 185 | 157 | |
| 33 | 33 | ! | | 73 | 73 | I | | 103 | 100 | d | | 134 | 168 | ¨ | | 186 | 158 | |
| 34 | 34 | " | | 73 | 204 | Ì | | 104 | 240 | ð | | 135 | 175 | ¯ | | 187 | 159 | |
| 35 | 35 | # | | 73 | 205 | Í | | 105 | 101 | e | | 136 | 176 | ° | | 188 | 160 | |
| 36 | 36 | $ | | 73 | 206 | Î | | 105 | 232 | è | | 137 | 161 | ¡ | | 189 | 166 | ¦ |
| 37 | 37 | % | | 73 | 207 | Ï | | 105 | 233 | é | | 138 | 191 | ¿ | | 190 | 169 | © |
| 38 | 38 | & | | 74 | 74 | J | | 105 | 234 | ê | | 139 | 164 | ¤ | | 191 | 172 | ¬ |
| 39 | 39 | ' | | 75 | 75 | K | | 105 | 235 | ë | | 140 | 163 | £ | | 192 | 170 | ª |
| 40 | 40 | ( | | 76 | 76 | L | | 106 | 102 | f | | 141 | 165 | ¥ | | 193 | 174 | ® |
| 41 | 41 | ) | | 77 | 77 | M | | 107 | 103 | g | | 142 | 167 | § | | 194 | 178 | ² |
| 42 | 42 | * | | 78 | 78 | N | | 108 | 104 | h | | 143 | 162 | ¢ | | 195 | 179 | ³ |
| 43 | 43 | + | | 79 | 209 | Ñ | | 109 | 105 | i | | 144 | 181 | µ | | 196 | 184 | ¸ |
| 44 | 44 | , | | 80 | 79 | O | | 109 | 236 | ì | | 145 | 182 | ¶ | | 197 | 185 | ¹ |
| 45 | 45 | - | | 80 | 210 | Ò | | 109 | 237 | í | | 146 | 183 | · | | 198 | 215 | × |
| 46 | 47 | / | | 80 | 211 | Ó | | 109 | 238 | î | | 147 | 190 | ¾ | | 199 | 247 | ÷ |
| 47 | 48 | 0 | | 80 | 212 | Ô | | 109 | 239 | ï | | 148 | 188 | ¼ | | | | |
| 48 | 49 | 1 | | 80 | 213 | Õ | | 110 | 106 | j | | 149 | 189 | ½ | | | | |
| 49 | 50 | 2 | | 80 | 214 | Ö | | 111 | 107 | k | | 150 | 170 | ª | | | | |
| 50 | 51 | 3 | | 80 | 216 | Ø | | 112 | 108 | l | | 151 | 186 | º | | | | |

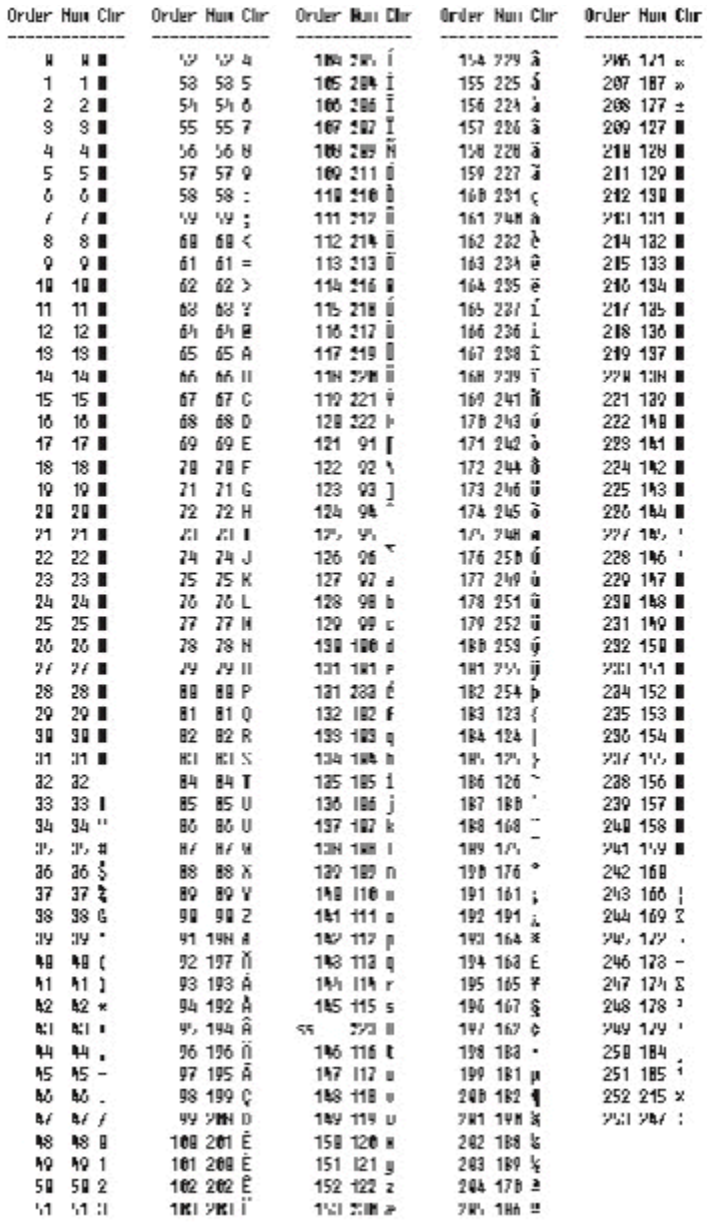
**LEXICAL ORDER IS GERMAN (Latin-1)**

| Order | Num | Chr |
|---|---|---|
| 0 | 0 | ■ |
| 1 | 1 | ■ |
| 2 | 2 | ■ |
| 3 | 3 | ■ |
| 4 | 4 | ■ |
| 5 | 5 | ■ |
| 6 | 6 | ■ |
| 7 | 7 | ■ |
| 8 | 8 | ■ |
| 9 | 9 | ■ |
| 10 | 10 | ■ |
| 11 | 11 | ■ |
| 12 | 12 | ■ |
| 13 | 13 | ■ |
| 14 | 14 | ■ |
| 15 | 15 | ■ |
| 16 | 16 | ■ |
| 17 | 17 | ■ |
| 18 | 18 | ■ |
| 19 | 19 | ■ |
| 20 | 20 | ■ |
| 21 | 21 | ■ |
| 22 | 22 | ■ |
| 23 | 23 | ■ |
| 24 | 24 | ■ |
| 25 | 25 | ■ |
| 26 | 26 | ■ |
| 27 | 27 | ■ |
| 28 | 28 | ■ |
| 29 | 29 | ■ |
| 30 | 30 | ■ |
| 31 | 31 | ■ |
| 32 | 32 |  |
| 33 | 33 | ! |
| 34 | 34 | " |
| 35 | 35 | # |
| 36 | 36 | $ |
| 37 | 37 | % |
| 38 | 38 | & |
| 39 | 39 | ' |
| 40 | 40 | ( |
| 41 | 41 | ) |
| 42 | 42 | * |
| 43 | 43 | + |
| 44 | 44 | , |
| 45 | 45 | - |
| 46 | 46 | . |
| 47 | 47 | / |
| 48 | 48 | 0 |
| 49 | 49 | 1 |
| 50 | 50 | 2 |
| 51 | 51 | 3 |
| 52 | 52 | 4 |
| 53 | 53 | 5 |
| 54 | 54 | 6 |
| 55 | 55 | 7 |
| 56 | 56 | 8 |
| 57 | 57 | 9 |
| 58 | 58 | : |
| 59 | 59 | ; |
| 60 | 60 | < |
| 61 | 61 | = |
| 62 | 62 | > |
| 63 | 63 | ? |
| 64 | 64 | @ |
| 65 | 65 | A |
| AE/Ae | 196 | Ä |
| 66 | 198 | Æ |
| 67 | 197 | Å |
| 68 | 193 | Á |
| 69 | 192 | À |
| 70 | 194 | Â |
| 71 | 195 | Ã |
| 72 | 66 | B |
| 73 | 67 | C |
| 74 | 199 | Ç |
| 75 | 68 | D |
| 76 | 208 | Ð |
| 77 | 69 | E |
| 78 | 201 | É |
| 79 | 200 | È |
| 80 | 202 | Ê |
| 81 | 203 | Ë |
| 82 | 70 | F |
| 83 | 71 | G |
| 84 | 72 | H |
| 85 | 73 | I |
| 86 | 205 | Í |
| 87 | 204 | Ì |
| 88 | 206 | Î |
| 89 | 207 | Ï |
| 90 | 74 | J |
| 91 | 75 | K |
| 92 | 76 | L |
| 93 | 77 | M |
| 94 | 78 | N |
| 95 | 209 | Ñ |
| 96 | 79 | O |
| OE/Oe | 214 | Ö |
| 97 | 211 | Ó |
| 98 | 210 | Ò |
| 99 | 212 | Ô |
| 100 | 213 | Õ |
| 101 | 216 | Ø |
| 102 | 80 | P |
| 103 | 81 | Q |
| 104 | 82 | R |
| 105 | 83 | S |
| 106 | 84 | T |
| UE/Ue | 220 | Ü |
| 107 | 218 | Ú |
| 108 | 217 | Ù |
| 109 | 219 | Û |
| 110 | 85 | U |
| 111 | 86 | V |
| 112 | 87 | W |
| 113 | 88 | X |
| 114 | 89 | Y |
| 115 | 221 | Ý |
| 116 | 90 | Z |
| 117 | 222 | Þ |
| 118 | 91 | [ |
| 119 | 92 | \ |
| 120 | 93 | ] |
| 121 | 94 | ^ |
| 122 | 95 | _ |
| 123 | 96 | ` |
| 124 | 97 | a |
| AE | 228 | ä |
| 125 | 230 | æ |
| 126 | 229 | å |
| 127 | 225 | á |
| 128 | 224 | à |
| 129 | 226 | â |
| 130 | 227 | ã |
| 131 | 98 | b |
| 132 | 99 | c |
| 133 | 231 | ç |
| 134 | 100 | d |
| 135 | 240 | ð |
| 136 | 101 | e |
| 137 | 233 | é |
| 138 | 232 | è |
| 139 | 234 | ê |
| 140 | 235 | ë |
| 141 | 102 | f |
| 142 | 103 | g |
| 143 | 104 | h |
| 144 | 105 | i |
| 145 | 237 | í |
| 146 | 236 | ì |
| 147 | 238 | î |
| 148 | 239 | ï |
| 149 | 106 | j |
| 150 | 107 | k |
| 151 | 108 | l |
| 152 | 109 | m |
| 153 | 110 | n |
| 154 | 241 | ñ |
| 155 | 111 | o |
| ue | 246 | ü |
| 156 | 243 | ó |
| 157 | 242 | ò |
| 158 | 244 | ô |
| 159 | 245 | õ |
| 160 | 248 | ø |
| 161 | 112 | p |
| 162 | 113 | q |
| 163 | 114 | r |
| 164 | 115 | s |
| ss | 223 | ß |
| 165 | 116 | t |
| 166 | 117 | u |
| ue | 252 | ü |
| 167 | 250 | ú |
| 168 | 249 | ù |
| 169 | 251 | û |
| 170 | 118 | v |
| 171 | 119 | w |
| 172 | 120 | x |
| 173 | 121 | y |
| 174 | 255 | ÿ |
| 175 | 253 | ý |
| 176 | 122 | z |
| 177 | 254 | þ |
| 178 | 123 | { |
| 179 | 124 | \| |
| 180 | 125 | } |
| 181 | 126 | ~ |
| 182 | 180 | ´ |
| 183 | 168 | ¨ |
| 184 | 175 | ¯ |
| 185 | 176 | ° |
| 186 | 161 | ¡ |
| 187 | 191 | ¿ |
| 188 | 164 | ¤ |
| 189 | 163 | £ |
| 190 | 165 | ¥ |
| 191 | 167 | § |
| 192 | 162 | ¢ |
| 193 | 182 | · |
| 194 | 181 | µ |
| 195 | 182 | ¶ |
| 196 | 190 | ¾ |
| 197 | 188 | ¼ |
| 198 | 189 | ½ |
| 199 | 178 | ² |
| 200 | 186 | º |
| 201 | 171 | « |
| 202 | 187 | » |
| 203 | 177 | ± |
| 204 | 127 | ■ |
| 205 | 128 | ■ |
| 206 | 129 | ■ |
| 207 | 130 | ■ |
| 208 | 131 | ■ |
| 209 | 132 | ■ |
| 210 | 133 | ■ |
| 211 | 134 | ■ |
| 212 | 135 | ■ |
| 213 | 136 | ■ |
| 214 | 137 | ■ |
| 215 | 138 | ■ |
| 216 | 139 | ■ |
| 217 | 140 | ■ |
| 218 | 141 | ■ |
| 219 | 142 | ■ |
| 220 | 143 | ■ |
| 221 | 144 | ■ |
| 222 | 145 | ‘ |
| 223 | 146 | ’ |
| 224 | 147 | ■ |
| 225 | 148 | ■ |
| 226 | 149 | ■ |
| 227 | 150 | ■ |
| 228 | 151 | ■ |
| 229 | 152 | ■ |
| 230 | 153 | ■ |
| 231 | 154 | ■ |
| 232 | 155 | ■ |
| 233 | 156 | ■ |
| 234 | 157 | ■ |
| 235 | 158 | ■ |
| 236 | 159 | ■ |
| 237 | 160 |  |
| 238 | 166 | ¦ |
| 239 | 169 | © |
| 240 | 172 | ¬ |
| 241 | 173 | - |
| 242 | 174 | ® |
| 243 | 175 | ¯ |
| 244 | 176 | ° |
| 245 | 184 | ¸ |
| 246 | 185 | ¹ |
| 247 | 215 | × |
| 248 | 247 | ÷ |

**LEXICAL ORDER IS SPANISH (Latin-1)**

| Order | Num | Chr | Order | Num | Chr | Order | Num | Chr | Order | Num | Chr | Order | Num | Chr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | ■ | 53 | 53 | 5 | 83 | 214 | Ö | 114 | 107 | k | 156 | 221 | Ý |
| 1 | 1 | ■ | 54 | 54 | 6 | NO | 216 | Ø | 115 | 108 | l | 157 | 253 | ý |
| 2 | 2 | ■ | 55 | 55 | 7 | 84 | 80 | P | 117 | 109 | m | 158 | 180 | ´ |
| 3 | 3 | ■ | 56 | 56 | 8 | 85 | 81 | Q | 118 | 110 | n | 159 | 181 | µ |
| 4 | 4 | ■ | 57 | 57 | 9 | 86 | 82 | R | 119 | 241 | ñ | 160 | 182 | ¶ |
| 5 | 5 | ■ | 58 | 58 | : | NU | NO | S | 120 | 111 | o | 161 | 148 | ¾ |
| 6 | 6 | ■ | 59 | 59 | ; | 88 | 84 | T | 120 | 242 | ò | 162 | 128 | ■ |
| 7 | 7 | ■ | 60 | 60 | < | 89 | 85 | U | 120 | 243 | ó | 163 | 129 | ■ |
| 8 | 8 | ■ | 61 | 61 | = | 89 | 217 | Ù | 120 | 244 | ô | 164 | 130 | ■ |
| 9 | 9 | ■ | 62 | 62 | > | N9 | 218 | Ú | 120 | 245 | õ | 165 | 131 | ■ |
| 10 | 10 | ■ | 63 | 63 | ? | 89 | 219 | Û | 120 | 246 | ö | 166 | 132 | ■ |
| 11 | 11 | ■ | 64 | 64 | @ | 89 | 220 | Ü | 120 | 248 | ø | 167 | 133 | ■ |
| 12 | 12 | ■ | 65 | 65 | A | 90 | 86 | V | 121 | 112 | p | 168 | 134 | ■ |
| 13 | 13 | ■ | 65 | 192 | À | 91 | NZ | W | 122 | 113 | q | 169 | 135 | ■ |
| 14 | 14 | ■ | 65 | 193 | Á | 92 | 88 | X | 123 | 114 | r | 170 | 136 | ■ |
| 15 | 15 | ■ | 65 | 194 | Â | 93 | 89 | Y | 124 | 115 | s | 171 | 137 | ■ |
| 16 | 16 | ■ | 65 | 195 | Ã | 94 | 90 | Z | ss | 223 | ß | 172 | 138 | ■ |
| 17 | 17 | ■ | 65 | 196 | Ä | 95 | 222 | Þ | 125 | 116 | t | 173 | 139 | ■ |
| 18 | 18 | ■ | 65 | 197 | Å | 96 | 91 | [ | 126 | 117 | u | 174 | 140 | ■ |
| 19 | 19 | ■ | 65 | 198 | Æ | 97 | 92 | \ | 126 | 249 | ù | 175 | 141 | ■ |
| 20 | 20 | ■ | 66 | 66 | B | 98 | 93 | ] | 126 | 250 | ú | 176 | 142 | ■ |
| 21 | 21 | ■ | 67 | 67 | C | 99 | 94 | ^ | 126 | 251 | û | 177 | 143 | ■ |
| 22 | 22 | ■ | 67 | 199 | Ç | 100 | 95 | _ | 126 | 252 | ü | 178 | 144 | ■ |
| 23 | 23 | ■ | 68 |  | Dh | 101 | 96 | ` | 127 | 118 | v | 179 | 145 | ‘ |
| 24 | 24 | ■ | 68 |  | DH | 102 | 97 | a | 128 | 119 | w | 180 | 146 | ’ |
| 25 | 25 | ■ | 69 | 68 | D | 102 | 224 | à | 129 | 120 | x | 181 | 147 | ■ |
| 26 | 26 | ■ | 70 | 208 | Ð | 102 | 225 | á | 130 | 121 | y | 182 | 148 | ■ |
| 27 | 27 | ■ | 71 | 69 | E | 102 | 226 | â | 130 | 255 | ÿ | 183 | 149 | ■ |
| 28 | 28 | ■ | 71 | 200 | È | 102 | 227 | ã | 131 | 122 | z | 184 | 150 | ■ |
| 29 | 29 | ■ | 71 | 201 | É | 102 | 228 | ä | 132 | 254 | þ | 185 | 151 | ■ |
| 30 | 30 | ■ | 71 | 202 | Ê | 102 | 229 | å | 133 | 123 | { | 186 | 152 | ■ |
| 31 | 31 | ■ | 71 | 203 | Ë | 102 | 230 | æ | 134 | 124 | \| | 187 | 153 | ■ |
| 32 | 32 |  | 72 | 70 | I | 103 | 98 | b | 135 | 125 | } | 188 | 154 | ■ |
| 33 | 33 | ! | 73 | 71 | G | 104 | 99 | c | 136 | 126 | ~ | 189 | 155 | ■ |
| 34 | 34 | " | 74 | 72 | H | 104 | 231 | ç | 137 | 180 | ´ | 190 | 156 | ■ |
| 35 | 35 | # | 75 | 73 | J | 105 |  | ch | 138 | 168 | ¨ | 191 | 157 | ■ |
| 36 | 36 | $ | 75 | 204 | Ì | 105 |  | cH | 139 | 175 | ¯ | 192 | 158 | ■ |
| 37 | 37 | % | 75 | 205 | Í | 106 | 100 | d | 140 | 176 | ° | 193 | 159 | ■ |
| 38 | 38 | & | 75 | 206 | Î | 107 | 240 | ð | 141 | 161 | ¡ | 194 | 160 |  |
| 39 | 39 | ' | 75 | 207 | Ï | 108 | 101 | e | 142 | 191 | ¿ | 195 | 160 | ! |
| 40 | 40 | ( | 76 | 74 | J | 108 | 207 | Þ | 143 | 164 | ¤ | 196 | 169 | × |
| 41 | 41 | ) | 77 | 75 | K | 108 | 233 | é | 144 | 168 | £ | 197 | 172 | ¬ |
| 42 | 42 | * | 78 | 76 | L | 108 | 234 | ê | 145 | 165 | ¥ | 198 | 173 | - |
| 43 | 43 | + | 79 |  | LL | 108 | 235 | ë | 146 | 167 | § | 199 | 174 | Σ |
| 44 | 44 | , | 79 |  | ll | 109 | 107 | k | 147 | 162 | ¢ | 200 | 175 | ‘ |
| 45 | 45 | - | 80 | 77 | M | 110 | 103 | g | 148 | 188 | ¼ | 201 | 170 | ª |
| 46 | 46 | . | 81 | 78 | N | 111 | 104 | h | 149 | 189 | ½ | 202 | 186 | º |
| 47 | 47 | / | 82 | 209 | Ñ | 112 | 105 | i | 150 | 170 | ª | 203 | 185 | ¹ |
| 48 | 48 | 0 | NO | 79 | O | 112 | 236 | ì | 151 | 186 | º | 204 | 215 | × |
| 49 | 49 | 1 | 83 | 210 | Ò | 112 | 237 | í | 152 | 171 | « | 205 | 247 | ÷ |
| 50 | 50 | 2 | 83 | 211 | Ó | 112 | 238 | î | 153 | 187 | » |  |  |  |
| 51 | 51 | 3 | 83 | 212 | Ô | 112 | 239 | ï | 154 | 177 | ± |  |  |  |
| 52 | 52 | 4 | NO | 213 | Õ | 113 | 106 | j | 155 | 127 | ■ |  |  |  |

**LEXICAL ORDER IS SWEDISH (Latin-1)**



**ISO-932 Character Set**

Note in the table below, the characters marked with diamonds are taken to be introductory bytes for two-byte Shift-JIS characters.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 ■ | 32 | 64 @ | 96 ` | 128 □ | 160 | 192 ﾀ | 224 • |
| 1 ■ | 33 ! | 65 A | 97 a | 129 • | 161 ｡ | 193 ﾁ | 225 • |
| 2 ■ | 34 " | 66 B | 98 b | 130 • | 162 「 | 194 ﾂ | 226 • |
| 3 ■ | 35 # | 67 C | 99 c | 131 • | 163 」 | 195 ﾃ | 227 • |
| 4 ■ | 36 $ | 68 D | 100 d | 132 • | 164 、 | 196 ﾄ | 228 • |
| 5 ■ | 37 % | 69 E | 101 e | 133 • | 165 ・ | 197 ﾅ | 229 • |
| 6 ■ | 38 & | 70 F | 102 f | 134 • | 166 ヲ | 198 ﾆ | 230 • |
| 7 ■ | 39 ' | 71 G | 103 g | 135 • | 167 ァ | 199 ﾇ | 231 • |
| 8 ■ | 40 ( | 72 H | 104 h | 136 • | 168 ィ | 200 ﾈ | 232 • |
| 9 ■ | 41 ) | 73 I | 105 i | 137 • | 169 ゥ | 201 ﾉ | 233 • |
| 10 ■ | 42 * | 74 J | 106 j | 138 • | 170 ェ | 202 ﾊ | 234 • |
| 11 ■ | 43 + | 75 K | 107 k | 139 • | 171 ォ | 203 ﾋ | 235 • |
| 12 ■ | 44 , | 76 L | 108 l | 140 • | 172 ャ | 204 ﾌ | 236 • |
| 13 ■ | 45 - | 77 M | 109 m | 141 • | 173 ュ | 205 ﾍ | 237 • |
| 14 ■ | 46 . | 78 N | 110 n | 142 • | 174 ョ | 206 ﾎ | 238 • |
| 15 ■ | 47 / | 79 O | 111 o | 143 • | 175 ッ | 207 ﾏ | 239 • |
| 16 ■ | 48 0 | 80 P | 112 p | 144 • | 176 ー | 208 ﾐ | 240 • |
| 17 ■ | 49 1 | 81 Q | 113 q | 145 • | 177 ｱ | 209 ﾑ | 241 • |
| 18 ■ | 50 2 | 82 R | 114 r | 146 • | 178 ｲ | 210 ﾒ | 242 • |
| 19 ■ | 51 3 | 83 S | 115 s | 147 • | 179 ｳ | 211 ﾓ | 243 • |
| 20 ■ | 52 4 | 84 T | 116 t | 148 • | 180 ｴ | 212 ﾔ | 244 • |
| 21 ■ | 53 5 | 85 U | 117 u | 149 • | 181 ｵ | 213 ﾕ | 245 • |
| 22 ■ | 54 6 | 86 V | 118 v | 150 • | 182 ｶ | 214 ﾖ | 246 • |
| 23 ■ | 55 7 | 87 W | 119 w | 151 • | 183 ｷ | 215 ﾗ | 247 • |
| 24 ■ | 56 8 | 88 X | 120 x | 152 • | 184 ｸ | 216 ﾘ | 248 • |
| 25 ■ | 57 9 | 89 Y | 121 y | 153 • | 185 ｹ | 217 ﾙ | 249 • |
| 26 ■ | 58 : | 90 Z | 122 z | 154 • | 186 ｺ | 218 ﾚ | 250 • |
| 27 ■ | 59 ; | 91 [ | 123 { | 155 • | 187 ｻ | 219 ﾛ | 251 • |
| 28 ■ | 60 < | 92 ¥ | 124 \| | 156 • | 188 ｼ | 220 ﾜ | 252 • |
| 29 ■ | 61 = | 93 ] | 125 } | 157 • | 189 ｽ | 221 ﾝ | 253 □ |
| 30 ■ | 62 > | 94 ^ | 126 ~ | 158 • | 190 ｾ | 222 ﾞ | 254 □ |
| 31 ■ | 63 ? | 95 _ | 127 ■ | 159 • | 191 ｿ | 223 ﾟ | 255 □ |

# Overview of the Shift-JIS Character Set

The table below shows the categories of two-byte characters in the Shift-JIS character set. The range values are in hexadecimal. The second byte in a Shift-JIS character may have values between hexadecimal 40 and FC, excluding 7F.

| Range | Type of character |
|---|---|
| 8140-81FC | Symbols |
| | |
| 8240-824E | undefined |
| 824F-8258 | Digits |
| 8259-825F | undefined |
| 8260-8278 | Uppercase Roman letters |
| 8279-8280 | undefined |
| 8281-8299 | Lowercase Roman letters |
| 829B-829E | undefined |
| 829F-82F1 | Hiragana |
| 82F2-82FC | undefined |
| | |
| 8300-8396 | Katakana |
| 8397-839E | undefined |
| 839F-83B6 | Uppercase Greek letters |
| 83B7-83BE | undefined |
| 83BF-83D6 | Lowercase Greek letters |
| 83D7-83FC | undefined |
| | |
| 8440-8461 | Uppercase Russian Cyrillic |
| 8462 | undefined |
| 8463-8491 | Lowercase Russian Cyrillic |
| 8492-849E | undefined |
| 849F-84BE | Box drawing |
| 84BF-84FC | undefined |
| | |
| 85xx-86xx | undefined |
| | |
| 8740-879C | symbols |
| 879D-97FC | undefined |
| | |
| 8840-889E | undefined |
| 889F-8FFC | Level 1 kanji |
| | |
| 89xx-9872 | Level 1 kanji |
| 9873-989E | undefined |
| | |
| 989F-9FFC | Level 2 kanji |
| | |
| E0xx-EFxx | Level 2 kanji |
| | |
| F0xx-FCxx | Level 3 kanji (undefined in most implementations) |

The level 1 kanji are arranged in the order of the hiragana representations of their most common *on* pronunciation. The level 2 and level 3 kanji are arranged in order of the stroke count of their principal radical followed by the stroke count of the remaining portion of the character.

All two-byte characters, including the Roman, Greek, and Russian characters, are twice as wide when displayed as the one-byte characters. The HTBasic CVT$ function can convert between one- two-byte Roman and katakana characters.

Note that voiced katakana and hiragana characters are represented by a single two-byte character in the shift-JIS character set while they are represented by a one-byte character plus a separate one-byte voicing mark in the ISO-932 character set.

# Ancillary files

The set of files needed for proper operation of HTBasic Plus.

# Angle

Angles can be specified in radians or degrees. When specifying angles for graphic statements, the angle is relative to the positive x axis. Positive angles specify counter-clockwise movement about the origin.

# Array

An array is a multi-dimensional ordered set of values. Each member of the set is called an array element. All the members of the set have the same simple data type which can be integer, long, real, complex or string. The dimension of the set is called the RANK of the array. Arrays may have a rank from one to six.

Local array variables are declared using INTEGER, LONG, REAL, COMPLEX, DIM and STATIC. ALLOCATE can be used to dynamically declare an array. COM can be used to declare a global array. Consult these entries in the on-line *Reference Manual* to learn how to declare array variables. OPTION BASE is available to change the default lower bound for indices.

# Array Name

The rules for naming an array are the same as for a variable (see Variable Name). Array variables and simple variables share the same name space. Thus, you cannot have a simple variable and an array variable with the same name in the same context.

# ASCII file type

In the HTBasic manual set, the term ASCII file refers to a LIF ASCII file, not a DOS ASCII or UNIX ASCII ordinary file. A LIF ASCII file is a typed file which contains string items preceded by an item length, and followed by a pad byte when the string length is odd. Do not confuse the terms DOS ASCII, UNIX ASCII and LIF ASCII. A DOS ASCII file is an ordinary file which contains only printable characters and the end of each line is marked with a carriage return and line feed. A UNIX ASCII file is an ordinary file which contains only printable characters and the end of each line is marked with a line feed. HTBasic can read and write any of these file types. See CREATE and CREATE ASCII in the on-line *Reference Manual*.

# Attributes

Qualities of a dialog or a widget. Each attribute has a value or set of values that can be set with CONTROL or read with STATUS. The values of the attributes determine the function and appearance of the dialogs and/or widgets. Most attributes have default values.

# BDAT file type

BDAT files are used to hold binary data and can be used to exchange data with HP BASIC. See CREATE BDAT in the on-line *Reference Manual*. Ordinary (DOS, NT or UNIX) files can also be used to hold binary data.

# Boolean Expression

A boolean expression is simply a numeric expression whose result is tested for zero/non-zero. If the result is zero, the expression is considered FALSE. If the result is non-zero, the expression is considered TRUE.

# Child widget

A widget that is one level below its parent in the widget hierarchy. Child widgets are contained by their parent widget. Therefore, they cannot be level-0 widgets. A widget becomes a child when you assign it to a parent with the PARENT option to the ASSIGN keyword. A widget can be both a parent and child widget at the same time.

# Click

To press and immediately release the mouse button.

# COM Block

A COM block is a set of one or more variables that may be shared (in "COMmon"), among one or more contexts. Each COM block is uniquely identified with a name (although one block is allowed to be nameless). COM block names are explained below.

The value of a COM variable is global in lifetime, however, the name of a COM variable is not global. To access COM variables, a context must include a COM statement which identifies the COM block and gives the names by which the variables will be known in that context. Thus, each context can give a different name to the same COM variable. COM variables are hidden from all contexts which do not include a COM statement accessing that COM block. See COM in the on-line *Reference Manual*.

# COM Block Name

Rules for naming a COM block are the same as for a variable (see Variable Name).

# Common dialog attribute

An attribute of all of the dialogs. See the on-line *HTBasic Plus Reference Manual* for a discussion of common dialog attributes.

# Common widget attribute

An attribute of all of the widgets. See the on-line *HTBasic Plus Reference Manual* for a discussion of common widget attributes.

# COMPLEX

"Complex" is a data type. Other data types are integer, long, real, string, and I/O path. A complex number is an ordered pair (x, y) denoted by mathematicians as:

*x* + *iy*

Where:

*x* is the real part of the complex number.

*y* is the imaginary part of the complex number. The *i* in front of the *y* forms the imaginary number *iy* and is the same as multiplying *y* by the square root of -1or -1^(1/2). For example, the square root of -9 or -9^(1/2) could be considered as the -1^(1/2) x 9^(1/2) or 3*i*.

BASIC complex numbers are stored as two REAL numbers. This means that a complex number requires 16 bytes of memory (each REAL component takes 8 bytes).

The IBM PC and HP PA workstations use IEEE Std 754-1985 for Binary Floating point numbers. This gives the Complex data type an approximate range of 2E-308 to 1E+308 and 15 decimal digits of precision. Both positive and negative numbers are represented. MINREAL and MAXREAL are functions which return the smallest and largest positive real numbers. The range for negative numbers is -MINREAL to -MAXREAL.

Use the COMPLEX statement to declare local complex variables and the COM statement to declare global complex variables. Use the ALLOCATE statement to declare a local complex variable which can be DEALLOCATEd dynamically. If a variable is not declared, it will automatically be declared local and real unless CONFIGURE DIM OFF is used.

# Container widget

The container widget is always a parent widget. Most commonly, it is a PANEL widget in which you place a set of child widgets. That is, the child widgets are contained in the PANEL widget. A container widget can be, but does not have to be, a level-0 widget.

# Context

A context is a program unit with its own environment, including local variables, which can be called recursively by other contexts, and can pass arguments, either by reference or by value. There are four types of contexts: 1) main context, 2) subprogram context, 3) user defined function context, 4) CSUB context. Context changes occur when subprograms or functions are invoked or exited.

The main context begins with the first line of the program and ends with the program line containing the "END" statement. The main context is started by a RUN command.

A subprogram context begins with a SUB statement and ends with a SUBEND statement. It is called with a CALL statement and terminates with a SUBEND or SUBEXIT statement. Arguments can be passed to a subprogram.

A user defined function begins with a DEF statement and ends with an FNEND statement. It is called from within a numeric or string expression by referencing its name. It terminates and returns a value with a RETURN statement. The expression then continues to evaluate, using the value returned in place of the function reference. Arguments can be passed to a function.

A CSUB is a compiled subprogram created with special tools outside of HTBasic. It is loaded into memory with the LOADSUB statement and removed from memory with the DELSUB statement. It is called with a CALL statement.

# Context-sensitive Help

Online information that is relevant to what the user is doing with a widget or a dialog. When enabled via the attributes HELP FILE and HELP TOPIC, context-sensitive help is activated by the click of the right mouse button.

# Device Selector

A device selector is a number which specifies a device. It specifies the interface select code (ISC) to which a device is connected. If more than one device can be connected to that interface (i.e., the GPIB interface), then the address of the device is appended after the ISC. It can be just a primary address or a primary address and several secondary addresses. Each address is specified with two digits; thus 1 is specified as 01. A device selector can be up to 15 digits.

Several examples follow: If a printer has a primary address of 1 and is connected to a GPIB interface with ISC 7, then the device selector for the printer is 701. If an instrument is connected to the RS-232 interface with ISC 9, then the device selector for the instrument is 9. If a GPIB plotter has a primary address of 2, a secondary address of 11 and is connected to a GPIB interface with ISC 14, then the device selector for the plotter is 140211.

# Dialog

One of the fundamental HTBasic Plus entities. A dialog is created on the computer screen with the DIALOG statement from an executing BASIC program or from the command line.

# DOS file type

HTBasic supports ordinary files as well as typed files. HTBasic file types are LIF ASCII, BDAT, BIN and PROG. In a CAT listing ordinary files are listed as "DOS" files by the DOS versions of HTBasic. Other versions leave the file type column blank for ordinary files. Unlike typed files, no special header or other embedded information is placed in the file. Under DOS, an ordinary file with FORMAT ON is compatible with all programs that support DOS ASCII files. See CREATE in the on-line *Reference Manual*.

# Event

An event is the occurrence of an action or condition which can be trapped by an ON statement that directs program execution to a service routine. See ON in the on-line *Reference Manual*.

# Event-initiated Branching

A programming technique that uses interrupts to redirect program flow.

# File Specifier

A file specifier identifies a file. A file specifier consists of an optional drive letter, an optional path, a filename and an optional filename extension combined as follows:

d:\path\filename.ext

The drive letter specifies the disk drive, A, B, C, etc. If it is present, it must be followed by a colon, ":". The path is a series of one or more directory names, separated by the backslash character, "\", leading from the root directory to the file in question. A legal directory name follows the same rules as a legal filename.

For the FAT 16 file system, the filename consists of 1 to 8 characters. The extension consists of a period, "." followed by 1 to 3 characters. Case is ignored and when a new filename is specified all lowercase characters are converted to uppercase. Some characters are not legal in a filename. A period is only legal between the filename and the extension. Characters less than CHR$(32) are not legal. The characters in the following list are also illegal: "*+,/:;<=>?[\]|.

For the NTFS and FAT 32 file systems, the filename consists of 1 to 256 characters, including one or more extensions. Case is ignored, although when a new filename is specified, case is preserved for display in a directory listing. Some characters are not legal in a filename. Characters less than CHR$(31) are not legal. The characters in the following list are also illegal: "*/:<>?\|. Trailing spaces are ignored; elsewhere spaces are acceptable.

# Focus

When you click the pointer on the screen (with the mouse or with the keyboard) within the border of a dialog or widget, the dialog or widget "gains the focus". That is, any mouse clicks or keystrokes that you make will be input to the widget that has the focus. Programmatically, focus can be obtained by using the STACKING ORDER attribute.

# Full Array Specifier

A full array specifier is the symbol "(*)" and is used to reference an entire array rather than an individual element.

# Function Name

The rules for naming a function are the same as for a variable (see Variable Name). A User Defined Function is one of several types of contexts (see Context).

# Help

The online help system that includes information on all HTBasic for Windows keywords, programming examples, and other online information. Online help topics are linked to one another through hyperlinks.

# HPGL

Hewlett Packard Graphics Language. Used to communicate between HTBasic and plotters.

# I/O PATH

"I/O path" is a data type. Other data types are integer, long, real, complex and string. An I/O path is implicitly declared whenever you use it in a program. It must be initialized with the ASSIGN statement before it is used. Input and Output statements use an I/O path to specify the entity (device, file, pipe, buffer, etc.) that the computer communicates with during the I/O operation. When an input/output statement does not explicitly involve an I/O path, one is created internally, used for the duration of the statement and then discarded.

# Integer

"Integer" is a data type. Other data types are I/O path, long, real, complex, static and string. Integers are whole numbers (-1, 35) as opposed to real numbers that can have fractional parts (1.7, 2.34). Integers are stored in two bytes and have a range of -32,768 to +32,767. Integer operations are faster and integers take less space to store.

Use the INTEGER statement to declare local integer variables, the COM statement to declare global integer variables and STATIC to declare local persistent variables. Use the ALLOCATE statement to declare a local integer variable which can be DEALLOCATEd dynamically. If a variable is not declared, it will automatically be declared local and real unless CONFIGURE DIM OFF is used.

# Integer Array

Each element of an array (see Array) is an integer declared with INTEGER.

# Interface Select Code

Interface select codes (ISC) specify hardware interfaces that connect the computer to devices. Some ISCs are fixed:

| ISC | Fixed Devices |
|---|---|
| 1 | CRT display |
| 2 | Keyboard |
| 3 | Graphic display |
| 6 | Bit mapped graphic |
| 10 | Windows Print Manager |
| 26 | Parallel Port |
| 32 | Processor |

Others can be specified when the device is loaded with LOAD BIN. If the ISC is not specified, the following defaults are used:

| ISC | Loadable Devices |
|---|---|
| 7 | GPIB Board |
| 8 | 2nd GPIB Board |
| 9 | RS-232 Port (COM1) |
| 11 | 2nd RS-232 Port (COM2) |
| 12 | GPIO Board |
| 18 | Several data acquisition boards |

# Level-0

The first level of the widget management software's hierarchy of widgets. The first level of the hierarchy is the one just below the screen level.

# Level-0 widget

A level-0 widget has a title bar on top of the widget and a resize border around the widget. You can assign most widgets to be level-0 widgets. (All dialogs are level-0.) You create a level-0 widget by not using the PARENT option to the ASSIGN keyword when you create the widget.

# Line Label

Line labels may optionally follow any line number. The use of line labels results in more structured programming. Line references to labels are unaffected by line numbering. The rules for naming a line label are the same as for variables (see Variable Names). A colon follows the name in the line that is labeled, but does not follow the name in lines referencing that line.

# Line Number

Each program line requires a unique line number at the beginning of the line. Line numbers must be in the range of 1 to 4,194,304. HTBasic ignores leading zeros and spaces before line numbers. Optionally, one may elect not to display line numbers. Line numbers are used to:

- indicate the order of statement execution
- provide control points for branching
- help in debugging and updating programs
- indicate the location of run-time errors

# Local Variable

All variables are local and are accessible only in the current context unless declared as COM variables. When the context begins execution, storage space is allocated for all local variables and their values are set to zero. When execution of the context is completed, the local variable storage space is released and their values are lost.

# LONG

"Long" is a data type. Other data types are I/O path, integer, real, complex and string. Longs are actually "long integers" and are in all ways identical to the integer data type, except that they have a range of -2,147,483,648 to 2,147,483,647. LONGs are stored in four bytes.

# Matrix

A matrix is a two dimensional numeric array. The RANK of a matrix is two.

# Menu bar

The area just below the title bar in a level-0 widget. The menu bar contains menu widgets.

# Notepad

A text editing application, one of the Windows accessories.

# Numeric Array

A numeric array is an array (see Array) in which the data type of each element is either integer, long, real or complex.

# Numeric Array Element

A numeric array element is a simple value, either an integer, long, real, or complex number and is compatible with any operation which expects a single value. An element is specified by following the array name with a left parenthesis, "(", a comma-separated list of subscripts and a right parenthesis, ")". The number of subscripts specified must match the RANK of the array. The value of each subscript must lie in the legal range for that dimension as defined in the declaration statement (ALLOCATE, COM, COMPLEX, DIM, INTEGER, LONG, REAL, REDIM, STATIC). Some matrix operations redefine the range of a dimension.

# Numeric Constant

A constant is an entity with a fixed value. There are three types of numeric constants: integer, long and real. An integer constant is a whole number not specified with a decimal point, ".", nor with scientific notation, which falls in the range -32,768 to 32,767. Integer constants can be expressed in decimal, octal (base 8) or hexadecimal (base 16). An octal constant must begin with the characters "&O" or simply "&". A hexadecimal constant must begin with the characters "&H. A long constant is in all ways identical to an integer constant, except that it can have a range of -2,147,483,648 to 2,147,483,647. A real constant is specified with a decimal point or scientific notation, or is outside the integer range. Some integer constants are "1", "-20000", "&H7FFF" and "&O377. Some real constants are "-1.0", "1E+10" and "6000000000".

# Numeric Expression

A numeric expression is any legal combination of operands and operators joined together in such a way that the expression as a whole can be reduced to a numeric value. The following syntax diagram defines the legal combination of operands and operators. Precedence rules provide additional constraints on an expression (see Precedence). The syntax is:

numeric-expression =
    { + | - | NOT } numeric-expression |
    ( numeric-expression ) |
    numeric-expression operator numeric-expression |
    numeric-constant | numeric-name |
    numeric-array-element |
    numeric-function [ ( param [,param...] ) ] |
    FN function-name [ ( param [,param...] ) ] |
    string-expression compare-operator string-expression

where:

operator = + | - | * | / | DIV | MOD | MODULO | ^ |
    AND | OR | EXOR | compare-operator
compare-operator =  <> | = | < | > | <= | >=
numeric-function = a function, like COS, which returns a numeric value.
param = legal parameters for numeric functions and user defined
    functions are explained in the on-line *Reference Manual*.

# Numeric Name

The rules for naming a numeric variable are explained under "Variable Name". A numeric variable is of type integer, long, real or complex.

# Operator

The person who interacts (using the mouse or keyboard) with the widgets and dialogs on the screen as the program is running.

# Ordinary file

HTBasic supports ordinary files as well as typed files. HTBasic file types are LIF ASCII, BDAT, BIN and PROG. All other files are ordinary files. In a CAT listing, the file type column is blank for ordinary files or gives the operating system (i.e., "DOS" or "HP-UX"). Unlike typed files, no special header or other embedded information is placed in the file. Under DOS or NT, an ordinary file with FORMAT ON is compatible with all programs that support DOS/NT ASCII files. See CREATE in the on-line *Reference Manual*.

# Parent widget

A widget that is one level above all of its children in the widget hierarchy. A parent widget contains its child widgets. A parent widget can be a level-0 widget, but is not necessarily one.

A widget becomes a parent when you assign child widgets to it using the PARENT option in the child widgets' ASSIGN statements. A widget can be both a parent and child widget at the same time.

# Path Specifier

A path specifier in HTBasic is similar to an MSUS (Mass Storage Unit Specifier) in HP BASIC. It identifies a place where files are stored. Depending on your operating system, the necessary information to uniquely identify such a place includes: the device, address, volume, unit, and directory path list. A summary of the rules is given here.

Under Windows, a path specifier consists of an optional disk drive letter and an optional directory path. If the disk drive letter is omitted, the default disk is used. A directory path is composed of the names of the directories which form the path from the root directory "\", to the directory where you wish to access files. Each directory name is separated from the others with the backslash, "\", symbol. The rules for each directory name are the same as for a filename (File Specifier). If the directory path is omitted, the default directory is used.

For example, suppose that you wish to use drive "C:" and a catalog of the root directory "C:\" shows a directory named "HTB". Suppose that a catalog of "C:\HTB" shows a directory named "FILES.BIN". And suppose that it is this directory you wish to specify with a path specifier. The correct path specifier is "C:\HTB\FILES.BIN". If drive "C:" is the default drive, then the "C:" could be omitted. If directory HTB is the default directory, then the "\HTB\" could be omitted. Please read your operating system manual for a greater understanding of these concepts.

# Pen

Is used to specify the colors in widgets, and may be set to any legal HTBasic for Windows pen number. See the PEN keyword for more information.

# Pen Number

The term "pen number" is used in two different ways. The appropriate range is explained in the text describing the statement.

The first way in which the term "pen number" is used is for CRT color attribute values. The legal values are:

| Pen | Color |
|---|---|
| 136 | White |
| 137 | Red |
| 138 | Yellow |
| 139 | Green |
| 140 | Cyan |
| 141 | Blue |
| 142 | Magenta |
| 143 | Black |

The second way in which the term "pen number" is used is in statements affecting graphic colors. In these instances, pen numbers begin at zero and go to N-1, where N is the number of colors displayable at the same time on the computer display.

# Pixel

The image on a computer monitor comprises an array of dots that vary in color and intensity. A single dot is called a pixel.

# Pointer

The arrow-shaped cursor that appears within widgets and dialogs. You move the focus from one widget to the next by moving the pointer from one widget to the next.

# Precedence

Mathematical precedence describes the order in which operators in an expression are evaluated. Some cheap calculators execute each operation as it is entered. If you are used to this type of calculator, you may be confused by the concept of precedence. For example, the correct answer to the formula:

```
1+2*3+4
```

is 11, not 13. This is because multiplication (2*3) has a higher precedence than addition (1+2). If the two operators are on the same row in the precedence chart, the operations occur in left to right order (i.e. 1+2-3+4).

HP BASIC (and HTBasic) has an odd quirk in its definition of precedence which you should be aware of. Most computer languages place all monadic operators (operators which operate on one operand) at a higher precedence than dyadic operators (operators which operate on two operands). However, HTBasic and HP BASIC place monadic + and - below some of the dyadic operators. The following is one example of an expression that will evaluate differently because of this:

```
-4^0.5
```

With HTBasic, this is equivalent to -(4^0.5) which is equal to -2. With most other computer languages, this is equivalent to (-4)^0.5 which is an illegal operation.

# Primary Address

A primary address is a numeric expression which can be rounded to an integer in the range 0 to 31. It specifies the address of a device on the GPIB bus. Usually, GPIB devices have a switch which allows their primary address to be set to any of the values 0 through 31.

# Priority

Priority is a measure of the relative importance of the currently executing line and allows higher priority events to interrupt lower priority events, while preventing lower priority events from interrupting higher priority events. Priority values can range from 0 (least important) to 15 (most important). The ON statement which defines the service routine for an event also allows the priority for that service to be defined. The system priority is the priority of the currently executing line and can be changed with the SYSTEM PRIORITY statement.

# PROG file type

PROG files are used to hold binary program images and are the most efficient file type for storing an HTBasic program. See STORE in the on-line *Reference Manual* for information about PROG files.

# Real

"Real" is a data type. Other data types are integer, long, complex, string and I/O path. The Real data type is a subset of all rational numbers. The particular subset depends on your computer. Most computers, including the IBM PC and HP PA workstations use IEEE Std 754-1985 for Binary Floating point numbers. This gives the Real data type an approximate range of 2E-308 to 1E+308 and 15 decimal digits of precision. Both positive and negative numbers are represented. MINREAL and MAXREAL are functions which return the smallest and largest positive real numbers. The range for negative numbers is -MINREAL to -MAXREAL.

Use the REAL statement to declare local real variables and the COM statement to declare global real variables. Use the ALLOCATE statement to declare a local real variable which can be DEALLOCATEd dynamically. If a variable is not declared, it will automatically be declared local and real unless CONFIGURE DIM OFF is used.

Please Note: Internally, real numbers are represented in a binary format. You need not understand this format, but you should understand its implications. It is possible to have two different numbers in this format whose 15 digit decimal representations are the same. However, when comparing or subtracting these two "look-equal" numbers, you will find they are not equal. Also, when the result of an arithmetic operation is a number not representable in the binary format, an approximation must be used instead. You should take this into account and keep track of the error bounds as approximate numbers are used in further calculations.

# Record Number

The record number is a numeric expression which is rounded to an integer to specify a record within a file. The first record is one. BDAT and ordinary files allow random access by specifying a record number in the I/O statement. The record length for ordinary files is always one. The record length for BDAT files is defined when the file is created with the CREATE BDAT statement.

# Resize border

The border that appears around level-0 widgets and dialogs and is used
to resize them. (You cannot use the resize border from the keyboard.) To use the resize border:

- Use the mouse to move the pointer to the resize border.
- When the pointer changes shape, press-and-hold the left mouse button and drag the border to the new position.
- Release the left mouse button and the widget will redraw at the new size.

# Scientific Notation

Scientific notation can be used to represent numbers by using the shorthand notation "n.nnnEmmm" instead of "n.nnn x 10^mmm".

# Screen Builder

An application that allows the user to build a user interface by selecting from a menu of graphic objects (dialogs and widgets) and editing on the screen.

# Screen origin

The upper-left corner of the screen, which has the coordinates (0,0).

NOTE: The screen origin in HTBasic Plus is different than for HTBasic for Windows. In HTBasic for Windows, the screen origin is in the lower-left corner of the screen.

# Sibling widgets

Sibling widgets are child widgets with the same parent, or at the same level in the widget hierarchy, as the widget for which you are setting the STACKING ORDER attribute value.

# Signal Number

A signal number is a numeric expression rounded to an integer in the range 0 to 15. A signal is an event which can be generated by the SIGNAL statement and can be handled by a routine set up with the ON SIGNAL statement.

# Softkey Macro

Also called a typing aid, a softkey macro is a sequence of keys assigned to a softkey. When the softkey is pressed, the sequence is typed into the keyboard buffer just as if you had typed them yourself. The definition of the softkey macro is user definable.

# STATIC

"Static" is a data condition. A static variable is persistent during a single run of an HTBasic program. Typically, static variables will only be used in SUB programs and/or FN functions because the MAIN context is usually called only once.

Static variables can effectively take the place of COM variables as they are presently used in many cases. If access to a COM variable is required in multiple SUBs and/or Functions (DEF FN) and/or the Main context, then a static variable is not appropriate. The scope of a static variable is limited to the context in which it is declared. In other words, a static variable declared in a SUB program cannot be accessed anywhere other than within that particular SUB program.

Up to 6 array bounds may be specified, the initial values are optional. Specifying an initial value for an array initializes each individual element in all dimensions of the array to the initial value specified.

# String

"String" is a data type. Other data types are integer, long, real, complex and I/O path. A string is a combination of ASCII characters. These are the letters, numbers and symbols that you can type on the keyboard. ASCII characters also include control characters such as carriage return, etc. A string can be just one character long or it can be one word, one sentence, one paragraph long or any combination of letters, numbers, spaces and symbols up to a maximum length of 32,767 characters.

Use the DIM statement to declare a local string variable and define its maximum length. The length of a string variable can never exceed its declared length. Use the ALLOCATE statement to declare a local string variable which can be DEALLOCATEd dynamically. Use the COM statement to declare a global string variable. If a string variable is not declared, it will be automatically declared as an 18 character maximum length local string variable unless CONFIGURE DIM OFF is used.

# String Array

A string array is an array (see Array) in which the data type of each element is string.

# String Array Element

A string array element is a simple string and is compatible with any function or operation which expects a single string value. An element is specified by following the array name with a left parenthesis, "(", a comma-separated list of subscripts and a right parenthesis, ")". The number of subscripts specified must match the RANK of the array.

# String Expression

A string expression is any legal combination of operands and operators joined together in such a way that the expression as a whole can be reduced to a string value.

# String Literal

A string literal is a string of characters delimited by the quote (") character. To include a quote character in the string, include two quote characters in the place of the one you wish to include. For example """hello""".

# String Name

The rules for naming a string variable are the same as for a variable (see Variable Name) plus the addition of a trailing dollar sign, "$". A string variable is a variable whose data type is "string".

# Sub-string

A substring defines a portion of a string variable or string array element. It is selected by specifying a starting position within the string value and optionally, either the length of the sub-string, or the ending position within the string value. If only the starting position is specified, the rest of the string value from that point on is used for the sub-string. String positions are one-based, i.e., the first character of a string is in position one.

The beginning position must be at least one and no greater than the current length plus one. When only the beginning position is specified, the substring includes all characters from that position to the current end of the string.

The ending position must be no less than the beginning position minus one and no greater than the dimensioned length of the string. When both beginning and ending positions are specified, the substring includes all characters from the beginning position to the ending position or current end of the string, whichever is less.

The maximum substring length must be at least zero and no greater than one plus the dimensioned length of the string minus the beginning position. When a beginning position and substring length are specified, the substring starts at the beginning position and includes the number of characters specified by the substring length. If there are not enough characters available, the substring includes only the characters from the beginning position to the current end of the string.

# Subprogram Name

The rules for naming a subprogram are the same as for a variable (see Variable Name). A subprogram is one type of context (see Context).

# Subscript

A subscript is a numeric expression rounded to an integer to specify an array dimension. The value of each subscript must lie in the legal range for that dimension as defined in the declaring statement (ALLOCATE, COM, COMPLEX, DIM, INTEGER, LONG, REAL, REDIM, STATIC). Some matrix operations automatically redefine the range of a dimension.

# System font

The default font used in dialogs and widgets. It can be changed in the configuration file. The default depends on the CRT resolution.

# Tab group

A group of *like, child* widgets that are created within ASSIGN statements, one after the other in the program. You create a tab group
so that the operator can move the focus from one widget in the group
to the next (and wrap around again), without having to traverse every widget in the PANEL.

Only the widgets that have a TAB STOP attribute can be members of a tab group and can accept the system focus. Level-0 widgets cannot be members of a tab group.

A common example of a tab group is a row of buttons across the bottom of a PANEL. The buttons present the operator with a group of related choices, such as YES, NO and CANCEL.

# Title bar

The area at the top of the application and program windows. Also, the area at the top of a level-0 widget that contains the title text for the widget as well as the window menu button, minimize button, and maximize button.

# Transient widget

A widget that is created from a parent widget as a result of some program or operator action and is used to create a custom dialog. A transient widget is created using the TRANSIENT option to the ASSIGN keyword.

# UNIX file type

HTBasic supports ordinary files as well as typed files. HTBasic file types are LIF ASCII, BDAT, BIN and PROG. In a CAT listing, the file type column is blank for ordinary files or gives the operating system (i.e., "DOS" or "HP-UX"). Unlike typed files, no special header or other embedded information is placed in the file. Under UNIX, an ordinary file with FORMAT ON and EOL of CHR$(10) is compatible with all programs that support UNIX ASCII files. See CREATE in the on-line *Reference Manual*.

# Variable Name

A variable name can have up to fifteen characters. The characters can be alphabetic, numerals, underlines and characters in the range CHR$(128) to CHR$(254). (HP BASIC and some versions of HTBasic use the range CHR$(161) to CHR$(254).) The first character may not be a numeral or an underline. A variable name can be the same as a keyword if it is entered partly in upper case and partly in lower case. Variable names are listed with the first character in upper case and the remaining characters in lower case.

# Vector

A vector is a one dimensional numeric array, i.e., the RANK of the array is one.

# Volume Label

A volume label is present in some operating systems to label a mass storage volume (usually a disk).

A legal volume label is 11 characters long. Legal characters are the same as for file specifiers. The volume label, however, does not divide the 11 characters with a period between the 8th and 9th characters.

# Volume Specifier

A volume specifier in HTBasic is similar to an MSUS (Mass Storage Unit Specifier) in HP BASIC. However, for disk volumes with multiple directories, a volume specifier does not completely identify a place to store files (see Path Specifier).

Two types of volume specifiers are supported by HTBasic. The first is the native type used by your operating system. For DOS, Windows and NT, a volume specifier is the drive letter followed by a colon. For example, "C:". If used with a file specifier, it is appended onto the front of the filename, "C:DATA". For other operating systems, consult your manuals.

The second type of volume specifier supported by HTBasic is the HP BASIC compatible msus style. For example, ":CS80,700,0". Support for this type is included for compatibility with old HP programs. To use this type of volume specifier you must use the CONFIGURE MSI statement to define a translation between this type of volume specifier and the native type used by your system. For example:

```
CONFIGURE MSI ":CS80,700,0" TO "B:"
CONFIGURE MSI ":A" TO "A:"
CONFIGURE MSI ":,1400,1" TO "C:\HTB\1400\1"
```

The first example would allow a file specifier such as "DATA:CS80,700,0". The second example would allow a file specifier such as "DATA:A". If the CONFIGURE statement is not used, then an HP BASIC style volume specifier will cause an error. The third example shows an HP style volume specifier being equated with a DOS style path specifier.

# Widget

One of the fundamental HTBasic Plus entities. A widget is created on the computer screen with the ASSIGN statement from an executing BASIC program. See the *HTBasic Plus Reference Manual* for details on widgets.

# Widget Management Software

The software that controls all widget levels and positions on the computer screen. It maintains a hierarchy for widgets and dialogs and keeps track of all level-0/parent/child widget relationships.

# Work Area

The area in which the widget performs its essential function. For example, in the PRINTER widget the area that contains the text is the work area. In the METER widget, the work area is the area that contains the meter arc, needle, tick marks, limits boxes, and value box.

A level-0 widget surrounds the work area with a title bar, resize border, and (sometimes) a menu bar.