HEWLETT-PACKARD

# Software Development System BASIC Reference Manual

HP-94
Handheld Industrial
Computer

# HP-94 Handheld Industrial Computer

# Software Development System BASIC Reference Manual


**Edition 1 December 1986**

**Reorder Number**
**82520-90001**

## Notice

**Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.** Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.



**Portable Computer Division**
**1000 N.E. Circle Blvd.**
**Corvallis, OR 97330 U.S.A.**

**Printing History** **Edition 1** December 1986 Mfg. No. 82520-90002

# Contents

# 1

# Introduction

This *BASIC Reference Manual* provides programming reference material for the HP-94 Handheld Industrial Computer. The manual is divided into the following three chapters:

- The **Introduction** provides fundamental programming information and general information that applies to all of the BASIC keywords.
- The **Keyword Dictionary** defines all of the BASIC keywords and includes syntax diagrams to illustrate their use.
- The **Appendixes** contain a keyword list with a one-line summary of their use, reference tables for errors, keyboard layout, the on-board character set, and the display control codes.

## About This Manual

Take a few minutes to study the following paragraphs which define the formats and conventions used in this manual. They may not all conform to your previous experience.

## Keyword Descriptions

Each keyword is defined by a **description** of the keyword, a **syntax diagram** showing pictorially how the keyword is used, and a **table** listing parameters and their allowable ranges that may be used with the keyword. **Examples** of the use of keywords and **related keywords** are listed.

## Legend

The following legend appears at the top of each keyword page:

☐ Exists in Development System

☐ Works Same in Development System

☐ Allowed in `IF...THEN`

If the square is filled in (■), the statement in the legend applies to the keyword.

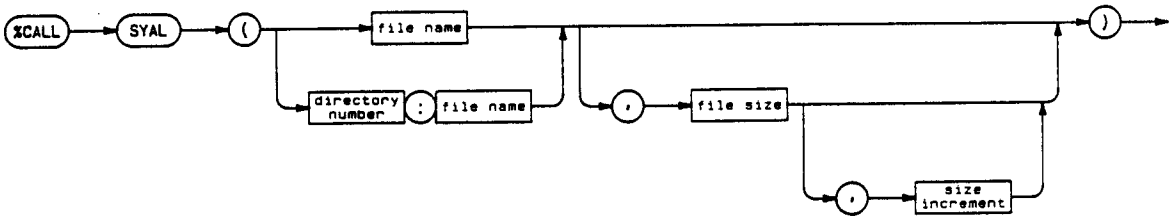
**Exists in Development System.** Keywords that exist in the development system are implemented in the `HXBASIC` Program Development Utility (see the *Utilities Reference Manual* for details about the utility). If this square is not filled in, the keyword exists only in the handheld. In some cases, the keyword description will instruct you to download an assembly-language subroutine or device handler to the handheld (using the utilities `HXC` and `HXCOPY`) before you can use the particular keyword.

**Works Same In Development System.** As you will note, not all keywords behave in the same way on both the development system and the HP-94 computers. If this is the case, the description of the keyword will contain a section called "Differences Between Development System and Handheld" that lists the differences of which you should be particularly aware.

**Allowed in `IF...THEN`.** Keywords with this feature can be included in `IF...THEN` statements. See the `IF...THEN` statement for details.

## The Syntax Diagram

The syntax diagram describes pictorially how to assemble a proper expression, statement, or command using the keyword. Items enclosed in ovals, circles, and rectangles are the elements of the expressions, statements, and commands.




**Format Conventions.** The shapes in the syntax diagrams follow the these conventions:

- The elements enclosed in ovals are keywords that must be typed in exactly as shown, except that uppercase and lowercase letters may be used interchangeably.
- The elements enclosed in circles are punctuation and keys that must be typed in exactly as shown.
- The elements enclosed in rectangles are parameters which are described in the table. Generally, uppercase and lowercase letters are NOT interchangeable.

The elements are connected into paths by arrows. Starting at the left of the diagram, you may follow any path in the direction indicated by the associated arrows. You must, however, end at the far right of the diagram. If several paths exist around one or more elements, each of the paths is optional; you should follow the path that does what you want to do. For example, the following are all valid statements:

```
%CALL SYAL ("PDAT")
%CALL SYAL ("PDAT",100)
%CALL SYAL ("PDAT",100,5)
```

Many optional elements have default values listed in the table of parameters. Line numbers and line labels are not shown in syntax diagrams.

**Spaces.** You may use one or more spaces between elements shown connected by an arrow. Consecutive ovals must have at least one space separating them. You may *NOT* use spaces between elements shown next to each other on a path without an arrow connecting them.

## Table of Parameters

The table describes each parameter used in the syntax diagram. When the parameter is required to assume values within a specified range, that range is listed. A dash ("—") indicates no range restrictions.

## Syntax Guidelines

*Syntax* is the way that instructions must be types so they can be understood by the computer. The following conventions are used throughout this manual.

| | |
|---|---|
| COMMANDS | Words in courier type (like DEF FN) represent commands and keywords that should be typed exactly as shown. The same type is also used to indicate sample statements and other computer output. |
| *parameters* | Items in italics are parameters you supply, such as the date *MM/DD/YY* to the TOD$ function. |
| non-printable characters | Non-printable control characters are represented by a two-letter icon. Common characters and their representations are:<br>Carriage Return – CR<br>Line Feed – LF<br>Escape – EC |
| keystrokes | This manual represents keystrokes in two ways:<br>■ Keystrokes that display characters are generally indicated by those characters. For example, * means "Press the Shift and * keys."<br>■ Keystrokes that do not display characters are represented by a keyshape ( ) printed with the key's symbol as shown on the keyboard. |

## Some BASIC Programming Information

Study the following paragraphs to make sure you understand the BASIC programming information they contain. It may not always conform to your previous experience.

## Files

The following information describes the characteristics of files:

**File Names.** File names on the handheld are restricted to four alphanumeric characters in length, the first of which must be an alphabetic character. Names beginning with SY are reserved for system files.

**File Types.** There are four types of files:

- A - assembly language programs
- B - BASIC language programs
- D - data files
- H - device handlers

**Autostart.** When the HP-94 is turned on, any program (file type A or B) named MAIN will start running automatically. Directories 0 through 4 (see below) are searched in ascending order and the first program found with the name MAIN will be executed.

**Directories.** There are six directories in the HP-94:

- Directory 0 refers to the main memory (64K for the HP-94D, 128K for the HP-94E, or 256K for the HP-94F).
- Directory 1 refers to plug-in memory (either the HP82411A 40KB RAM Card or the HP82412A ROM/EPROM Card).
- Directories 2-4 refer to additional plug-in read-only memory (on the HP82412A ROM/EPROM Card).
- Directory 5 contains the built-in software (HP-94 operating system and BASIC interpreter).

A directory can be specified with a file name in the format *directory*`:`*filename* (for example, `0:TEST`, `1:DAT`). When BASIC programs access files with OPEN # or SYAL, they can use the file name with or without the directory number. CALL and %CALL do not allow directory numbers.

## Program Lines

**Line Length.** The maximum number of characters that can be entered as a BASIC line is 127. This includes the line number and any embedded blank spaces.

**Line Numbers and Line Labels.** Every line in a program must be preceded by a unique line number – an integer in the range 0 through 32,767. The line number may be followed by a an optional line label. Letters, numbers, spaces and underscores may be used in labels. Label names are enclosed in brackets ([ ]). There is no limit to the number of characters in a label. To reference the label `Finished`, write, for example,

```
300 IF X<5 GOTO [Finished]
```

To include the label `Finished` on line `800`, write

```
800 [Finished] END
```

**Multistatement Lines.** A multistatement line contains two or more BASIC statements joined by the ":" character. If GOTO branching occurs in the middle of a line, the remaining statements on the line are not executed. Like single-statement lines, multistatement lines are limited to 127 characters.

## Constants

Constants used in programs are divided into the following three types:

**Real Constant.** This is a number within the range: $10^{-64} \leq x \leq 10^{+63}$ It is treated as 8-byte data and can be assigned to a real variable (described below).

Real constants can be specified with either of the following types of notation:

- Floating – Indicated by a real number of up to 14 digits. Examples are:
  `1.234, -0.2345, 10000000`
- Scientific – Indicated by a real number with a positive or negative mantissa of up to 14 digits, and a positive or negative exponent of up to two digits. Examples are:
  `1.23E12, -5.687E-12`

**Integer Constant.** This is an integer with the range $-32{,}768 \leq x \leq 32{,}767$. It is treated as 2- byte data and can be assigned to either an integer variable or a real variable (described below). It is processed according to the type of variable to which it is assigned.

**Literal.** This is a character string made up of 1-byte characters. It must be enclosed within double quotation marks and can be assigned to a string variable (described below).

Examples: `"RFF"`, `"1234"` (This is not the same as the numerical value 1,234.)

If you wish to use a double quote (") or an & in a character string, you must use two double quotes or two &s. For example, the string `RF"F` is represented by `"RF""F"` and `A&B` is represented by `"A&&B"`.

Any character can be represented by an & followed by a two-digit hexadecimal value for the ASCII code (for example "`&10`"). This is particularly useful for including control codes in strings sent to the display.

## Variables

BASIC uses the following variable types:

- Simple numeric: real or integer (default = real)
- Numeric array: real or integer (default = real)
  Dimensions: There can be up to 255 dimensions for numeric arrays. Any array is limited to a maximum of 65,535 bytes.
  The lower bound of array subscripts can be set using the `OPTION BASE` command. The maximum upper bound is 32,767.
- Simple string: Maximum string length: 255 (default = 8)
- String array: Maximum element length: 255 (default = 8)
  Dimensions: There can be up to 255 dimensions for string arrays. Any array is limited to a maximum of 65,535 bytes.
  The lower bound of array subscripts can be set using the `OPTION BASE` command. The maximum upper bound is 32,767.

String variables are differentiated from numeric variables by using a dollar sign ($) as the final character in all string variable names. Variable names can be up to 32 characters long. Any sequence of letters, numbers, and underscore characters can be used, except that the first character must be a letter. Variable names must be different from BASIC keyword names.

**NOTE** Uppercase and lowercase letters are not interchangeable in variable names.

While long variable names take up space in the BASIC program entered using HXBASIC, the names are removed by HXC so that they take no space on the handheld.

## Real-to-Integer Conversion

When a real constant or variable is assigned to an integer variable or interpreted as an integer by a BASIC keyword, the fractional part is dropped (for example, -1.9 becomes -1 and +1.9 becomes +1). This manual refers to this process as truncation. A conversion error will occur if the real number being converted is outside the range of integers.

## Comments

Comments may be added to the program by using the REM statement. For example:

```
400 REM This is a comment
```

Anything written on the line after REM is considered part of the comment; therefore, REMs should be the last statement on a multistatement line.

## Numeric Operators

The following table shows the numeric operators, in order of precedence:

**Numeric Operators**

| Operation | Symbol |
|---|---|
| Exponentiation | ** |
| Multiplication or Division | * or / |
| Addition or Subtraction | + or - |

In a calculation, operations inside parentheses take precedence.

## String Operators

There is only one string operator, the plus-sign (+). It is used for string concatenation.

## Relational Operators

The following table shows the relational operators:

**Relational Operators**

| Operation | Symbol |
|---|---|
| Equal | = |
| Greater than | > |
| Greater than or equal to | >= |
| Less than | < |
| Less than or equal to | <= |
| Not equal | <> |

Relational operators always return -1 for true and 0 for false. Relational operators can be included in logical expressions using AND, NOT, OR, and XOR, and in numeric and string expressions.

## Numeric and String Expressions

Numeric expressions are combinations of operators, BASIC functions, variables, and constants in standard BASIC (algebraic) notation which, when executed, return a single numeric results. String expressions are similar in concept, but return a single string result.

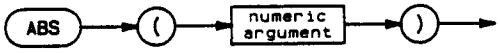
# 2

# BASIC Keyword Dictionary

# ABS

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The ABS function returns the absolute value of the numeric argument.




| Item | Description | Range |
|---|---|---|
| numeric argument | numeric expression | — |

## Examples

```
PositiveValue=ABS(Value)
PRINT ABS(Variable)
```

## Related Keywords

SGN

# ACS

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The ACS function returns the arccosine of the numeric argument as a real number in the range 0 through 180 degrees.


| Item | Description | Range |
|---|---|---|
| `numeric argument` | numeric expression | -1 through 1 |

## Examples

```
Theta=ACS(Y)
PRINT ACS(.5)
```

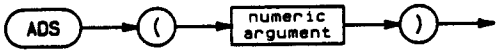
## Related Keywords

ADS, ARD, ASN, ATN, COS, DMS, PI, RAD, SIN, TAN

# ADS

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The ADS function interprets the numeric argument as an angle measured in degrees, minutes, and seconds and returns the value of the angle in decimal degrees.




| Item | Description | Range |
|---|---|---|
| numeric argument | numeric expression | — |

## Examples

```
Decdeg=ADS(Degminsec)
IF ADS(TIM)<ADS(start)+ADS(.0005) GOTO 100
```

## Description

The format of the argument is *DD*.*MMSSxxxxxx* as shown in the table below.

| Item | Description | Range |
|---|---|---|
| *DD* | Degrees. | — |
| *MM* | Minutes. | — |
| *SS* | Integer seconds. | — |
| *xx* | Fractional seconds | — |

ADS can also be used to convert hours, minutes, and seconds into decimal hours.

## Related Keywords

DMS, TIM

# AND

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The AND operator returns the bit-by-bit AND of the binary representation of the operands.


| Item | Description | Range |
|---|---|---|
| operand | numeric expression | -32,768 through +32,767 |

## Examples

```
IF S<>0 AND P<>0 THEN GOSUB 400
S=J(1) AND J(2)
```

## Description

The operands are truncated to integers represented as two's-complement. The results of each bit-by-bit AND are used to construct the integer result. Each bit is computed according the following truth table:

**Bit-by-Bit AND**

| Operand 1 | Operand 2 | Result |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Relational operators (=,<,>,<=,>=, <>) always return -1 for true and 0 for false. The bit-by-bit AND of these results will always be 0 or -1.

## Related Keywords

NOT, OR, XOR

# ARD

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The ARD function interprets the numeric argument as an angle measured in radians, and returns the value of the angle in decimal degrees.


| Item | Description | Range |
|---|---|---|
| `numeric argument` | numeric expression | — |

## Examples

```
Degrees=ARD(Radians)
PRINT ARD(PI*B)
```

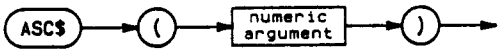
## Related Keywords

`ACS`, `ADS`, `ASN`, `ATN`, `COS`, `DMS`, `PI`, `RAD`, `SIN`, `TAN`

# ASC$

Exists in Development System ■
Works Same in Development System □
Allowed in IF...THEN ■

The ASC$ function converts a numeric value into a string character according to the built-in character set and the user-defined characters.




| Item | Description | Range |
|---|---|---|
| numeric argument | numeric expression, truncated to an integer and modulo 256 to evaluate within the range 0 through 255 | -32,767 through +32,767 |

## Examples

```
PRINT A;B;ASC$(13);C
IF STR$(A$,X,1)=ASC$(10) GOTO 300
```

## Description

ASC$ can be used to include non-displayable characters in strings. Refer to the keyword description for PRINT for a description of display control characters.

An ampersand and the hex representation of a character, &xx, can also be used to include non-displayable characters in a string. An ampersand itself can be included in strings as ASC$(38), &26, or &&. A quotation mark can be included in strings as ASC$(34), &22, or "". A NUL character can only be included in a string as ASC$(0); &00 is not allowed in a string. Because the NUL character is used to terminate strings, if you create a string with a NUL somewhere before the end of the string, all characters after the NUL will be ignored.

**Differences Between Development System and Handheld.** The handheld uses the Roman-8 character set. The development system may use either the Roman-8 or the IBM-compatible character sets as set by the HXCHRSET utility described in chapter 5 of the *Utilities Reference Manual*. The difference between the two character sets occurs in the control codes (ASC$(0) through ASC$(31)) and in the upper half of the character set (ASC$(128) through ASC$(255)).

## Related Keywords

COD

# ASN

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The ASN function returns the arcsine of the numeric argument as a real number in the range -90 through +90 degrees.


| Item | Description | Range |
|---|---|---|
| numeric argument | numeric expression | -1 through 1 |

## Examples

```
Theta=ASN(.5)
PRINT ASN(X*Y)
```

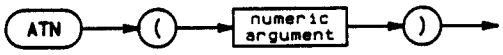
## Related Keywords

ACS, ADS, ARD, ATN, COS, DMS, PI, RAD, SIN, TAN

# ATN

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The ATN function returns the arctangent of the numeric argument as a real number in the range -90 through 90 degrees.




| Item | Description | Range |
|---|---|---|
| `numeric argument` | numeric expression | — |

## Examples

```
Theta=ATN(1)
PRINT ATN(A)
```

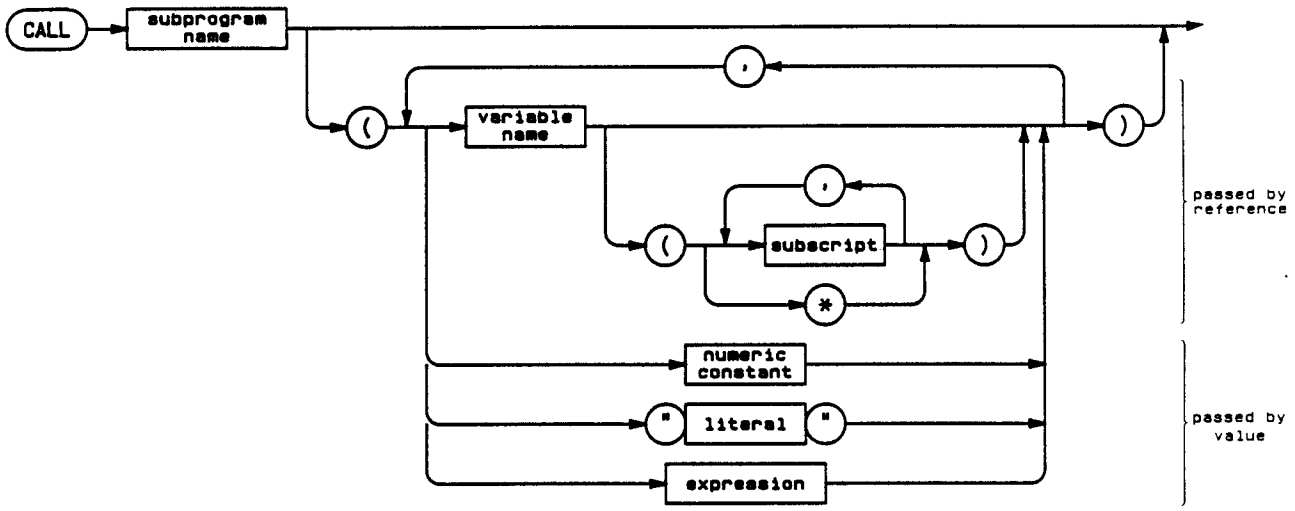
## Related Keywords

`ACS, ADS, ARD, ASN, COS, DMS, PI, RAD, SIN, TAN`

# CALL

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The CALL statement transfers program execution to the specified subprogram and, optionally, passes parameters into the subprogram.




| Item | Description | Range |
|---|---|---|
| subprogram name | unquoted name of BASIC sub-program | any valid file name (directory number not allowed) |
| variable name | name of a simple numeric or string variable | any valid name |
| subscript | numeric expression, truncated to an integer | 0 through 32,767 |
| numeric constant | numeric expression that can contain digits 0 through 9, plus or minus sign, a decimal point, and exponential notation | — |
| literal | string constant | — |
| expression | numeric or string expression | — |

# ...CALL

## Examples

```
CALL EJCT
CALL MENU (Number,String$,Array$(*),Element$(3,7))
CALL SUB1 (3,"test",4.7,(X),(Elem$(3,7)),A*B/2,STR$(A$,4,5))
```

## Description

The CALL statement searches for the designated subprogram. Execution begins when the subprogram is found. Directories 0 through 4 are searched in ascending order.

There are two ways to pass parameters between the calling (sub)program and the called subprogram:

- Parameters can be passed by reference (as illustrated in the second example). The declared precision of numeric variables accompanies them into the subprogram. Changes in the values assigned to the variables are returned to the calling program. Entire arrays or individual array elements can be passed this way.
- Parameters can be passed by value (as illustrated in the third example). Changes in the values assigned to the variables are *local* to the subprogram; they are not transferred back to the calling program. Individual variables or elements of arrays can be passed this way if enclosed in parentheses; entire arrays cannot be passed by value unless they are specified element by element. Numeric and string expressions are only passed by value.

Parameters are passed in the order in which they appear, left to right. The CALL statement must contain the same number and type of parameters as the PARAM statement of the subprogram it calls.

Recursive calls are allowed; a subprogram can call itself. Subprograms can be nested a maximum of 16 levels deep (32 if the subprograms use neither local variables nor parameters). This is limited by the number of scratch areas available for programs to use. User-defined assembly language programs or device handlers may use some of these scratch areas, thereby limiting the subprogram nesting limit.

When END is executed, program execution returns to the statement immediately following the CALL (on the same line, if the CALL is in a multistatement line). CALL cannot be executed in the interrupt-processing routines defined by SYLB or SYSW.

---

**NOTE** The name of the subprogram being called is the same as the name of the file containing the program. Changing the file name (through HXC, for example) can cause the calling program not to find the subprogram.

---

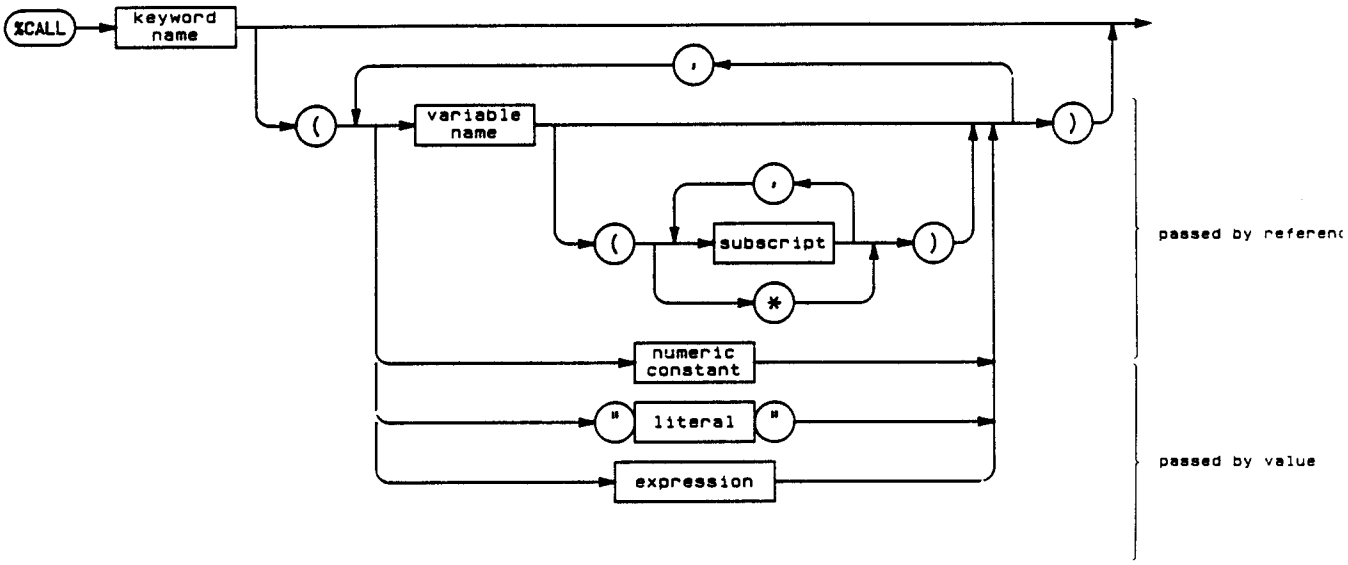
## Related Keywords

END, PARAM

# %CALL

Exists in Development System ■
Works Same in Development System □
Allowed in IF...THEN ■

The %CALL statement executes the specified assembly-language subprogram and, optionally, passes parameters to the keyword.




| Item | Description | Range |
|---|---|---|
| subprogram name | unquoted name of assembly-language subroutine | any valid file name (directory number not allowed) |
| variable name | name of a simple numeric or string variable | any valid name |
| subscript | numeric expression, truncated to an integer | 0 through 32,767 |
| numeric constant | numeric expression that can contain digits 0 through 9, plus or minus sign, a decimal point, and exponential notation | — |
| literal | string constant | — |
| expression | numeric or string expression | — |

# ...%CALL

## Examples

```
%CALL INIT
%CALL PRPT("Name",3)
%CALL MENU (Number,STRING$,Array$(*),Element$(3,7))
%CALL SUB1 (3,"test",4.7,(X),(Elem(3,7)),A*B/2,STR$(A$,4,5))
```

## Description

The %CALL statement searches for the designated assembly language program file (type A). See the introduction of this manual for a list of HP-94 file types. Execution begins when the subprogram is found. Directories 0-4 are searched in ascending order. Built-in keywords are not overridden by %CALL. Do not give your your assembly-language subroutine the name SYRS, for instance.

There are two ways to pass parameters between the calling program and the subprogram.

- Parameters can be passed by reference (as illustrated in the second example). The declared precision of numeric variables accompanies them into the subprogram. Changes in the values assigned to the variables are returned to the calling program. Entire arrays or individual array elements can be passed this way.
- Parameters can be passed by value (as illustrated in the third example). Changes in the values assigned to the variables are *local* to the subprogram; they are not transferred back to the calling program. Individual elements of arrays can be passed this way; entire arrays cannot be passed by value unless they are specified element by element. Numeric and string expressions are only passed by value.

Parameters are passed in the order in which they appear, left to right.

The system keywords SYAL through SYTO are examples of assembly language subprograms executed by %CALL. The reference section in this manual for each of the keywords specifies which parameters can be passed by reference and which can be called by value.

---

**NOTE** The name of the subprogram being called is the same as the name of the file containing the program. Changing the file name (through HXC, for example) can cause the calling program not to find the subroutine.

---

**Differences Between Development System and Handheld.** Assembly-language subprograms for the development system must be in a file whose extension is LIB and which has a different assembly language header than in the handheld's type A file. Refer to the *Technical Reference Manual* for details.

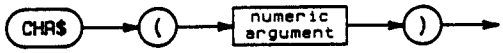
## Related Keywords

CALL

# CHR$

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The CHR$ function evaluates the numeric argument and returns the string representation of the argument in standard number format.




| Item | Description | Range |
|---|---|---|
| `numeric argument` | numeric expression | — |

## Examples

```
amount$="$"+CHR$(D)
PRINT #1,CHR$(Xcoordinate)
```

## Description

CHR$ returns the string representation of the numeric argument. The string returned has a leading blank and a minus sign if the argument is negative. Numbers between -1 and 1 have a leading zero preceding the decimal point. Numbers in exponential notation have E+ or E- between the mantissa and the exponent. The expression `CHR$(1.732)` will return " `1.732`". `CHR$(-.698)` will return " `-0.698`".

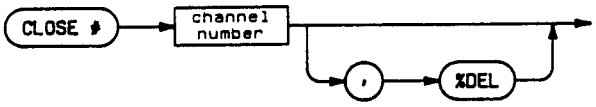
## Related Keywords

NUM

# CLOSE #

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The CLOSE # statement closes a device or data file.




| Item | Description | Range |
|---|---|---|
| channel number | numeric expression, truncated to an integer | 0 through 15 |

## Examples

```
CLOSE #5
IF EOF(file) THEN CLOSE #file, %DEL
CLOSE #1
```

## Description

CLOSE # ends the association between a device or data file and the channel number previously assigned to it by an OPEN # statement. The meaning of each channel number is defined in the table below.

| Channel Number | Meaning |
|---|---|
| 0 | console (keyboard and display) |
| 1 | serial port |
| 2 | bar-code port |
| 3-4 | reserved for future use |
| 5-15 | data file |

The %DEL option deletes the data file associated with the channel number. The %DEL option cannot be used for channels 0-4.

# ...CLOSE #

CLOSE #1 turns off power to the serial port and to the HP 82470A RS-232C Level Converter (if one is connected to the serial port).

**CAUTION** Do not use the statement CLOSE #0 on the development system. It will lock up HXBASIC. On the handheld, CLOSE #0 has no effect.

**Bar-Code Data.** Closing the port to which a bar-code device is attached will power down the device. The HP Smart Wand will power down 240 milliseconds after the port has been closed.

The code segment below illustrates a reset of the HP Smart Wand. It powers down the device, pauses for 240 milliseconds, and then restores power to the Smart Wand.

```
100  CLOSE #2
200  FOR I=1 to 160 : NEXT I
210  REM LOOP WAITS 240 MSECS IF I IS AN INTEGER
300  OPEN #2, "HNBC"
```

## Related Keywords

OPEN #

# COD

Exists in Development System ■
Works Same in Development System □
Allowed in IF...THEN ■

The COD numeric function returns the decimal value of the first character in the string argument.


| Item | Description | Range |
|---|---|---|
| string argument | string expression | — |

## Examples

```
X=COD(String$)
IF COD(A$)=32 GOTO [Skip]
```

## Description

The value returned is in the range 0 through 255. When the argument is the null string, COD returns 0.

**Differences Between Development System and Handheld.** The handheld uses the Roman-8 character set. The development system may use either the Roman-8 or the IBM-compatible character sets as set by the HXCHRSET utility described in chapter 5 of the *Utilities Reference Manual*. The difference between the two character sets occurs in the control codes (ASC$(0) through ASC$(31)) and in the upper half of the character set (ASC$(128) through ASC$(255)).

## Related Keywords

ASC$

# COS

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The COS function interprets the numeric argument as an angle measured in degrees, and returns the cosine of the angle.


| Item | Description | Range |
|---|---|---|
| `numeric argument` | numeric expression | — |

## Examples

```
Y=COS(Angle)
X=R*COS(Theta)
```

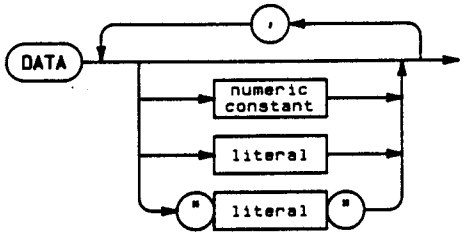
## Related Keywords

ACS, ADS, ARD, ASN, ATN, DMS, PI, RAD, SIN, TAN

# DATA

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN □

The DATA statement contains numeric and/or string data which is assigned to program variables listed in one or more READ statements.




| Item | Description | Range |
|---|---|---|
| numeric constant | a numeric expression that can contain digits 0 through 9, plus or minus sign, a decimal point, and exponential notation | — |
| literal | string constant | — |

## Examples

```
DATA 2,4,6,8
DATA ABC,2.5E20,DEF,3,"   leading spaces"
```

## Description

A program can contain any number of DATA statements. The statement is declaratory, and extra data is ignored if there are no corresponding READ variables. A *data pointer* is used to access data items. A (sub)program's read operations start with the first item in the lowest-numbered DATA statement. When all data items in a DATA statement have been read, the pointer moves to the next-higher numbered DATA statement.

When a READ statement accesses a DATA statement for a numeric variable assignment, the data constant must be a numeric value. When the READ statement is assigning a value to a string variable, the DATA statement can contain a numeric value, an unquoted string, or a quoted string; a numeric value is interpreted as a literal containing digits. Quotation marks are regarded as string delimiters, and are not part of the string. Strings delimited by quotation marks, however, can contain commas and leading

and trailing blanks.

Quotation marks around literals are optional and are not part of the assignment. String data delimited by quotation marks can include any character, including leading and trailing blanks, commas, non-displayable characters (using &*xx*, as in DATA "&06"), quotation marks (using double quotes, as in DATA """"), and ampersands (using &&, as in DATA "&&" ).

If the keyword is not followed by a numeric constant or literal, the statement is interpreted as DATA "" (null string).

When a READ statement is successful (there is a DATA statement containing a data item of the proper type), execution proceeds to the next *statement* after the READ (on the same line, if the READ is in a multistatement line). When a READ statement attempts to read past the last data item in the program, execution proceeds to the next *line* after the READ, and the previous value of the variable being read into remains unchanged. This is similar to the input aborted condition that occurs for the INPUT statement.

Subprograms maintain their own data pointers. When a subprogram is being executed, READ statements access DATA statements within the subprogram, starting with the lowest-numbered DATA statement in the subprogram. When program execution returns to the calling program, read operations resume where they left off before the subprogram was called.

---

**NOTE** DATA statements can be included in multistatement lines only if there are BASIC statements preceding DATA. Anything after a DATA statement in a multistatement line will be treated as unquoted string data. For this reason, comments using the REM statement cannot be added to DATA statements.

---

An interrupt-processing routine defined by SYLB or SYSW has a separate READ data pointer than the one used in the non-interrupt portion of the program. All DATA statements are treated identically, regardless of where they appear in the program (interrupt routine or non-interrupt routine). When the READ statement is executed in an interrupt-processing routine, it will start reading at the first DATA statement in the program, regardless how many data items had been read before the interrupt routine was executed. When the interrupt routine ends, subsequent READ statements in the non-interrupt portion of the program will continue reading DATA statements as if the interrupt routine had not been executed. That is, reading will start after the last data item read before the interrupt routine was executed. In an interrupt routine, RESTORE will affect the interrupt routine's data pointer independently of the non-interrupt routine's data pointer.

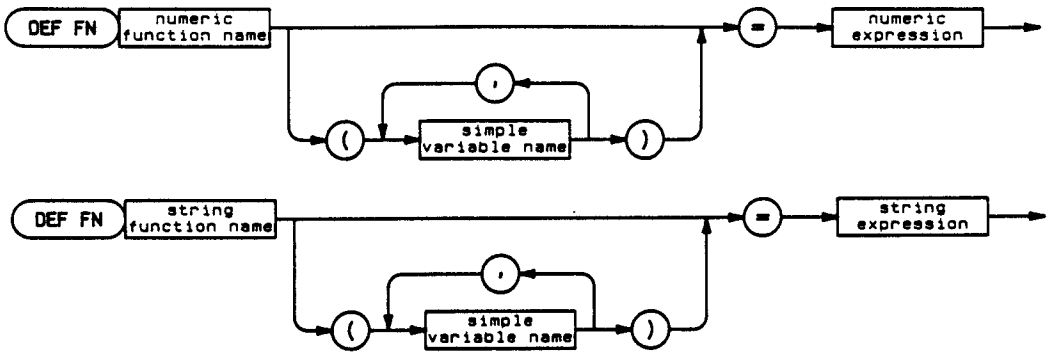
## Related Keywords

READ, RESTORE, SYLB, SYSW

# DEF FN

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN □

The DEF FN statement defines a single-line user-defined function and its formal parameters.




| Item | Description | Range |
|---|---|---|
| numeric function name | name of the user-defined function | any valid simple numeric variable name |
| simple variable name | name of a simple numeric or string variable | array element not allowed |
| numeric expression | (see introduction) | — |
| string function name | name of the user-defined function | any valid simple string variable name |
| string expression | (see introduction) | — |

## Examples

```
DEF FNCube(Number)=Number**3
PRINT FNCube(Side)

DEF FNname$(X$,start)=STR$(X$,start,IDX(X$," ")-start)
PRINT #1, A(1);FNname$(answer$,3)
```

# ...DEF FN

## Description

A maximum of 255 parameters can be passed into the function subject to the 127-characters-per-line limit of HXBASIC. The formal parameters listed in the DEF FN statement must match the actual parameters listed in the calling FN statement in number and type (numeric versus string). The actual parameters are passed into the user-defined function by value. All program variables (except those whose names are the same as formal function parameters) are available in the user-defined function. If a formal parameter has the same name as a variable in the using program, then any reference to that variable in the function will point to the formal parameter.

DEF FN is a declaratory statement; it is ignored if the function is not referenced. It must appear in the program before any FN statements that invoke the function.

Function definitions are local to the program or subprogram in which they are located.

---

**CAUTION** Do not use recursive user-defined functions. A recursive user-defined function will never terminate, and will require that you reset the handheld (by pressing the reset switch) or the development system (by pressing CTRL Alt DEL at the same time), whichever is executing the function.

---

The FN keyword must be in all capital letters.

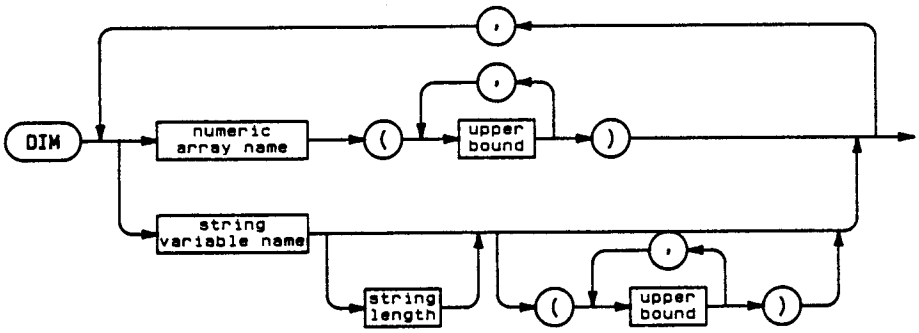
## Related Keywords

FN

# DIM

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN □

The DIM statement allocates memory for real and integer numeric arrays, string variables, and string arrays.




| Item | Description | Range |
|---|---|---|
| numeric array name | name of a numeric array | any valid name |
| upper bound | integer constant | 1 through 32,767 |
| string variable name | name of a simple string variable | any valid name |
| string length | integer constant | 1 through 255 |

## Examples

```
DIM A(300),B(2,50),C$20
DIM D$30(25),E$7(3,3)

INTEGER IntegerArray1,IntegerArray2
DIM IntegerArray1(10),IntegerArray2(5,3)
```

# ...DIM

## Description

A program can contain any number of DIM statements. Only INTEGER, OPTION BASE, PARAM, and REM can appear before DIM.

There can be up to 255 dimensions for numeric or string arrays subject to the 127-characters-per-line limit of HXBASIC. Any array is limited to a maximum of 65,535 bytes.

The default lower bound of the array is 1. The OPTION BASE statement is used to set the lower bound equal to 0.

DIM must be executed before any of the elements of an array are referenced. There is no automatic dimensioning of arrays. If an element is referenced before the array is dimensioned, an error will occur. If a string variable is referenced before the string length is dimensioned, the default string length is 8.

A variable can be dimensioned only once within a program; an attempt to dimension a variable that has already been dimensioned causes an error.

The dimension(s) of a variable are global, known to the program and any subprograms to which the variable is passed.

All numeric variables are real unless declared integer with an INTEGER statement.

---

**NOTE** Pay special attention to the syntax used for dimensioning strings and string arrays - the dimensioned length immediately after the variable name, and then the array bounds in parentheses. This is different than in many BASIC languages.

---

The following tables provide reference information about the different BASIC variable types. Under "Memory Usage" for arrays, the total number of elements is the product of the number of elements in each dimension of the array. If the statement OPTION BASE 0 appears, the number of elements in each dimension is the specified upper limit plus 1. Because memory is allocated in *paragraphs*, or blocks of 16 bytes, the total memory required for all variables and arrays in a program will be the calculated memory usage rounded up to the next higher 16-byte boundary.

**Real Numeric Variables**

| Description | Value |
|---|---|
| Initial Value | 0 |
| Numeric Precision | 14 Decimal Digits |
| Exponent Range | -64 to +63 |
| Maximum Array Size (bytes) | 65,535 |
| Maximum No. of Dimensions | 255 |
| Maximum No. of Elements | 8,191 (1 dimension) to 8,128 (255 dimensions) |
| Memory Usage (bytes) | |
| Simple Variable | 8 |
| Array | 8 * (no. of elements) + 2 * (no. of dimensions) + 1 |

**Integer Numeric Variables**

| Description | Value |
|---|---|
| Initial Value | 0 |
| Range | -32,768 to +32,767 |
| Maximum Array Size (bytes) | 65,535 |
| Maximum No. of Dimensions | 255 |
| Maximum No. of Elements | 32,766 (1 dimension) to 32,512 (255 dimensions) |
| Memory Usage (bytes) | |
| Simple Variable | 2 |
| Array | 2 * (no. of elements) + 2 * (no. of dimensions) + 1 |

**String Variables**

| Description | Value |
|---|---|
| Initial Value | null string |
| Default Maximum Length | 8 characters |
| Possible Maximum Length | 255 characters |
| Character Range | Any ASCII character (01-FFh) except null (00h), which marks the end of the string |
| Maximum Array Size (bytes) | 65,535 |
| Maximum No. of Dimensions | 255 |
| Maximum No. of Elements | 32,767 (1 dimension, character length 1) to 254 (255 dimensions, character length 255) |
| Memory Usage (bytes) | |
| Simple Variable | dimensioned length |
| Array | (dimensioned length) * (no. of elements) + 2 * (no. of dimensions) + 1 |

## Related Keywords

INTEGER, OPTION BASE, PARAM

# DMS

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The DMS function interprets the numeric argument as an angle measured in decimal degrees, and returns the value of the angle in degrees, minutes, and seconds.


| Item | Description | Range |
|---|---|---|
| numeric argument | numeric expression | — |

## Examples

```
Degminsec=DMS(Decdeg)
Newtime=DMS(ADS(Oldtime)+ADS(.0005))
```

## Description

The format of the value returned is *DD.MMSSxxxxxx* as shown in the table below.

| Item | Description | Range |
|---|---|---|
| *DD* | Degrees. | — |
| *MM* | Minutes. | — |
| *SS* | Integer seconds. | — |
| *xx* | Fractional seconds | — |

DMS can also be used to convert decimal hours into hours, minutes, and seconds.

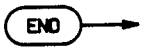
## Related Keywords

ADS, TIM

# END

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The END statement returns program execution to the calling (sub)program or halts main program execution.




## Examples

```
END
IF A<0 THEN PRINT "Done" : END
```

## Description

When END is executed in a subprogram, program execution resumes at the statement in the calling program that immediately follows the `CALL` statement. Local variable space required by the subprogram is released. When END is executed in a main program, program execution halts.

The END statement can appear anywhere in a program. More than one END statement is allowed. END is required as the last line of a program or subprogram unless the last line is either GOTO, RETURN, or `%CALL SYRT`.

**NOTE** If the END statement is executed in an interrupt-processing routine defined by `SYLB` or `SYSW`, the program or subprogram will end and return control back to the operating system (not to the calling (sub)program).

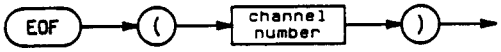
## Related Keywords

`CALL`

# EOF

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The EOF function returns a value indicating the current data access status.




| Item | Description | Range |
|---|---|---|
| channel number | numeric expression, truncated to an integer | 5 through 15 |

## Examples

```
IF EOF(5) THEN PRINT E$
ON EOF(channel)+2 GOTO 50,100
```

## Description

EOF examines the data file associated with the specified channel number. If the most recent serial or random access read operation did not read past either the end of the data in the file (EOD) or the end of the file itself (EOF), EOF returns 0 (false). If the most recent serial or random access read operation *did* read beyond either EOD or EOF, EOF returns -1 (true).

The value returned by EOF does not change because of any operations except serial or random reads (serial reads using GET #, INPUT #, or INPUT$; random reads using GET #).

## Related Keywords

GET #, INPUT #, INPUT$

# EXP

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The EXP function returns the natural (base *e*) antilogarithm by raising *e* to the power of the argument.


| Item | Description | Range |
|---|---|---|
| `numeric argument` | numeric expression | — |

## Examples

```
K=A*EXP(-E/RT)
PRINT A;EXP(A)
```

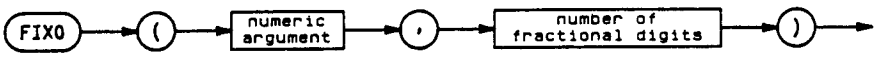
## Related Keywords

LGT, LOG

# FIX0

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The FIX0 function returns the numeric argument, rounded *down* zero to the specified number of digits after the decimal point.




| Item | Description | Range |
|---|---|---|
| numeric argument | numeric expression | — |
| number of fractional digits | numeric expression, truncated to an integer | 0 through 63 |

## Examples

```
Y=FIX0(1/3,4)
PRINT FIX0(X,N)
```

## Description

Regardless of the arguments to this function, the maximum number of significant digits is 14.

Numbers are rounded towards zero, so negative numbers become "less negative." FIX0(-1.5) will return -1.

For numbers with negative exponents, the number of fractional digits for rounding purposes may be greater than 14 because the value of the argument is relative to the decimal point of the unexponentiated representation of the number to be rounded. For example, to round 0.000087854321 (or 8.7654321E-5) down at the digit "4", use FIX0(0.000087654321,9). The parameter 9 is the sum of the absolute value of the negative exponent (5) and the desired number of fractional digits (4).

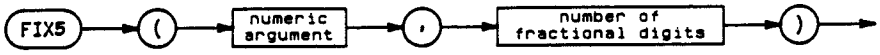
## Related Keywords

FIX5, FIX9, FIXE

# FIX5

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The `FIX5` function returns the numeric argument, rounded off to the nearest value at the specified number of digits after the decimal point.




| Item | Description | Range |
|---|---|---|
| `numeric argument` | numeric expression | — |
| `number of fractional digits` | numeric expression, truncated to an integer | 0 through 63 |

## Examples

```
Y=FIX5(1/3,4)
PRINT FIX5(X,N)
```

## Description

`FIX5` examines digit *n*+1 after the decimal point, where *n* is the number of fractional digits specified. If digit *n* +1 is 4 or less, `FIX5` rounds the argument down. If digit *n* +1 is 5 or greater, `FIX5` rounds the argument up.

Regardless of the arguments to this function, the maximum number of significant digits is 14.

For numbers with negative exponents, the number of fractional digits for rounding purposes may be greater than 14 because the value of the argument is relative to the decimal point of the unexponentiated representation of the number to be rounded. For example, to round 0.000087854321 (or 8.7654321E-5) at the digit "4", use `FIX5(0.000087654321,9)`. The parameter 9 is the sum of the absolute value of the negative exponent (5) and the desired number of fractional digits (4).

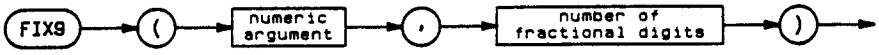
## Related Keywords

FIX0, FIX9, FIXE

# FIX9

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The `FIX9` function returns the numeric argument, rounded up to the specified number of digits after the decimal point.




| Item | Description | Range |
|---|---|---|
| `numeric argument` | numeric expression | — |
| `number of fractional digits` | numeric expression, truncated to an integer | 0 through 63 |

## Examples

```
Y=FIX9(1/3,4)
PRINT FIX9(X,N)
```

## Description

Regardless of the arguments to this function, the maximum number of significant digits is 14.

Numbers are rounded away from zero, so negative numbers become "more negative." `FIX9(-1.5)` will return -2.

For numbers with negative exponents, the number of fractional digits for rounding purposes may be greater than 13 because the value of the argument is relative to the decimal point of the unexponentiated representation of the number to be rounded. For example, to round 0.000087854321 (or 8.7654321E-5) up at the digit "4", use `FIX9(0.000087654321,9)`. The parameter 9 is the sum of the absolute value of the negative exponent (5) and the desired number of fractional digits (4).

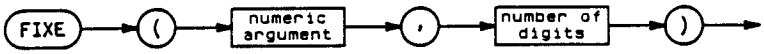
## Related Keywords

`FIX0`, `FIX5`, `FIXE`

# FIXE

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The FIXE function returns the numeric argument, treated as a number in scientific notation, whose mantissa is rounded off to the nearest value with the specified number of digits after the decimal point.




| Item | Description | Range |
|---|---|---|
| numeric argument | numeric expression | — |
| number of digits | numeric expression, truncated to an integer | 0 through 63 |

## Examples

```
Y=FIXE(1/3,4)
PRINT FIXE(X,N)
```

## Description

FIXE treats the argument as a number in scientific notation and examines digit $n+1$ after the decimal point, where $n$ is the number of digits specified. If digit $n+1$ is 4 or less, FIXE rounds the argument down. If digit $n+1$ is 5 or greater, FIXE rounds the argument up.

If the number of digits is greater than or equal to 13, FIXE returns the argument unchanged.

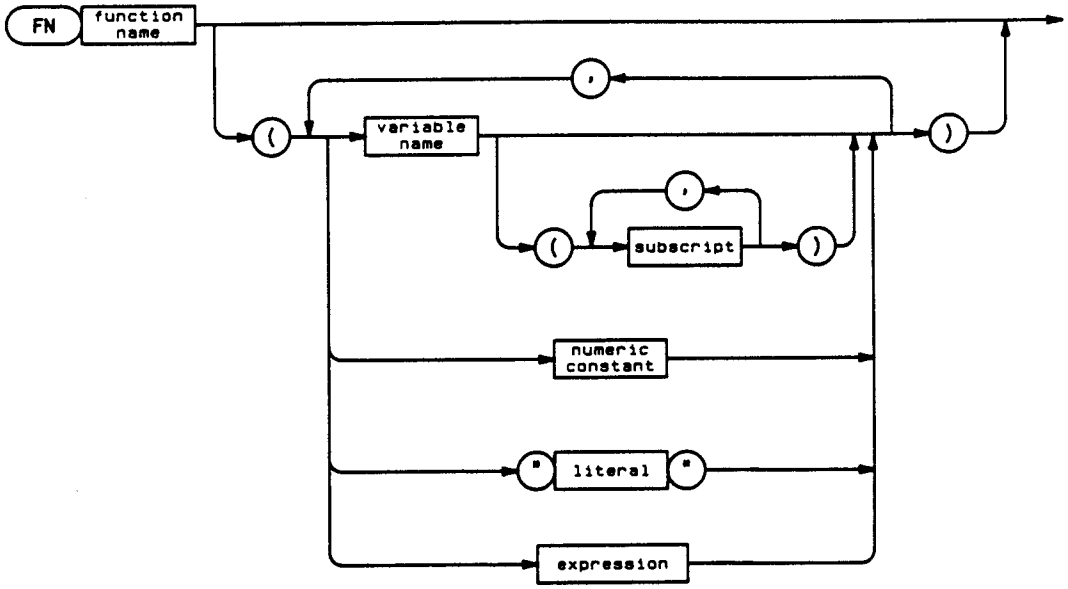
## Related Keywords

FIX0, FIX5, FIX9

# FN

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The FN keyword is a prefix used before the name of a user-defined function to identify a call to the function. Optional parameters in parentheses are passed to the function. The function returns a value used by the expression containing the function call.




| Item | Description | Range |
|---|---|---|
| `function name` | name of the user-defined function | any valid simple numeric or string variable name |
| `variable name` | name of a simple numeric or string variable | any valid name |
| `subscript` | numeric expression, truncated to an integer | 0 through 32,767 |

| Item | Description | Range |
|---|---|---|
| numeric constant | numeric expression that can contain digits 0 through 9, plus or minus sign, a decimal point, and exponential notation | — |
| literal | string constant | — |
| expression | numeric or string expression | — |

## Examples

```
Y=FNInverse/A
B$=A$+FNstar$
```

## Description

When FN invokes a user-defined function, the function type (numeric versus string) must match the context of the expression invoking the function. For example, the value returned by a string function cannot be assigned to a numeric variable.

The definition of the invoked function (DEF FN statement) must precede the use of the function.

The parameters passed into a user-defined function by FN must match the DEF FN parameter list in number and type (numeric versus string). The parameters are passed by value. Numeric and string variables, elements of numeric and string arrays, substrings, numeric constants and literals, and numeric and string expressions can be passed to a function.

**CAUTION** Do not use recursive user-defined functions. A recursive user-defined function will never terminate, and will require that you reset the handheld (by pressing the reset switch) or the development system (by pressing Ctrl Alt Del at the same time), whichever is executing the function.

The FN keyword must be in all capital letters.

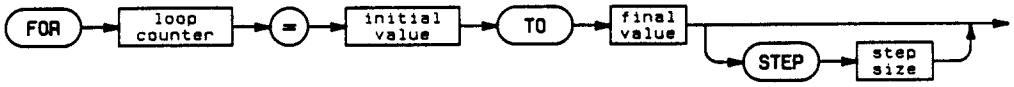
## Related Keywords

DEF FN

# FOR...NEXT

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The FOR and NEXT statements together comprise a program loop that is repeated until a loop counter passes a specified value.





| Item | Description | Range |
|---|---|---|
| `loop counter` | simple numeric variable name | array element not allowed |
| `initial value` | numeric expression | — |
| `final value` | numeric expression | — |
| `step size` | numeric expression (default = 1) | — |

## Examples

```
FOR Counter=1 TO 100
    PRINT Counter
NEXT Counter

FOR I=N TO N+M STEP stepsize
    A(I)= .592*ABS(I**3)
    IF A(I)>X GOTO 400
    PRINT I; A(I)
NEXT I
```

## Description

The FOR statement defines the beginning of the loop, sets the loop counter equal to the initial value, and stores the final value and step size. Each time the NEXT statement is executed, the loop counter is

incremented (or decremented, in the case of a negative step value) by the step value and then compared to the final value. If the final value has not been passed, program execution is transferred to the statement immediately following the FOR statement. If the final value has been passed, program execution continues with the line immediately following the NEXT statement. (The loop counter is not equal to the final value when the loop has been ended.)

**NOTE** Because the loop counter is not tested until after the NEXT statement is executed (see flowchart), the loop is always executed once, even if the loop counter initial value is already past the final value. For example, a loop beginning with the statement `FOR I=3 TO 5 STEP -.3` will be executed once with I equal to 3. When the `NEXT I` statement is executed, I will be decremented by 0.3, and the loop will terminate (with I equal to 2.7), since 2.7 is already past (less than) the final value 5.

If the loop counter has been declared as `INTEGER`, the loop counter control values (initial value, final value, and step size) will be truncated to integers. This may result in unexpected behavior. For the previous example (`FOR I=3 TO 5 STEP -.3`), if `I` has been declared `INTEGER`, the step size will be truncated to 0, and the loop will never terminate.

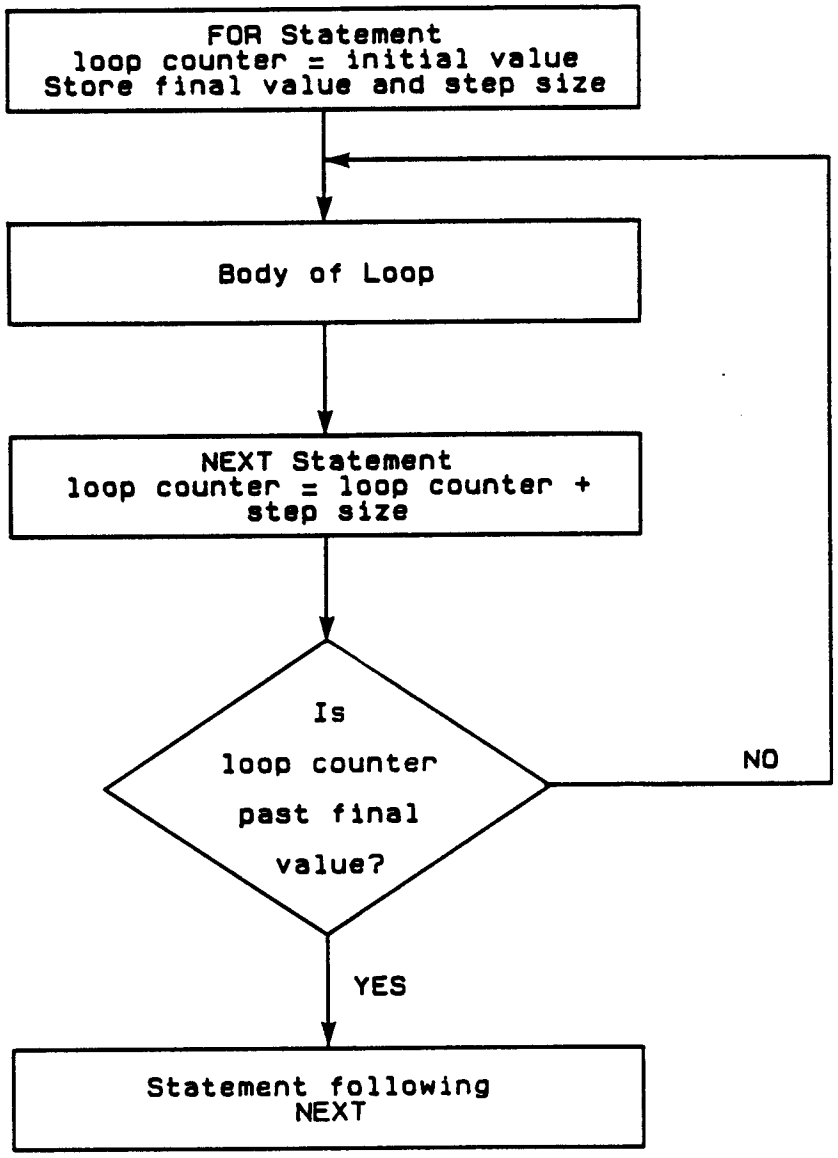
The loop can be ended by unconditional or conditional branching; the loop counter retains its current value. The loop may be re-entered in the body of the loop or at the FOR statement. Entering a loop at the FOR statement reinitializes the loop counter.

The FOR statement stores the loop counter, final value, and step size, and these values remain unchanged for the loop until the FOR statement is executed again. When the loop counter, final value, and step size are numeric expressions containing variables, the values of those variables can be changed within the loop without affecting how many times the loop is executed. However, changing the value of the loop counter within the loop can affect how many times the loop is executed. The loop counter can be used in expressions defining the initial value, final value, and step size.

Each FOR statement can have one, and only one, matching `NEXT` statement. When `FOR...NEXT` loops are nested, one loop must be contained entirely within another.

## Related Keywords

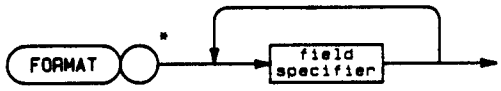
None.


FOR Statement
loop counter = initial value
Store final value and step size
Body of Loop
NEXT Statement
loop counter = loop counter +
step size
Is
loop counter
past final
value?
NO
YES
Statement following
NEXT

# FORMAT

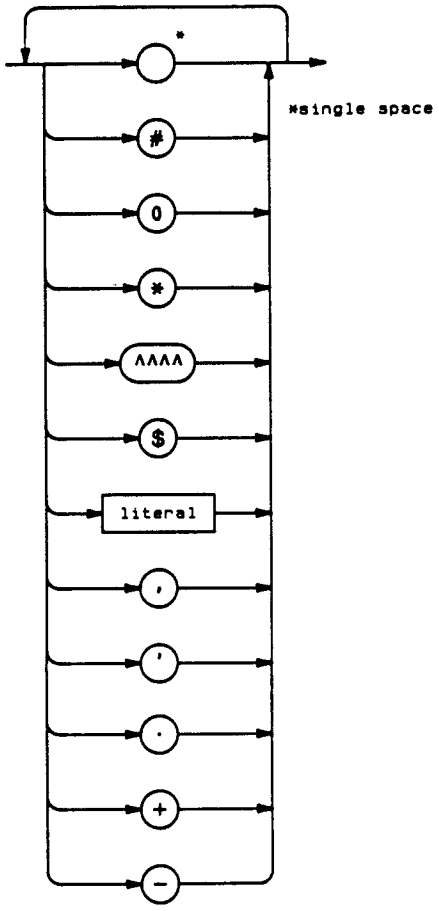
Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The FORMAT statement specifies a format string referenced by PRINT USING and PRINT # ...USING. The format string contains one or more field specifiers that describe the format of the data to be output.




*single space

field specifier



# ...FORMAT

| Item | Description | Range |
|---|---|---|
| `field specifier` | one or more format specifier characters | — |
| `literal` | string | cannot contain format specifier characters |

## Examples

```
FORMAT ##,###
FORMAT Price= $$$.##
FORMAT +++  ---
```

## Description

The format string begins after the first space after the keyword FORMAT. The format string consists of one or more *field specifiers* placed together in the format string. Items in a PRINT USING or PRINT #...USING statement are paired with their corresponding field specifiers from left to right. Certain field specifiers do not use a PRINT or PRINT # item (for example, a literal).

A field specifier consists of one or more *format specifiers*. The format specifiers within a field specifier describe the format of one output item. Items can be numeric or string expressions.

The end of a field specifier is defined as the place within the format string where two different format specifiers are placed adjacent to one another. Exceptions to this rule are the format specifiers $, ,, ', ^^^^, ., +, and −, which can be interpreted as part of another field specifier. If the format string is exhausted before the entire list of items is output, the format string is reused from the beginning. Extra field specifiers are ignored.

For numeric fields, if a field specifier is larger than the item, the number is right-justified in the field. A format overflow occurs when a numeric item requires more digit spaces to the left of the decimal point than are specified. The overflow causes the field specifier (not the associated numeric item) to be output. If a numeric item contains more decimal places than the field specifier, the number is truncated to fit the field. No rounding occurs in displaying numeric items.

For string items, if a field specifier is larger than the item, the item is left-justified in the field. If a string item requires more character spaces than are specified, the field is filled with the left-most characters of the item. The right-most characters that do not fit in the print field are not printed. Notice that you can put literals in a FORMAT statement, but you cannot put field or format specifiers in a PRINT USING or PRINT #...USING statement. This is different than many BASIC languages.

FORMAT statements are declaratory; they are ignored if they are not referenced.

The table below describes each format specifier.

**Format Specifiers for** PRINT USING **and** PRINT #...USING

| Format Specifier | Meaning |
|---|---|
| (space) | Outputs a blank space. |
| # | String character position or numeric digit position to left or right of the radix symbol. If the field to the left of the radix is larger than the number, the number is right-justified with leading blanks. You must supply a # for the sign position for negative numbers. Numbers between -1 and 1 must have at least one # to the left of the radix. # is the only format specifier that can specify a string character position. |
| 0 | Digit position to left of the radix symbol. If the field to the left of the radix is larger than the number, the number is right-justified with leading zeros. |
| * | Digit position to left of the radix symbol. If the field to the left of the radix is larger than the number, the number is right-justified with leading asterisks. |
| ^^^^ | Exponential format; exponent consists of character E with a sign and two digits. ^ characters in excess of four are treated as a literal; fewer than four cause an error. You must precede the exponential format with one # for the sign position of the number, and one # for each digit of the mantissa to be displayed. |
| $ | Digit position to left of the radix symbol. If the field to the left of the radix is larger than the number, $ is right-justified with leading blanks. |
| literal | String consisting of any characters that are not image specifiers. |
| , (comma) | Digit separator; places a comma in that position. Comma is output only if digits on both sides of the separator are output. |
| ' (single quote) | Digit separator; places a single quote in that position. Quote is output only if digits on both sides of the separator are output. |
| . (period) | Radix symbol; specifies a decimal point and places a period in that position. Period is output only if digits on both sides of the separator are output. |
| + | Sign and digit position to left of radix symbol; outputs – if negative, + if positive. If the field to the left of the radix is larger than the number, sign is right-justified with leading blanks. |
| - | Sign and digit position to left of radix symbol; outputs – if negative, blank if positive. If the field to the left of the radix is larger than the number, sign is right-justified with leading blanks. |

## Related Keywords

PRINT USING, PRINT #...USING

# FRC

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The FRC function returns the fractional part of the numeric argument. The function returns a value between -1 and 1. A negative argument returns a negative value.


| Item | Description | Range |
|---|---|---|
| `numeric argument` | numeric expression | — |

## Examples

```
Y=FRC(X+1.23)
IF FRC(X)=0 THEN PRINT "X is an integer"
```

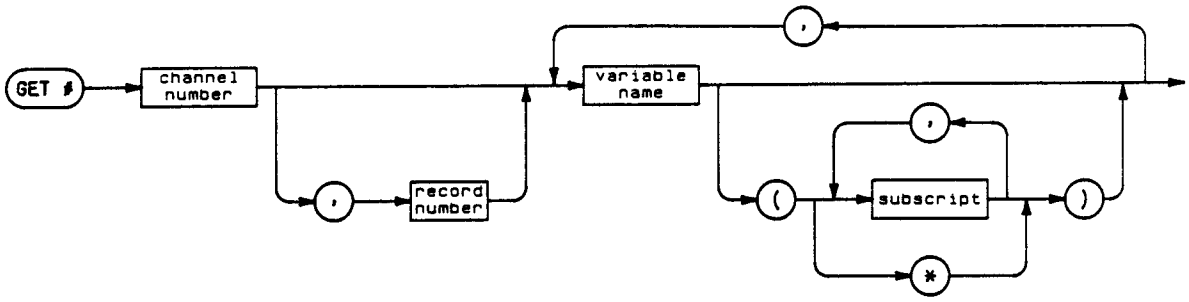
## Related Keywords

`INT`

# GET #

Exists in Development System ■
Works Same in Development System □
Allowed in IF...THEN ■

The GET # statement inputs items from the specified device or data file.




| Item | Description | Range |
|---|---|---|
| channel number | numeric expression, truncated to an integer | 0 through 15 |
| record number | numeric expression, truncated to an integer | 1 through 32,767 |
| variable name | name of a simple numeric or string variable | any valid name |
| subscript | numeric expression, truncated to an integer | 0 through 32,767 |

## Examples

```
GET #5 Height, Width, Length
GET #channel Stats(*)
GET #chno,recno Potential, Unit$
GEt #1 barcode$
```

## Description

The GET # statement inputs data from the device or data file associated with the specified channel number. All devices (except channel 0) and data files must be opened with OPEN # before they can be accessed with GET #. When a data file is opened, an associated file access pointer is positioned at the beginning of the file. The counterpart to GET # for output operations is PUT #.

GET # inputs data until each variable in the input list is "full"; that is, until the number of bytes defined by the type of each variable has been input. For example, GET # inputs 10 bytes of data to fill a simple string variable dimensioned to 10 characters. See the section on bar-code data for an exception to this rule. The table below shows the size of each type of variable.

**Sizes of Different Variables**

| Variable Type | Size (bytes) |
|---|---|
| Real variable | 8 |
| Integer variable | 2 |
| String variable | dimensioned length |
| Real array | 8 * (number of elements in each dimension) |
| Integer array | 2 * (number of elements in each dimension) |
| String array | (dimensioned length) * (number of elements in each dimension) |

An entire array can be input by including the * subscript option with a variable name. Only one asterisk is required regardless of the number of dimensions of the array.

If input is from a data file, the optional record number determines whether serial or random access will be performed. The record number has meaning only when reading from data files.

**Serial Access.** When the record number is omitted, serial access is performed. Each record is read from the location in the file immediately after the previous record (serially).

**Random Access.** When the record number is included, random access is performed. Each record is read from the location in the file specified by the record number (randomly).

When the GET # statement is executed, the file pointer is positioned to (*record size*) * (*record number* - 1) bytes from the beginning of the file, where *record size* is the total size (in bytes) of all the variables in the input list.

---

**CAUTION** The definition of a *record* is totally arbitrary; GET # does not look for end-of-record markers within a data file, nor does its counterpart for writing, PUT #, place any end-of-record markers in a file. Therefore, the application program is responsible for maintaining a suitable data file structure. The way data will be read is dependent solely on the lengths of the variables in the input list, regardless of how the data was originally written to the file.

---

As each input variable is processed, the file access pointer advances beyond the current data item to the next byte in the file. When input ends, the access pointer remains positioned after the last data item read. Subsequent file I/O statements that perform serial access continue reading data from or writing data to that position.

Because file access is controlled by the lengths of the variables in the input list, the simplest data file structure will have records that are the same fixed length, even though an entire record may be read by using several variables of different lengths. For a file structure using variable-length records, careful selection of variable lengths in different GET # or PUT # statements will move the file access pointer to different parts of a data file. In addition, the SYPT statement can be used to move the

pointer to the desired position.

Input continues until all input variables have been assigned data, or until input ends because of one of the conditions described in the next table.

**Behavior When GET # Ends**

| Ending Condition | Channel 0 Behavior | Channels 1-4 Behavior | Channels 5-15 Behavior |
|---|---|---|---|
| Number of characters defined by the size of the input variable received (normal condition). | Characters typed on the key are placed in the input variable. | Characters received from the device are placed in the input variable. | Characters read from the file are placed in the input variable. |
| Port terminate character received (see SYBC, SYRS, or SYSP). | N/A. | Ends input for that variable. | N/A. |
| EOD or EOF encountered. | N/A. | N/A. | Characters read up to the EOD or EOF are placed in the input variable. All subsequent variables in the input list are set to 0 or the null string. |
| Timeout (error 118). | Input aborted. | Input aborted. | N/A. |
| Power switch pressed (error 119). | Input aborted. | Input aborted. | N/A. |
| Low battery (error 200). | Input aborted. | Input aborted. | N/A. |
| Port errors 201-208. | N/A. | Input aborted. | N/A. |
| Key abort (see SYBC or SYSP) | N/A. | Input aborted. | N/A. |

Note: N/A means the ending condition will not occur for those channels.

**Input Aborted.** In the above table, "input aborted" means that the input operation has been interrupted. When input is aborted, the input operation is ended, and any characters received up to that point are placed in the input variable. This may result in part of the previous value of the variable being overwritten. All subsequent variables in the input list are unchanged. This is in contrast to INPUT, INPUT #, and INPUT$, in which any received data for that variable is discarded. When input is aborted for GET #, program execution continues on the next line of the program (not on the next statement, if GET # is in a multistatement line). When input is aborted because of a numeric error, the I/O length reported by %CALL SYIN is set to the number of bytes actually received up to that point, since that data has already been placed in the input variable.

Specifying more than one variable in the GET # statement is not recommended for channels 0 through 4. If input is aborted for any reason, the calling program will not be able to identify which variable was being loaded at the time input was aborted.

**Bar-Code Data.** The GET # command is the recommended keyword for reading data from a bar-code reader. Input must be obtained from either the bar-code port (channel 2) or the serial port (channel 1). Be sure the channel has been opened with the correct bar-code handler (see OPEN # for details). The keywords SYBC, SYSP, and SYWN allow configuration of the port to which the bar-code reader is connected.

The variable into which the bar-code data is loaded should be longer than the longest string expected from the bar-code device. The longest message sent by the HP Smart Wand is its configuration dump message (223 bytes with the default trailer string ↵). The longest standard bar code decodes to 64 ASCII characters plus any termination characters sent by the bar-code device. If the string variable is dimensioned shorter than the data input, the bar-code handler will fill the variable and then pass control to the next statement in the program. The remaining bar-code data will be returned by the next GET # statement to the port. If the string variable is dimensioned longer than the bar-code message (this is the recommended procedure), the GET # statement will end after a 104 millisecond pause occurs between characters sent from the bar-code device wand to the handheld. This behavior is unique to the bar-code handlers supplied with the Software Development System. The built-in serial-port handler (configured with the SYRS keyword) will wait until either the current timeout expires or all characters of the input variable have been received, whichever occurs first.

Avoid bar-code labels containing null characters (ASC$(0)). The handheld uses the null character to terminate a string. Characters following a null will be ignored by the handheld.

The handheld is unable to receive data through its serial port while it is reading data from the bar-code port. This is true even if the OPEN statement is used for the serial port. It is recommended that only one of the two ports be open at any given time. If both ports must be open, send an XOFF (ASC$(19)) to the serial port before reading data from the bar-code port.

Escape sequences that may be received from the HP Smart Wand are described with the SYWN keyword.

**Differences Between Development System and Handheld.** The development system does not produce errors 118, 119, 200, or 201 through 208.

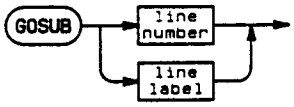
## Related Keywords

INPUT #, INPUT$, PUT #, SYBC, SYIN, SYPT, SYRS, SYSP

# GOSUB

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The GOSUB statement causes program execution to branch to the the specified line and return to the statement following the GOSUB when a RETURN is encountered.




| Item | Description | Range |
|---|---|---|
| line number | integer constant identifying a program line | 0 through 32,767 |
| line label | name of a program line | any valid name |

## Examples

```
GOSUB 760
GOSUB [marine]
```

## Description

The specified line must be in the same program or subprogram as the GOSUB statement. If the specified statement is declaratory (for example, DIM, DATA, or REM), the program branches to the next executable statement.

When GOSUB is executed, execution of the subroutine continues until a RETURN statement is encountered. The RETURN causes branching to the statement following the GOSUB or ON...GOSUB (on the same line, if the GOSUB or ON...GOSUB is in a multistatement line).

Subroutines can be recursive; i.e., a subroutine can invoke itself.

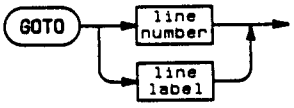
## Related Keywords

GOTO, ON...GOSUB, ON...GOTO, RETURN

# GOTO

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The GOTO statement causes program execution to branch unconditionally to the specified line.




| Item | Description | Range |
|---|---|---|
| `line number` | integer constant identifying a program line | 0 through 32,767 |
| `line label` | name of a program line | any valid name |

## Examples

```
GOTO 340
GOTO [Increment]
IF Happy GOTO [Smile]
```

## Description

The specified line must be within the same program or subprogram as the GOTO statement. If the specified statement is declaratory (for example, DIM, REM, or DATA), the program branches to the next executable statement.

When GOTO is used as the THEN condition in an `IF...THEN` statement, the THEN keyword can be omitted.

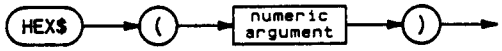
## Related Keywords

GOSUB, `IF...THEN`, `ON...GOSUB`, `ON...GOTO`

# HEX$

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The HEX$ function returns a two-character string containing the base 16 representation of the decimal argument.




| Item | Description | Range |
|---|---|---|
| numeric argument | numeric expression, truncated to an integer and modulo 256 to evaluate within the range 0 through 255 | -32,768 through 31,767 |

## Examples

```
PRINT HEX$(COD(A$))
IF HEX$(I(5))="A4" THEN J=12
```

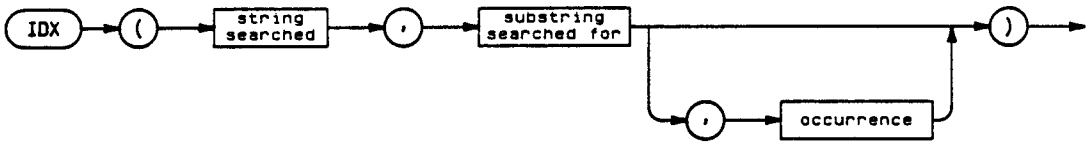
## Related Keywords

None.

# IDX

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The `IDX` function returns the position of the first character of a substring within another string.




| Item | Description | Range |
|---|---|---|
| `string searched` | string expression | — |
| `substring searched for` | string expression | — |
| `occurrence` | numeric expression, truncated to an integer (default = 1) | -32,768 through 32,767 |

## Examples

```
Index=IDX(A$,"1")
Index2=IDX("12341234","1",2)
Index4=IDX(A$,"1",4)
PRINT STR$(choice$,IDX(choice$,answer$)+1)
```

## Description

`IDX` finds a substring within another string, and returns the position of the first character of the located substring. If the substring searched for occurs in more than one place, the occurrence parameter allows you to specify which occurrence is returned. `IDX` will then return the character position of the specified occurrence of the substring.

If the substring searched for is the null string or is not contained within the string searched, `IDX` returns 0. `IDX` also returns 0 if the occurrence specified is less than 1.

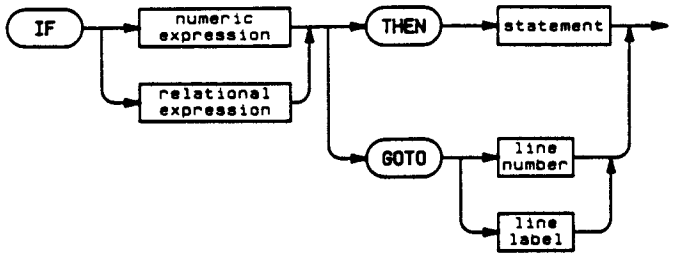
## Related Keywords

STR$

# IF...THEN

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The `IF...THEN` statement causes conditional branching, based on the value of a relational or numeric expression.




| Item | Description | Range |
|---|---|---|
| `numeric expression` | evaluated as true if non-zero and false if zero | — |
| `relational expression` | an expression comparing two numeric or string expressions using relational operators (=, <, >, <=, >= or <>). | — |
| `statement` | a programmable statement "Allowed in `IF...THEN`" | refer to individual keywords |
| `line number` | integer constant identifying a program line | 0 through 32,767 |
| `line label` | name of a program line | any valid name |

## Examples

```
IF SIN(Angle) THEN GOSUB [DrawLine]
IF Variable<5 GOTO 200
IF errorcode=103 THEN IF status=4 GOTO 750
```

## Description

When the expression following IF evaluates as true (non-zero), the portion of the statement following THEN is executed. When the expression following IF is false, program execution proceeds to the next line.

When GOTO or GOSUB is used as the THEN condition in an IF...THEN statement, the THEN keyword can be omitted.

THEN can be followed by:

- An executable statement. The statement must be one whose "Allowed in IF...THEN" square in the legend is filled in (■). If the executable statement is a GOSUB statement, the subroutine RETURN statement returns execution to the statement immediately following the GOSUB (on the same line, if the GOSUB is in a multistatement line).
- A sequence of statements concatenated with :.
- Another IF...THEN statement.

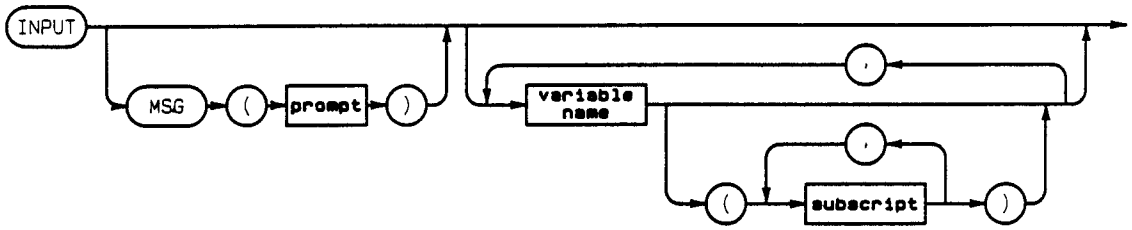
## Related Keywords

GOTO, GOSUB

# INPUT

- ■ Exists in Development System
- □ Works Same in Development System
- ■ Allowed in `IF...THEN`

The `INPUT` statement is used to assign values entered from the keyboard to program variables.




| Item | Description | Range |
|---|---|---|
| `prompt` | string expression | — |
| `variable name` | name of a simple numeric or string variable | any valid name |
| `subscript` | numeric expression, truncated to an integer | 0 through 32,767 |

## Examples

```
INPUT Height, Width, Length, Other(2,3)
INPUT MSG("Your Name: ") Name$
```

## Description

The `INPUT` statement causes program execution to halt until a value has been entered from the keyboard for each input item. Input items are separated by pressing the [ENTER] key. If no prompt is specified, `INPUT` displays the default prompt `?` for each input item. You can edit each input item with the [←] and [CLEAR] keys before pressing [ENTER]. When [ENTER] is pressed after the last input item, program execution continues with the statement after `INPUT` (on the same line, if the `INPUT` is in a multistatement line). If nothing was entered on the keyboard, the input variable remains unchanged, and program execution continues on the next line of the program (see "Input Aborted").

If a prompt is supplied with the `MSG` option, then the prompt string replaces `?` as the input prompt for each input item. All prompting can be suppressed by specifying a null string prompt (`""`).

Individual items must match the specified `INPUT` variable(s) in type (numeric or string). If you attempt to enter alphabetic characters into a numeric variable, the error prompt `??` will be displayed,

and you will be prompted again for the input. The input statement can include simple numeric and string variables and numeric and string array elements. Entries from the keyboard can include numbers and character strings.

If no variable list is supplied, INPUT will read and echo characters from the keyboard until [ENTER] is pressed. The received data will be discarded and program execution will continue.

Input continues until all input variables have been assigned data, or until input has ended because of one of the conditions described in the next table.

**Behavior When INPUT Ends**

| Ending Condition | Behavior |
|---|---|
| [ENTER] key pressed | Characters typed on the keyboard (except [ENTER]) are placed in the input variable. Input is aborted if nothing is typed before [ENTER] is pressed. |
| Timeout (error 118) | Input aborted. |
| Power switch pressed (error 119) | Input aborted. |
| Low battery (error 200) | Input aborted. |
| Key abort (see SYBC or SYSP) | N/A. |

**Input Aborted.** In the above table, "input aborted" means that no data has been received or that the input operation has been interrupted. When input is aborted, the input operation is ended, and any characters received up to that point are discarded. The current input variable and all subsequent variables in the input list are left unchanged (note that variables prior to the one at which input was aborted will already have been changed.)

**NOTE** When input is aborted for INPUT, program execution continues on the next line of the program (not on the next statement, if INPUT is in a multistatement line). Notice from the table that this also occurs if [ENTER] is pressed but nothing else has been typed on the keyboard.

When input is aborted because of a numeric error, the I/O length reported by SYIN is always set to 0, since no data is placed in the input variable.

**Differences Between Development System and Handheld.** The development system does not produce errors 118, 119, or 200.

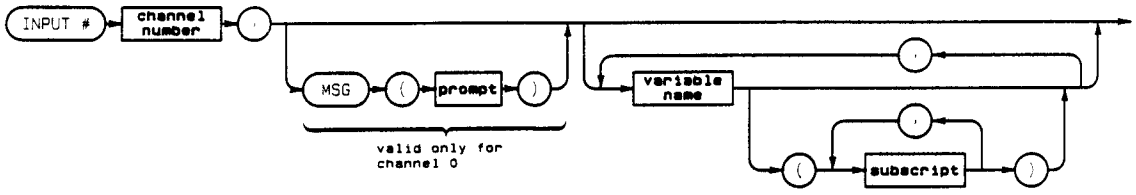
## Related Keywords

GET #, INPUT #, INPUT$, SYIN

# INPUT #

- ■ Exists in Development System
- □ Works Same in Development System
- ■ Allowed in IF...THEN

The INPUT # statement inputs items from the specified device or data file.




| Item | Description | Range |
|---|---|---|
| channel number | numeric expression, truncated to an integer | 0 through 15 |
| prompt | string expression | — |
| variable name | name of a simple numeric or string variable | any valid name |
| subscript | numeric expression, truncated to an integer | 0 through 32,767 |

## Examples

```
INPUT #5, Height, Width, Length
INPUT #channel, Stats(Index)
INPUT #0, MSG("Number of units: ") Unit$
```

## Description

The INPUT # statement inputs data from the device or data file associated with the specified channel number. All devices (except channel 0) and data files must be opened with OPEN # before they can be accessed with INPUT #. When a data file is opened, an associated file access pointer is positioned at the beginning of the file. The counterpart to INPUT # for output operations is PRINT #. (Note: INPUT #0 is equivalent to the INPUT statement. Refer to the keyword description for INPUT for details.)

INPUT # inputs data from a device or data file until it receives an end-of-line sequence consisting of carriage return and line feed. If input is from a data file, serial access only is performed. Each input item is read from the location in the file immediately after the previous item (serially). As each input variable is processed, the file access pointer advances beyond the data item in the file. When input

ends, the access pointer remains positioned after the last data item read. Subsequent input variables or file I/O statements continue reading data from or writing data to that position.

If no variable list is supplied, INPUT # will read until the end-of-line sequence is received. The received data will be discarded and program execution will continue.

Input continues until all input variables have been assigned data, or until input ends because of one of the conditions described in the next table.

**Behavior When INPUT # Ends**

| Ending Condition | Channel 0 Behavior | Channels 1-4 Behavior | Channels 5-15 Behavior |
|---|---|---|---|
| [ENTER] key pressed | Characters typed on the keyboard (except the [ENTER]) are placed in the input variable. Input is aborted if nothing is typed before [ENTER] is pressed. | N/A. | N/A. |
| CRLF received | N/A. | Characters received from the device (except the CRLF) are placed in the input variable. Input is aborted if nothing is received before the CRLF are received. | Characters read from the file (except the CRLF) are placed in the input variable. Input is aborted if there is no data to read before the CRLF are read. |
| Port terminate character received (see SYBC, SYRS, or SYSP) | N/A. | Ignored (input operation for that variable not ended). | N/A. |
| EOD or EOF encountered | N/A. | N/A. | Characters read up to the EOD or EOF are placed in the input variable. Input is aborted if the file access pointer is already at the EOD or EOF (no data to read). |
| Timeout (error 118) | Input aborted. | Input aborted. | N/A. |
| Power switch pressed (error 119) | Input aborted. | Input aborted. | N/A. |
| Low battery (error 200) | Input aborted. | Input aborted. | N/A |
| Port errors 201-208 | N/A. | Input aborted. | N/A. |

| Ending Condition | Channel 0 Behavior | Channels 1-4 Behavior | Channels 5-15 Behavior |
|---|---|---|---|
| Key abort (see SYBC or SYSP) | N/A. | Input aborted. | N/A. |
| Note: N/A means the ending condition will not occur for those channels. | | | |

**Input Aborted.** In the above table, "input aborted" means that no data has been received or that the input operation has been interrupted. When input is aborted, the input operation is ended, and any characters received up to that point are discarded. The current input variable and all subsequent variables in the input list are left unchanged (note that variables prior to the one at which input was aborted will already have been changed.) This is in contrast to GET #, in which any received data for that variable is saved.

**NOTE** When input is aborted for INPUT #, program execution continues on the next line of the program (not on the next statement, if INPUT # is in a multistatement line).

When input is aborted because of a numeric error, the I/O length reported by SYIN is set to 0, since no data is placed in the input variable.

**Bar-Code Data.** INPUT # is not recommended for reading bar-code data. Use GET # instead. Use of INPUT # can cause two problems.

The first problem might cause an infinite wait during the read operation. INPUT # will not end until it receives an end-of-line sequence consisting of a carriage return and a line-feed character. If the bar-code device configuration is changed to append a different trailer, INPUT # will not complete until it receives a CRLF.

The second problem might cause lost data. If the bar-code data has the end-of-line sequence (CRLF) embedded in it, INPUT # will only load the data up the the end-of-line sequence. The remaining data will be buffered for the next INPUT # statement.

See the description of the GET # keyword for more details on reading bar-code data.

**CAUTION** Do not mix INPUT # with GET # or INPUT$ when reading bar-code data. Doing so may result in unexpected error conditions.

**Differences Between Development System and Handheld.** The development system does not produce errors 118, 119, 200, or 201-208.

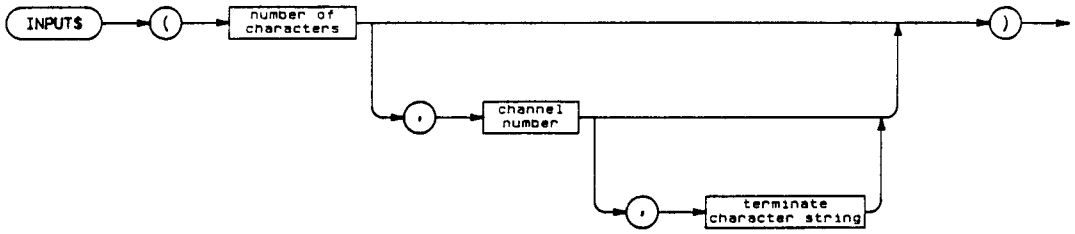
## Related Keywords

GET #, INPUT, INPUT$, PRINT #, SYIN

# INPUT$

Exists in Development System ■
Works Same in Development System □
Allowed in IF...THEN ■

The INPUT$ function inputs items from the specified device or data file.




| Item | Description | Range |
|---|---|---|
| number of characters | numeric expression, truncated to an integer | 0 through 255 |
| channel number | numeric expression, truncated to an integer | 0 through 15 |
| terminate character string | string expression | 4 characters maximum; 00h not allowed |

## Examples

```
I$=INPUT$(80, 5)
PRINT #6,INPUT$(RecLength, 5, ".!?")
```

## Description

INPUT$ inputs data from the device or data file associated with the specified channel number. All devices (except channel 0) and data files must be opened with OPEN # before they can be accessed with INPUT$. When a data file is opened, an associated file access pointer is positioned at the beginning of the file.

INPUT$ inputs data from a device or data file until it receives the specified number of characters or an end-of-line sequence consisting of any one of the characters in the terminate character string. If no terminate character string is specified, INPUT$ does not look for an end-of-line sequence.

If input is from a data file, serial access only is performed. Each input item is read from the location in the file immediately after the previous item (serially). As the data is read, the file access pointer advances beyond the data in the file. When input ends, the access pointer remains positioned after the last data item read. Subsequent file I/O statements continue reading data from or writing data to that position.

Input continues until one of the conditions described in the next table arises.

**Behavior When INPUT$ Ends**

| Ending Condition | Channel 0 Behavior | Channels 1-4 Behavior | Channels 5-15 Behavior |
|---|---|---|---|
| Requested number of characters received | Characters typed on the keyboard are placed in the input variable. | Characters received from the device are placed in the input variable. | Characters read from the file are placed in the input variable. |
| Character from the terminate character string received | Characters typed on the keyboard (including the terminate character) are placed in the input variable. | Characters received from the device (including the terminate character) are placed in the input variable. | Characters read from the file (including the terminate character) are placed in the input variable. |
| Port terminate character received (see SYBC, SYRS, or SYSP) | N/A. | Marks the end of the received data. The character is counted as one of the received characters, but is not placed in the input variable. INPUT$ continues to wait for another ending condition to occur. | N/A. |
| EOD or EOF encountered | N/A. | N/A. | Characters read up to the EOD or EOF are placed in the input variable. |
| Timeout (error 118) | Input aborted. | Input aborted. | N/A. |
| Power switch pressed (error 119) | Input aborted. | Input aborted. | N/A. |
| Low battery (error 200) | Input aborted. | Input aborted. | N/A. |
| Port errors (201-208) | N/A. | Input aborted. | N/A. |
| Key abort (see SYBC or SYSP) | N/A. | Input aborted. | N/A. |

Note: N/A means the ending condition will not occur for those channels.

**Input Aborted.** In the above table, "input aborted" means that no data has been received or that the input operation has been interrupted. When input is aborted, the input operation is ended, and any characters received up to that point are discarded. The input variable is left unchanged. This is in contrast to GET #, in which any received data is saved and the variables are set to 0 or the null string.

**NOTE** When input is aborted for INPUT$, program execution continues on the next line of the program (not on the next statement, if INPUT$ is in a multistatement line).

When input is aborted because of a numeric error, the I/O length reported by SYIN is set to 0, since no data is placed in the input variable.

**Bar-Code Data.** INPUT$ is not recommended for reading bar-code data. Use GET # instead. If you do use INPUT$, be sure to use the optional terminate character string and set it equal to the last character sent by the bar-code reader. This is normally the line-feed character (ASC$(10)) for HP Smart Wands and is *different* from the terminate character used by SYBC and SYSP.

Use of INPUT$ without the terminate character properly set can cause unpredictable results. INPUT$ will not complete until the requested number of characters has been received. If the bar code is longer than that length, INPUT$ will return only the number of characters of the input variable. The remaining bytes will be buffered for the next read statement to the port. If the bar code is shorter than the requested number of characters, INPUT$ will not return until enough characters have been scanned to fill the input variable. That might require several scans.

Even if the terminate character is used correctly, an additional problem can result. If the bar-code character has the terminate character embedded in it, INPUT$ will only load the data up to and including the terminate character. The remaining data will be buffered for the next read statement to that port.

**Differences Between Development System and Handheld.** The development system does not produce errors 118, 119, 200, or 201 through 208.

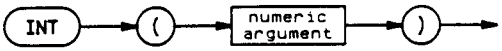
## Related Keywords

GET #, INPUT, INPUT #

# INT

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The `INT` function returns the integer part of the numeric argument.




| Item | Description | Range |
|---|---|---|
| `numeric argument` | numeric expression | — |

## Examples

```
PRINT INT(number)
Counter=INT(X+9.6)
```

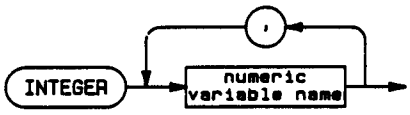
## Related Keywords

`FRC`

# INTEGER

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN □

The INTEGER statement declares integer variables and arrays.




| Item | Description | Range |
|---|---|---|
| numeric variable name | name of a simple numeric variable or numeric array | any valid name |

## Examples

```
INTEGER IntegerVariable

INTEGER IntegerArray1,IntegerArray2
DIM IntegerArray1(10),IntegerArray2(5,3)
```

## Description

To declare an integer array, the INTEGER statement must precede the DIM statement. Only the variable name is specified in the INTEGER statement, regardless of whether the declaration is for a simple integer or an integer array. It is the DIM statement that actually reserves the memory for an integer array, and the upper bounds of each dimension of the array are declared only in the DIM statement (see the second example). The default lower bound of the array is 1. The OPTION BASE statement is used to set the lower bound equal to 0.

When a real number is assigned to an integer variable, the number is truncated. Overflow occurs if the value of the number is outside the range of integers.

When variables are passed to a subprogram by reference, the precision declarations accompany the variable into the subprogram.

# ...INTEGER

The following table provides reference information about integer numeric variables. Refer to the keyword description for DIM for information about all the different BASIC variable types. Under "Memory Usage" for arrays, the total number of elements is the product of the number of elements in each dimension of the array. If the statement OPTION BASE 0 is used, the number of elements in each dimension is the specified upper limit plus 1. Because memory is allocated in *paragraphs*, or groups of 16 bytes, the total memory required will be the calculated memory usage rounded up to the next-higher 16-byte boundary.

**Integer Numeric Variables**

| Description | Value |
|---|---|
| Initial Value | 0 |
| Range | -32,768 through +32,767 |
| Maximum Array Size (bytes) | 65,535 |
| Maximum No. of Dimensions | 255 |
| Maximum Number of Elements | 32,766 (1 dimension) to 32,512 (255 dimensions) |
| Memory Usage (bytes) | |
| Simple Variable | 2 |
| Array | 2*(no. of elements)+2*(no. of dimensions)+1 |

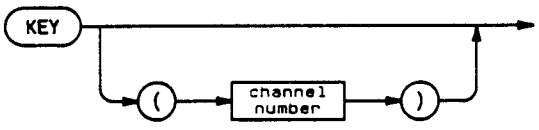
## Related Keywords

DIM, OPTION BASE

# KEY

Exists in Development System ■
Works Same in Development System □
Allowed in IF...THEN ■

The KEY function returns the number of characters currently in the key buffer or the serial port buffer.




| Item | Description | Range |
|---|---|---|
| channel number | numeric expression, truncated to an integer (default=0) | 0 through 1 |

## Examples

```
keybuffer$=INPUT$(KEY)
IF KEY>0 THEN keybuffer$=INPUT$(KEY)
rsbuffer$=INPUT$(KEY(1),1)
```

## Description

Characters entered through the keyboard or serial port are first stored in the key buffer (eight characters) or serial port buffer (64 characters). KEY or KEY(0) returns the number of characters in the key buffer (channel 0). KEY(1) returns the number of characters in the serial port buffer (channel 1). Key works only for the built-in serial-port handler. It does not work for other handlers such as HNSP.

The KEY function does not change the contents of the key buffer or serial port buffer. The buffers are only cleared by reading them with either the GET #, INPUT, or INPUT # statements or the INPUT$ function. INPUT$(KEY) (or INPUT$ (KEY(0), 0)) will read and clear the key buffer. INPUT$(KEY(1),1) will read and clear the serial port buffer for the built-in serial-port handler.

**Differences Between Development System and Handheld.** On the development system, KEY with no parameters (or KEY(0)) returns the number of characters in the key buffer (although the key buffer is 127 characters long). KEY(1) will always return 0 since there is no serial port buffer on the development system.

# ...KEY

## Related Keywords

GET #, INPUT, INPUT #, INPUT$

# LEN

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The LEN function returns the number of characters in the string argument.


| Item | Description | Range |
|---|---|---|
| string argument | string expression | — |

## Examples

```
Y=LEN(A$)
IF LEN(String$)<=10 THEN String$=String$+"/"
```

## Description

The value returned is the current number of characters in the string, regardless of its dimensioned length. The length of the null string is 0.

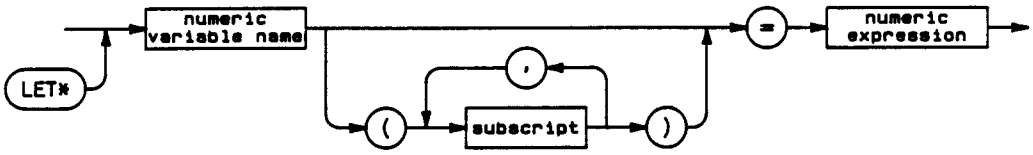
## Related Keywords

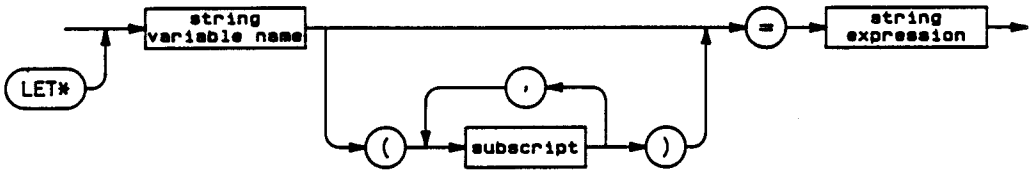
None.

# LET

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The LET statement assigns values to variables. The keyword is optional.




* Keyword is optional

| Item | Description | Range |
|---|---|---|
| `numeric variable name` | name of a simple numeric variable | any valid name |
| `subscript` | numeric expression, truncated to an integer | 0 through 32,767 |
| `numeric expression` | (see introduction) | — |
| `string variable name` | name of a simple string variable | any valid name |
| `string expression` | (see introduction) | — |

## Examples

```
LET Variable=5*X
Svariable$="ABC"+H$
LET A(2,4)=7
```

## Description

LET assigns the numeric or string value on the right side of the equation to the variable on the left side. Any variables used on the right side that have not previously been assigned will have the value 0 (numeric variables) or the null string (string variables). An error will occur if array variables that have not previously been dimensioned are referenced on either side of the equation.

A real expression is truncated when assigned to an integer variable. In this case the real expression must evaluate to a number within the integer range or an error will occur.

The following rules apply to string assignments:

- When a string expression is assigned to a string variable, excess characters are truncated on the right to the dimensioned size of the variable. For example, if A$ is dimensioned to 5, `A$="abcdefgh"` assigns `abcde` to A$.
- When the assigned expression is shorter than the dimensioned size, the remainder of the string is filled with nulls (ASCII 00h).

## Related Keywords

STR$

# LGT

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The LGT function returns the base 10 logarithm of the argument.


| Item | Description | Range |
|---|---|---|
| `numeric argument` | numeric expression | >0 |

## Examples

```
A(2)=A(1)*LGT(T)
IF LGT(X)=2 THEN PRINT X
```

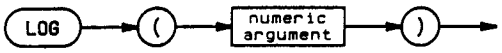
## Related Keywords

LOG

# LOG

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The LOG numeric function returns the natural (base e) logarithm of the argument.




| Item | Description | Range |
|---|---|---|
| numeric argument | numeric expression | >0 |

## Examples

```
T=1/K*LOG(N1/N2)
IF LOG(A)<=2 GOTO 900
```

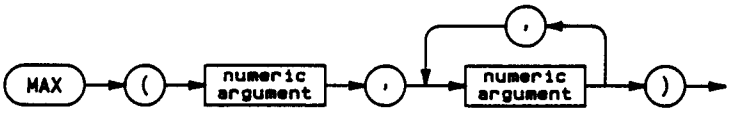
## Related Keywords

EXP, LGT

# MAX

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The MAX function compares a series of numeric arguments and returns the largest of the values.




| Item | Description | Range |
|---|---|---|
| `numeric argument` | numeric expression | — |

## Examples

```
Y=MAX(10,X)
Counter=INT(MAX(I,J,K,L,M,N))
```

## Description

The signs of the arguments are considered. `MAX(-1,-2,-3)` will return -1.

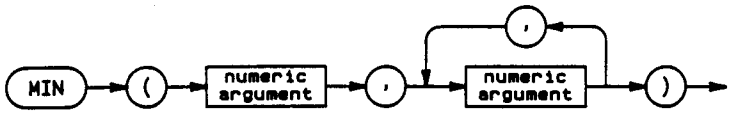
## Related Keywords

`MIN`

# MIN

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The MIN function compares a series of numeric arguments and returns the smallest of the values.




| Item | Description | Range |
|---|---|---|
| numeric argument | numeric expression | — |

## Examples

```
Y=MIN(10,X)
Counter=INT(MIN(I,J,K,L,M,N))
```

## Description

The signs of the arguments are considered. MIN(-1,-2,-3) will return -3.

## Related Keywords

MAX

# MOD

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The MOD function returns the remainder resulting from a division operation.


| Item | Description | Range |
|---|---|---|
| `dividend` | numeric expression | — |
| `divisor` | numeric expression | — |

## Examples

```
C=MOD(8,3)
IF MOD(Hours,Trip)<3 GOTO 300
```

## Description

The MOD operation is defined by the equation: `MOD(A,B)` = `A-B`×*INT(A/B)* where *INT(A/B)* is the integer part of `A/B`. `MOD(A,0)` is defined as 0.

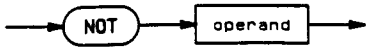
## Related Keywords

None.

# NOT

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The NOT operator returns the bit-by-bit NOT of the binary representation of the operand.




| Item | Description | Range |
|---|---|---|
| operand | numeric expression | -32,768 through +32,767 |

## Examples

```
IF NOT P THEN GOSUB 400
S=NOT J(1)
```

## Description

The operand is truncated to an integer represented as two's-complement. The results of each bit-by-bit NOT are used to construct the integer result. Each bit is computed according the following truth table.

**Bit-by-Bit NOT**

| Operand | Result |
|---|---|
| 0 | 1 |
| 1 | 0 |

Relational operators (=, >, <, <=, >=, and <>) always return -1 for true and 0 for false. The bit-by-bit NOT of these results will always be 0 or -1.

## Related Keywords

AND, OR, XOR, IF...THEN

# NUM

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The NUM function converts a string expression containing digits into a numeric value.


| Item | Description | Range |
|---|---|---|
| `string argument` | string expression | — |

## Examples

```
C=NUM(D$)
PRINT #1,NUM(Xcoordinate$)
```

## Description

The string can have leading blanks. The mantissa begins with the first non-blank character, which must be a plus or minus sign, decimal point, or digit. Additional characters can be digits or a decimal point; there can be only one decimal point per number.

If exponential notation is used, the exponent following E or e consists of an optional sign followed by one or two digits.

The argument must contain at least one digit. Embedded blanks and non-digit characters that are not used to build an exponent terminate the number.

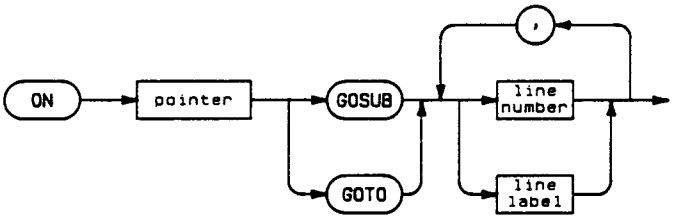
## Related Keywords

CHR$

# ON...GOSUB/GOTO

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The `ON...GOSUB/GOTO` statements transfer program execution to one of the specified program lines based on the value of a pointer.




| Item | Description | Range |
|---|---|---|
| `pointer` | numeric expression, truncated to an integer | (see Description) |
| `line number` | integer constant identifying a program line | 0 through 32,767 |
| `line label` | name of a program line | any valid name |

## Examples

```
ON P(1) GOTO 200,400,640
ON .5*Pointer1 GOSUB [Subroutine1],[Subroutine2]
IF Y THEN ON Y GOTO 300,[Odd],700
```

## Description

When the pointer evaluates to 1, execution is transferred to the first line number or line label. When the pointer evaluates to 2, execution is transferred to the second line number/label, and so on. If the pointer evaluates to a number less than 1 or greater than the number of line numbers/labels, execution is transferred to the statement after the ON.

If the GOSUB keyword is used, execution is transferred to the specified subroutine. When the RETURN statement of the subroutine is executed, execution branches to the statement immediately following the `ON...GOSUB` (on the same line, if the `ON...GOSUB` is in a multistatement line).

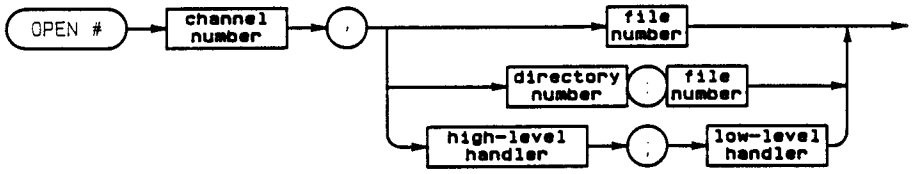
## Related Keywords

GOSUB, GOTO, IF...THEN, RETURN

# OPEN #

- ■ Exists in Development System
- □ Works Same in Development System
- ■ Allowed in IF...THEN

The OPEN # statement opens a device or data file by assigning to it a channel number.




| Item | Description | Range |
|---|---|---|
| channel number | numeric expression, truncated to an integer | 0 through 15 |
| directory number | non-negative integer | 0 through 4 |
| file name | string expression evaluating to null string or file name | — |
| high-level handler | string expression evaluating to null string or the file name | — |
| low-level handler | string expression evaluating to a file name | — |

## Examples

```
OPEN #1, ""
OPEN #5, "PDAT"
OPEN #channel+3, file$
OPEN #1, "HNWN;HNSP"
OPEN #2, "HNWN;HNBC"
```

## Description

OPEN # assigns (opens) a channel number to the specified device or data file. A data file must be created with %CALL SYAL or down-loaded from a host computer before it can be opened, and all devices (except channel 0) and data files must be opened before they can be accessed. Once a channel number is assigned to a device, it remains associated with that device until the channel is closed with the CLOSE # statement. When a data file is opened, an associated file access pointer is positioned at the beginning of the file.

The meaning of each channel number is defined in the table below.

| Channel Number | Meaning |
|---|---|
| 0 | console (keyboard and display) |
| 1 | serial port |
| 2 | bar-code port |
| 3-4 | reserved for future use |
| 5-15 | data file |

OPEN #1 turns on power to the serial port and to the HP 82470A RS-232C Level Converter (if one is connected to the serial port).

OPEN #2 turns on power to the bar-code port and to any bar-code reader attached to it.

Once a channel has been opened, it cannot be reopened without first being closed. Only one channel at a time can be assigned to a single file.

The use of the file specified by the file name in the OPEN # statement depends on the channel number, as defined in the table below.

| Channel Number | File Name Meaning |
|---|---|
| 0 | no meaning; console is always "open" |
| 1-4 | user-defined device handler file name (file type H) |
| 5-15 | data file name (file type D) |

**Device Handlers.** A device handler is an assembly-language program that controls programmatic access to an I/O device such as the serial or bar-code port.

For channel 1 (serial port), the built-in default device handler can be specified by supplying file name "" (null string). If a device handler name is supplied and no such handler exists in memory, the default handler will be used. For details on the capabilities of the built-in serial port device handler, refer to the keyword description for SYRS. Channels 2 through 4 have no default device handlers, so opening them to the default device ("") will give an error.

Three bar-code handlers are supplied with the Software Development System:

- HNSP is a low-level handler for smart bar-code readers attached to the serial port. For details on the capabilities of the handler and how to set options for the handler, refer to the SYSP keyword.
- HNBC is a low-level handler for smart bar-code readers attached to the bar-code port. For details on the capabilities of the handler and how to set options for the handler, refer to the SYBC keyword.

- HNWN is a high-level handler for the HP Smart Wand version 12.3 or later. It requires one of the two low-level bar-code handlers listed above. For details on the capabilities of the handler and how to set options for the handler, refer to the SYWN keyword.

Use the command GET # for input and PUT # or PRINT # for output.

---

**NOTE** Only the default serial port handler is always resident in the handheld. If you are using any other handlers (for example, HNWN, HNSP, or HNBC) be sure to download those files from the development system to the handheld. See the *Utilities Reference Manual* chapter 3 "HXC File Conversion Utility" and chapter 6 "HXCOPY File Copy Utility" for details.

---

**Differences Between Development System and Handheld.** The development system does not support directory numbers.

The development system does not support user-defined device handlers. For channel 1, the file name must be an MS-DOS device name (for example, "COM1:"). You cannot use the default device handler by supplying file name "". Use the MS-DOS command MODE to set the serial port configuration, since there is no SYRS keyword on the development system. HXBASIC does not perform XON/XOFF handshaking or receive-data buffering.

There is no equivalent for channels 2-4 on the development system, so they should not be used. On the development system, the OPEN # statement must be be used to create a data file since %CALL SYAL is not available. (File extension DAT is supplied automatically.) OPEN #n,file$ on the development system is equivalent to %CALL SYAL(file$) : OPEN #n,file$ on the handheld (i.e., the file size is 0 and the size increment is 1).

On the development system, OPEN # does not cause an error if the specified device or file is already open.

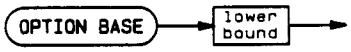
## Related Keywords

CLOSE #, GET #, INPUT #, INPUT$, PRINT #, PRINT #...USING, PUT #, SYAL, SYBC, SYRS, SYSP, SYWN

# OPTION BASE

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN □

The OPTION BASE statement specifies the lower bound of all arrays in a program or subprogram.




| Item | Description | Range |
|---|---|---|
| lower bound | integer constant (default = 1) | 0 |

## Examples

```
OPTION BASE 0
```

## Description

If used, an OPTION BASE statement must precede all array declarations. The option base is the lower bound of all numeric and string arrays in the program. (Upper bounds are declared in the DIM statement.)

The option base declaration is local; the option base must be declared by any subprograms called by the main program.

## Related Keywords

DIM, INTEGER

# OR

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The OR operator returns the bit-by-bit inclusive-OR of the binary representation of the operands.


| Item | Description | Range |
|---|---|---|
| operand | numeric expression | -32,768 through 32,767 |

## Examples

```
IF S<>0 OR P<>0 THEN GOSUB 400
S=J(1) OR J(2)
```

## Description

The operands are truncated to integers represented as two's-complement. The results of each bit-by-bit OR are used to construct the integer result. Each bit is computed according the following truth table.

**Bit-by-Bit OR**

| Operand 1 | Operand 2 | Result |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Relational operators (=, >, <, <=, >=, and <>) always return -1 for true and 0 for false. The bit-by-bit NOT of these results will always be 0 or -1.

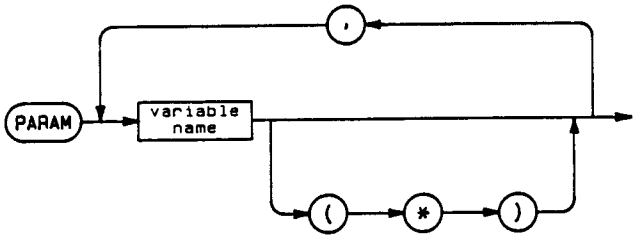
## Related Keywords

AND, NOT, XOR

# PARAM

- ■ Exists in Development System
- ■ Works Same in Development System
- □ Allowed in `IF...THEN`

The PARAM statement is the first statement in a subprogram. It defines the beginning of the subprogram and lists the formal parameters passed into the subprogram.




| Item | Description | Range |
|---|---|---|
| `variable name` | name of a simple numeric or string variable | any valid name |

## Examples

```
PARAM Xmin,Xmax,Yvar(*),Zvar(*)
PARAM choice$,answer$,list$(*),mean,stdev
```

## Description

If a subprogram is receiving or returning parameters to the calling program, the first line of the subprogram must be the PARAM statement. Only REM statements are allowed before PARAM. The statement cannot be part of a multistatement line. A subprogram can contain only one PARAM statement.

The variable names list the formal parameters passed from the calling program to the subprogram. The parameters become associated, from left to right, with the parameters listed in the CALL statement. The variable type (simple numeric, simple string, numeric array, string array) and the number of variables must agree with the parameters listed in the CALL statement. Entire arrays of any dimension are designated by a pair of parentheses containing only a single asterisk after the array name (regardless of the number of dimensions of the array). Variables in the calling program not explicitly passed to the subprogram are unknown to the subprogram.

The parameter list does not include precision declarations (real or integer), nor does it specify the dimensions of simple string variables and numeric and string arrays. The precision and dimensions of variables passed by reference accompany them as they are passed. When a numeric expression is passed (by value), the formal parameter to which it is passed is defined as a real number. When a string expression is passed (by value), the formal parameter to which it is passed is dimensioned to the current length of the string.

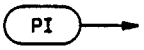
## Related Keywords

CALL, END

# PI

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The `PI` function returns the value of $\pi$.




## Examples

```
Tangent=TAN(PI*B)
Area=PI*Diameter**2/4
```

## Description

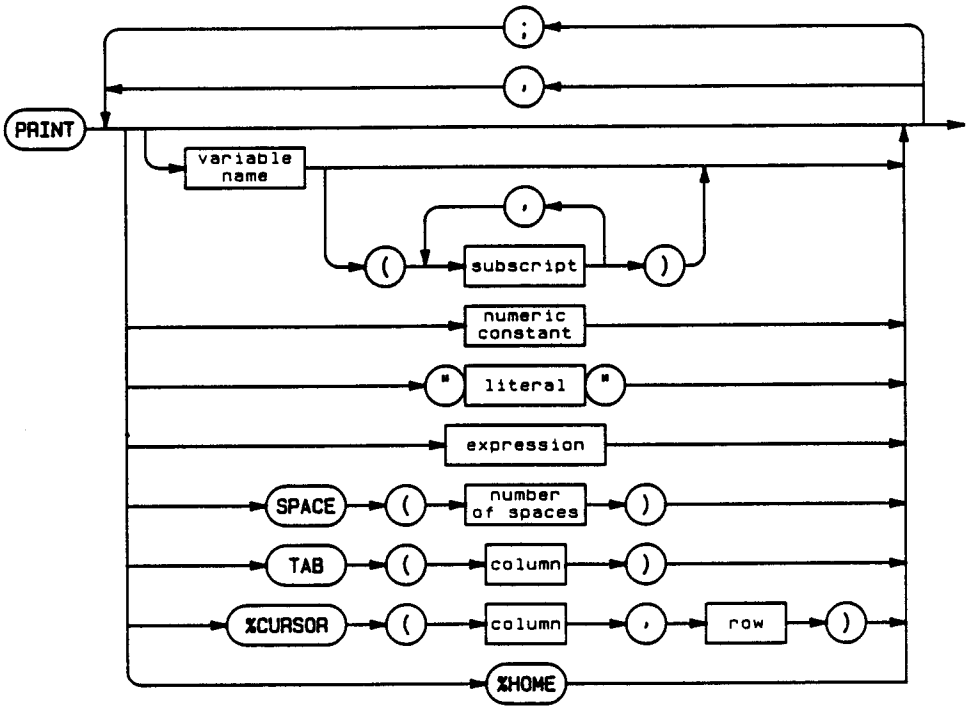
The exact value returned is 3.1415926535898.

## Related Keywords

`ACS, ADS, ARD, ASN, ATN, COS, DMS, RAD, SIN, TAN`

Exists in Development System ■
Works Same in Development System □
Allowed in IF...THEN ■

The PRINT statement outputs the print items to the current display line.




| Item | Description | Range |
|---|---|---|
| variable name | name of simple numeric or string variable | any valid name |
| subscript | numeric expression, truncated to an integer | 0 through 32,767 |
| numeric constant | numeric expression that can contain digits 0 through 9, plus or minus sign, a decimal point, and exponential notation | — |
| literal | string constant | — |
| expression | numeric or string expression | — |
| number of spaces | numeric expression, truncated to an integer | -255 through 255 |

| Item | Description | Range |
|---|---|---|
| column | numeric expression, truncated to an integer | 0 through 19 |
| row | numeric expression, truncated to an integer | 0 through 3 |

## Examples

```
PRINT Number; Letter$
PRINT TAB(10); A$; "Result="; Result
PRINT %CURSOR(col, row); "Up&0D&0ADown"
```

## Description

PRINT displays numbers and strings on the display. Numeric items are displayed in standard number format with a leading blank or minus sign. String items are displayed with no leading or trailing blanks.

When the length of data to be displayed exceeds the maximum line length, the cursor wraps around to the next line. When the cursor wraps around below the bottom line of the display, the entire contents of the display scrolls up one line to create a new line at the bottom, and the top line disappears off the top of the display.

When the list of display items is exhausted, an end-of-line sequence consisting of carriage return and line feed is sent to the display. The end-of-line sequence can be suppressed by including a semicolon or comma after the last display item.

**NOTE** When used to separate items in the list, the comma and semicolon behave identically. This is different than many other BASIC languages.

**Display Control Functions.** The functions SPACE, TAB, %CURSOR, and %HOME can be included as print items, and will output spaces or position the cursor as described in the table below (note that the cursor movement occurs even if the cursor is turned off):

**Display Control Functions**

| Function | Result |
|---|---|
| SPACE | Specified number of spaces (ASCII 20h) displayed. A negative number of spaces outputs backspaces (ASCII 08h) to the display. |
| TAB | Cursor moves to specified column (0-19) on current line. Column numbers greater than 19 are reduced MOD 20. |
| %CURSOR | Cursor moves to specified column (0-19) and row (0-3). Column 0 is the left column of the display; row 0 is the top line of the display. Negative column or row coordinates are treated as 0. Column or row coordinates that exceed the display boundaries (19 and 3) are ignored (the cursor does not move). |
| %HOME | Cursor moves to column 0, row 0 (the top left corner of the display) and clears the display. |

**Display Control Characters.** Any character in the handheld's character set can be specified with & and its two-digit hexadecimal ASCII code within a literal (& is specified by &&). This is especially useful for the display control characters, 01h through 1Fh. &00 (NUL) is not allowed in a string. Because the NUL character is used to terminate strings, if you create a string with a NUL somewhere before the end of the string, all characters after the NUL will be ignored.

The table below describes each display control character used by the handheld.

**Display Control Characters**

| Hex Value | Meaning |
|---|---|
| 01 (SOH) | Turn on cursor. |
| 02 (STX) | Turn off cursor. |
| 06 (ACK) | High tone beep for 0.5 second. |
| 07 (BEL) | Low tone beep for 0.5 second. |
| 08 (BS) | Move cursor left one column. When the cursor reaches the left end of the line, it will back up to the right end of the previous line. When the cursor reaches the top left corner, backspace will have no effect. |
| 0A (LF) | Move cursor down one line. If the cursor is on the bottom line, the display contents will scroll up one line. |
| 0B (VT) | Clear every character from the cursor position to the end of the current line. |
| 0C (FF) | Move cursor to top left corner and clear the display. |
| 0D (CR) | Move cursor to left end of current line. |
| 0E (SO) | Change keyboard to numeric mode (underline cursor). |
| 0F (SI) | Change keyboard to alpha mode (block cursor). |

| Hex Value | Meaning |
|---|---|
| 1E (RS) | Turn on electroluminescent backlight. |
| 1F (US) | Turn off electroluminescent backlight. |

**Differences Between Development System and Handheld.** Display control characters 076h and 07h sound the same beep on the development system.

The handheld uses the Roman-8 character set. The development system may use either the Roman-8 or the IBM-compatible character sets as set by the HXCHRSET utility described in chapter 5 of the *Utilities Reference Manual*. The difference between the two character sets occurs in the control codes (ASC$(0) through ASC$(31)) and in the upper half of the character set (ASC$(128) through ASC$(255)).

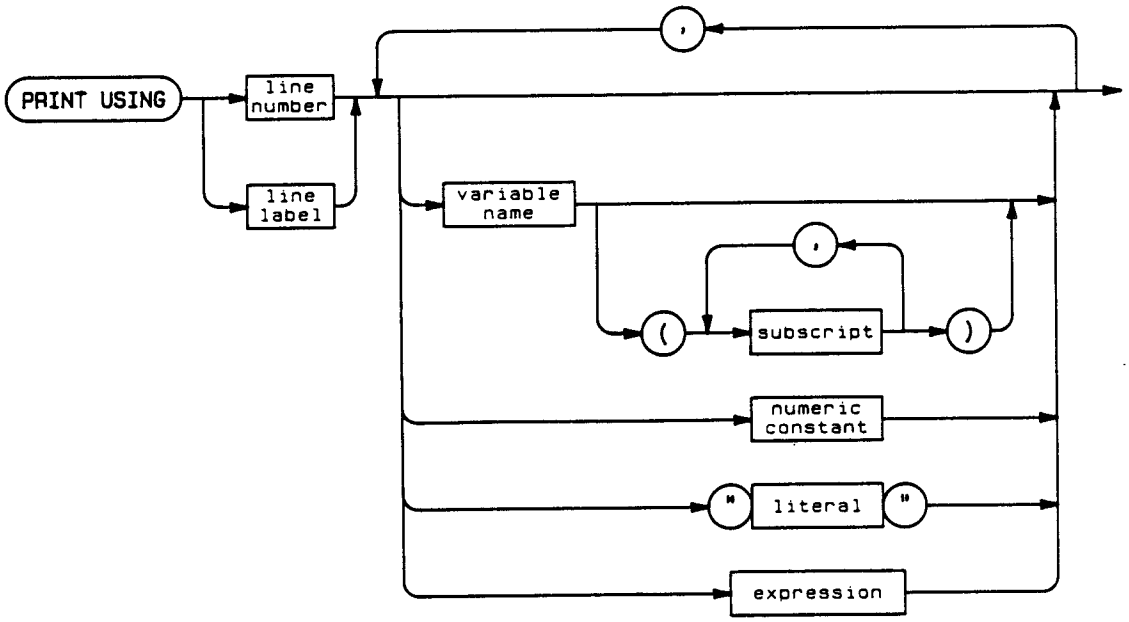
## Related Keywords

PRINT USING, PRINT #, PRINT #...USING

# PRINT USING

Exists in Development System ■
Works Same in Development System □
Allowed in IF...THEN ■

The PRINT USING statement outputs the print items to the current display line in a user-defined format.




| Item | Description | Range |
|---|---|---|
| line number | integer constant identifying a program line | 0 through 32,767 |
| line label | name of a program line | any valid name |
| variable name | name of simple numeric or string variable | any valid name |
| subscript | numeric expression, truncated to an integer | 0 through 32,767 |
| numeric constant | numeric expression that can contain digits 0 through 9, plus or minus sign, a decimal point, and exponential notation. | — |
| literal | string constant | — |
| expression | numeric or string expression | — |

# ...PRINT USING

## Examples

```
PRINT USING 100 Height, Width, Length
PRINT USING [numformat] "Result:",Stats(col,row),
PRINT USING 250 a*b, STR$(unit$,1), Potential
```

## Description

PRINT USING displays numeric and string items according to the format associated with the FORMAT statement on the specified program line. For a description of the display formats available, refer to the keyword description for FORMAT.

Notice that you can put literals in a FORMAT statement, but you cannot put field or format specifiers in a PRINT USING statement. This is different than many BASIC languages.

When the length of data to be displayed exceeds the current line length, the cursor wraps around to the next line. When the cursor wraps around below the bottom line of the display, the entire contents of the display scroll up one line to create a new line at the bottom.

When the list of display items is exhausted, an end-of-line sequence consisting of carriage return and line feed is sent to the display. The end-of-line sequence can be suppressed by including a comma (not a semicolon, as with PRINT) after the last display item.

**Display Control Characters.** Refer to the keyword description for PRINT for a list of all the display control characters.

**Differences Between Development System and Handheld.** The handheld uses the Roman-8 character set. The development system may use either the Roman-8 or the IBM-compatible character sets as set by the HXCHRSET utility described in chapter 5 of the *Utilities Reference Manual*. The difference between the two character sets occurs in the control codes (ASC$(0) through ASC$(31)) and in the upper half of the character set (ASC$(128) through ASC$(255)).

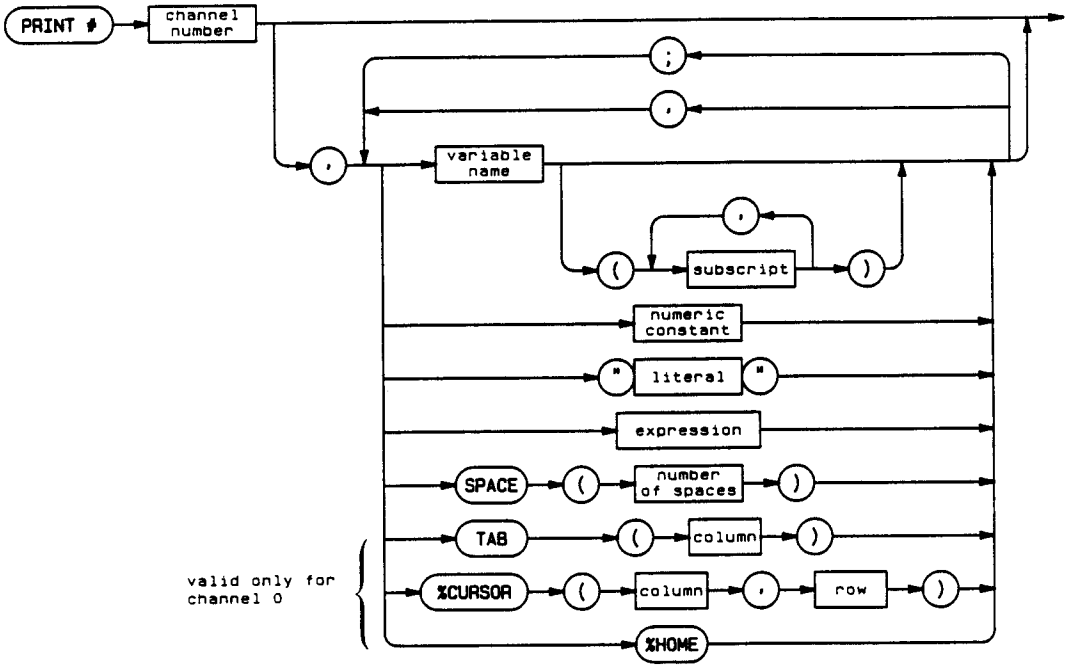
## Related Keywords

FORMAT, PRINT, PRINT #, PRINT #...USING

# PRINT #

Exists in Development System ■
Works Same in Development System □
Allowed in IF...THEN ■

The PRINT # statement outputs items to the specified device or data file.




| Item | Description | Range |
|---|---|---|
| channel number | numeric expression, truncated to an integer | 0 through 15 |
| variable name | name of a simple numeric or string variable | any valid name |
| subscript | numeric expression, truncated to an integer | 0 through 32,767 |
| numeric constant | numeric expression that can contain digits 0 through 9, plus or minus sign, a decimal point, and exponential notation | — |
| literal | string constant | — |
| expression | numeric or string expression | — |
| number of spaces | numeric expression, truncated to an integer | -255 through 255 |

| Item | Description | Range |
|---|---|---|
| column | numeric expression, truncated to an integer | 0 through 19 |
| row | numeric expression, truncated to an integer | 0 through 3 |

## Examples

```
PRINT #5, Height; Width; Length
PRINT #channel, "Result:";Stats(col,row);
PRINT #chno, a*b; STR$(unit$,1); Potential
```

## Description

The PRINT # statement outputs data to the device or data file associated with the specified channel number. All devices (except channel 0) and data files must be opened with OPEN # before they can be accessed with PRINT #. When a data file is opened, an associated file access pointer is positioned at the beginning of the file. The counterpart to PRINT # for input operations is INPUT #. (Note: PRINT #0 is equivalent to the PRINT statement. Refer to the keyword description for PRINT for details, including a list of display control characters.)

PRINT # outputs data from each item in the output list until the number of bytes contained by each item has been output. For example, PRINT # outputs as many bytes of data from a simple string variable as are contained in the string, with no leading or trailing blanks. If the variable contains less than the dimensioned number of bytes, only that many bytes are output. This is in contrast to PUT #, which outputs the number of bytes defined by the dimensioned length of the string, regardless of how many bytes the string actually contains.

Numeric items are output in standard number format with a leading blank or minus sign. This is in contrast to PUT #, which outputs numeric variables in the form they are stored internally.

When the list of output items is exhausted, an end-of-line sequence consisting of carriage return and line feed is output. The end-of-line sequence can be suppressed by including a semicolon or comma after the last output item.

---

**NOTE** When used to separate items in the list, the comma and semicolon behave identically. This is different than many other BASIC languages.

---

If output is to a data file, serial access only is performed. Each output item is written to the location in the file immediately after the previous item (serially). As each output item is processed, the file access pointer advances beyond the data item in the file. When output ends, the access pointer remains positioned after the last data item written. Subsequent file I/O statements continue reading data from or writing data to that position.

The SPACE function can be included as an output item for all channels. SPACE outputs the specified number of spaces (ASCII 20h) (or backspaces (ASCII 08h), if the number of spaces is negative). The other display control functions TAB, %CURSOR, and %HOME can be included as output

items for channel 0 only. Refer to the keyword description for PRINT for details on these functions.

Output continues until all output items have been written, or until output ends because of one of the conditions described in the next table.

**Behavior When PRINT # Ends**

| Ending Condition | Channel 0 Behavior | Channels 1-4 Behavior | Channels 5-15 Behavior |
|---|---|---|---|
| Number of characters contained in the output item written | Characters in the output item are displayed. | Characters in the output item are sent to the device. | Characters in the output item are written to the file. |
| EOF encountered | N/A. | N/A. | If the data file was created with a size increment greater than zero, the file will automatically expand as long as there is room in memory (see SYAL). |
| Timeout (error 118) | Output aborted. | Output aborted. | N/A. |
| Power switch pressed (error 119) | Output aborted. | Output aborted. | N/A. |
| Low battery (error 200) | Output aborted. | Output aborted. | N/A |
| Port errors 201-208 | N/A. | Output aborted. | N/A. |
| Lost connection while transmitting (error 218) | N/A | Output aborted | N/A. |

Note: N/A means the ending condition will not occur for those channels.

**Output Aborted.** In the above table, "output aborted" means that the output operation has been interrupted. When output is aborted, the output operation is ended. Subsequent variables in the output list are not output.

**CAUTION** When output is aborted for PRINT #, program execution continues on the next line of the program (not on the next statement, if PRINT # is in a multistatement line). When output is aborted because of a numeric error, the I/O length reported by SYIN is set to the number of bytes actually sent up to that point, since that data has already been written to the device or file.

**Bar-Code Data.** Both PUT # and PRINT # can be use to send data to a bar-code reader attached to the serial port. PRINT # is preferred because it allows string constants to be specified as part of the statement. Be sure you have opened the channel with the bar-code handler and have configured the channel correctly. See keywords OPEN #, SYSP, and SYWN for details.

**NOTE** The bar-code port is a read-only port, so writing data to that port will generate an error. To write data to a bar-code reader attached to the handheld, it must be attached to the serial port. The only exception to this rule is the port configuration escape sequence (see below for details).

Data sent to a bar-code reader are generally configuration instructions; responses from the bar-code device can be captured with the GET # statement. See the reference manual for the bar-code reader you are using for details on the escape sequences and responses for the bar-code reader you are using. When using the HNWN handler with an HP Smart Wand, however, there are two special escape sequences that are processed by the handler: the port configuration escape sequence and the status request escape sequence. Both are described below.

**Port Configuration Escape Sequence.** The port configuration escape sequence reconfigures both the serial port and the bar-code reader. The HNWN handler first sends the escape sequence to the bar-code reader using its current serial-port configuration and then changes the serial port to the configuration specified.

When sending the configuration escape sequence to the bar-code port, the handler will change only the port; the bar-code device itself must then be changed by the operator by scanning a configuration bar code that will set the bar-code reader to match the port configuration. Baud rate and parity should be changed as two separate steps, each comprising a programmatic write to the bar-code port and an operator scan of a configuration bar code.

When the HP Smart Wand is powered on and off (the OPEN # and CLOSE # keywords will do this), it may not return to its default serial configuration. This depends on whether the Smart Wand's serial port configuration has been saved. See documentation of your bar-code reader for details.

The port configuration escape sequence is a character string with the following format:

```
&c-ynP
```

where *n* is a sequence of numeric characters that specifies a decimal number between 0 and 255. Calculate the number by adding the values shown in the table below for each of the options selected.

| For each option... | Select one choice... | Add this value... |
|---|---|---|
| Baud rate | 150 | 0 |
| | 300 | 1 |
| | 600 | 2 |
| | 1,200 | 3 |
| | 2,400 | 4 |
| | 4,800 | 5 |
| | 9,600 | 6 |
| Stop bits* | 1 | 0 |
| | 2 | 8 |
| Parity | Always 0 | 0 |
| | Always 1 | 16 |
| | Even | 32 |
| | Odd | 48 |
| Character delay* | Off | 0 |
| | On | 64 |
| RTS/CTS Handshake* | Disabled | 0 |
| | Enabled | 128 |
| * Option does not affect HP-94 bar-code handler. Affects bar-code reader only. | | |

The statement `PRINT #1 ASC$(27);"-y62P"` would specify 9600 baud, 2 stop bits, odd parity, no character delay, and no handshaking.

**Status Request Escape Sequence.** The status request escape sequence is a way for an application program to obtain status information from the HP Smart Wand. It only works with the HNWN handler. The escape sequence causes the handler to buffer the bar-code device's response. The device responds so rapidly that the data would be lost without this feature.

---

**NOTE** For the status request escape sequence to work properly be sure that:

- You are using an HP Smart Wand connected to the serial port.
- The channel was opened with `OPEN # 1 "HNWN;HNSP"`.
- Transfer of escape sequences are enabled with the `SYWN` keyword.
- A `GET #` statement follows the statement that sends the status request.

---

The escape sequence has the format:

$^{E}$c-y*n*S

where *n* is a numeric character from the table below. Status messages other than those in the table are not supported.

| n | Type of Status Returned |
|---|---|
| 1 | Status message ending with a carriage return (ASC$(13)) |
| 2 | Status message with selected trailer. |
| 3 | Message ready/not ready response for single-read mode 2. |
| 5 | Serial number. |
| 6 | Configuration dump. |

See the reference manual for your bar-code reader for the format of the status messages sent by the bar-code device to the handheld. Status requests other than those in the above table are not supported.

**Differences Between Development System and Handheld.** The development system does not produce errors 118, 119, 200, or 218.

The handheld uses the Roman-8 character set. The development system may use either the Roman-8 or the IBM-compatible character sets as set by the HXCHRSET utility described in chapter 5 of the *Utilities Reference Manual*. The difference between the two character sets occurs in the control codes (ASC$(0) through ASC$(31)) and in the upper half of the character set (ASC$(128) through ASC$(255)).

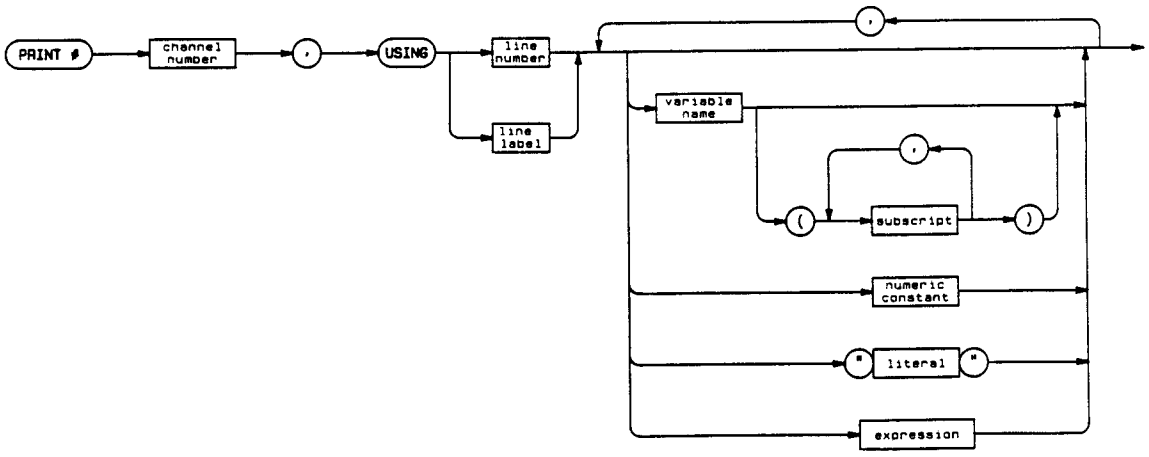
## Related Keywords

INPUT #, PRINT, PRINT USING, PRINT #...USING, PUT #, SYAL, SYIN

# PRINT #...USING

Exists in Development System ■
Works Same in Development System □
Allowed in IF...THEN ■

The PRINT #...USING statement outputs items to the specified device or data file in a user-defined format.




| Item | Description | Range |
|---|---|---|
| channel number | numeric expression, truncated to an integer | 0 through 15 |
| line number | integer constant identifying a program line | 0 through 32,767 |
| line label | name of a program line | any valid name |
| variable name | name of simple numeric or string variable | any valid name |
| subscript | numeric expression, truncated to an integer | 0 through 32,767 |
| numeric constant | numeric expression that can contain digits 0 through 9, plus or minus sign, a decimal point, and exponential notation | — |
| literal | string constant | — |
| expression | numeric or string expression | — |

# ...PRINT #...USING

## Examples

```
PRINT #1, USING 100 Height, Width, Length
PRINT #channel, USING [numformat] "Result:",Stats(col,row),
PRINT #chno, USING 250 a*b, STR$(unit$,1), Potential
```

## Description

The PRINT #...USING statement outputs data to the device or data file associated with the specified channel number, according to the format associated with the FORMAT statement on the specified program line. For a description of the display formats available, refer to the keyword description for FORMAT.

All devices (except channel 0) and data files must be opened with OPEN # before they can be accessed with PRINT #...USING. When a data file is opened, an associated file access pointer is positioned at the beginning of the file. There is no direct counterpart to PRINT #...USING for formatted input operations; the closest is INPUT #. (Note: PRINT #0,USING is equivalent to the PRINT USING statement. Refer to the keyword description for PRINT USING for details, including a list of display control characters.)

Notice that you can put literals in a FORMAT statement, but you cannot put field or format specifiers in a PRINT #...USING statement. This is different than many BASIC languages.

PRINT #...USING outputs data from each item in the output list until the number of bytes contained by each item has been output. For example, PRINT #...USING outputs as many bytes of data from a simple string variable as are contained in the string, with no leading or trailing blanks. If the variable contains less than the dimensioned number of bytes, only that many bytes are output. This is in contrast to PUT #, which outputs the number of bytes defined by the dimensioned length of the string, regardless of how many bytes the string actually contains.

Numeric items are output in standard number format with a leading blank or minus sign. This is in contrast to PUT #, which outputs numeric variables in the form they are stored internally.

When the list of output items is exhausted, an end-of-line sequence consisting of carriage return and line feed is output. The end-of-line sequence can be suppressed by including a comma (not a semicolon, as with PRINT #) after the last output item.

If output is to a data file, serial access only is performed. Each output item is written to the location in the file immediately after the previous item (serially). As each output item is processed, the file access pointer advances beyond the data item in the file. When output ends, the access pointer remains positioned after the last data item written. Subsequent file I/O statements continue reading data from or writing data to that position.

Output continues until all output items have been written, or until output ends because of one of the conditions described in the next table.

**Behavior When PRINT #...USING Ends**

| Ending Condition | Channel 0 Behavior | Channels 1-4 Behavior | Channels 5-15 Behavior |
|---|---|---|---|
| Number of characters contained in the output item written | Characters in the output item are displayed. | Characters in the output item are sent to the device. | Characters in the output item are written to the file. |
| EOF encountered | N/A. | N/A. | If the data file was created with a size increment greater than zero, the file will automatically expand as long as there is room in memory (see SYAL). |
| Timeout (error 118) | Output aborted. | Output aborted. | N/A. |
| Power switch pressed (error 119) | Output aborted. | Output aborted. | N/A. |
| Low battery (error 200) | Output aborted. | Output aborted. | N/A. |
| Port errors 201-208 | N/A. | Output aborted. | N/A. |
| Lost connection while transmitting (error 218) | N/A. | Output aborted. | N/A. |

Note: N/A means the ending condition will not occur for those channels.

**Output Aborted.** In the above table, "output aborted" means that the output operation has been interrupted. When output is aborted, the output operation is ended. Subsequent variables in the output list are not output.

**CAUTION** When output is aborted for PRINT #...USING, program execution continues on the next line of the program (not on the next statement, if PRINT #...USING is in a multistatement line).

When output is aborted because of a numeric error, the I/O length reported by SYIN is set to the number of bytes actually sent up to that point, since that data has already been written to the device or file.

**Differences Between Development System and Handheld.** The development system does not produce errors 118, 119, 200, or 218.

The handheld uses the Roman-8 character set. The development system may use either the Roman-8 or the IBM-compatible character sets as set by the HXCHRSET utility described in chapter 5 of the *Utilities Reference Manual*. The difference between the two character sets occurs in the control codes (ASC$(0) through ASC$(31)) and in the upper half of the character set (ASC$(128) through ASC$(255)).

# ...PRINT #...USING

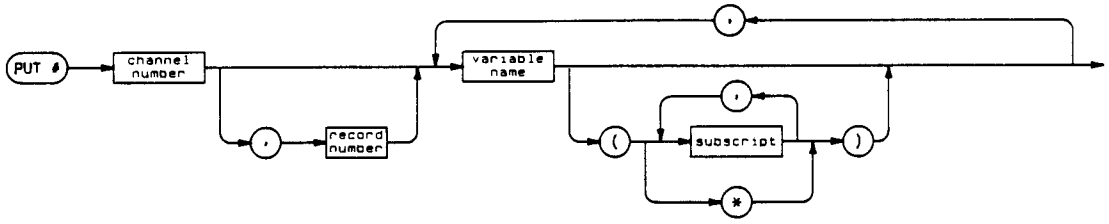
## Related Keywords

FORMAT, INPUT #, PRINT, PRINT USING, PRINT #, PUT #, SYAL, SYIN

# PUT #

Exists in Development System ■
Works Same in Development System □
Allowed in IF...THEN ■

The PUT # statement outputs items to the specified device or data file.




| Item | Description | Range |
|---|---|---|
| channel number | numeric expression, truncated to an integer | 0 through 15 |
| record number | numeric expression, truncated to an integer | 1 through 32,767 |
| variable name | name of a simple numeric or string variable | any valid name |
| subscript | numeric expression, truncated to an integer | 0 through 32,767 |

## Examples

```
PUT #5 Height, Width, Length
PUT #channel Stats(*)
PUT #chno,recno Potential, Unit$
```

## Description

The PUT # statement outputs data to the device or data file associated with the specified channel number. All devices (except channel 0) and data files must be opened with OPEN # before they can be accessed with PUT #. When a data file is opened, an associated file access pointer is positioned at the beginning of the file. The counterpart to PUT # for input operations is GET #.

# ...PUT #

PUT # outputs data from each variable in the output list until the number of bytes defined by the type of each variable has been output. For example, PUT # outputs 10 bytes of data from a simple string variable dimensioned to 10 characters. If there are actually less than 10 characters in the string, nulls are output for the remaining number of bytes. This is in contrast to PRINT # and PRINT #...USING, which output only the number of characters in the string, regardless of its dimensioned length.

Numeric variables are output in the form they are stored internally; for example, an integer is output as two bytes of binary data. This is in contrast to PRINT # and PRINT #...USING, which output numeric items in readable form. The table below shows the size of each type of variable.

**Sizes of Different Variables**

| Variable Type | Size (bytes) |
|---|---|
| Real variable | 8 |
| Integer variable | 2 |
| String variable | dimensioned length |
| Real array | 8 * (number of elements in each dimension) |
| Integer array | 2 * (number of elements in each dimension) |
| String array | (dimensioned length) * (number of elements in each dimension) |

An entire array (with any number of dimensions) can be output by specifying a single asterisk (*) as a subscript after the variable name.

If output is to a data file, access will be serial unless the optional record number is specified. When the record number is included, random access will be performed. The record number has meaning only when accessing data files.

**Serial Access.** When the record number is omitted, serial access is performed. Each record is written to the location in the file immediately after the previous record (serially).

**Random Access.** When the record number is included, random access is performed. Each record is read from the location in the file specified by the record number (randomly).

When the PUT # statement is executed, the file pointer is positioned to (*record size*) * (*record number* - 1) bytes from the beginning of the file, where *record size* is the total size (in bytes) of all the variables in the output list. Note that the definition of a *record* is totally arbitrary; PUT # does not place any end-of-record markers in a file, nor does its counterpart for reading, GET #, look for end-of-record markers within a data file. Therefore, the application program is responsible for maintaining a suitable data file structure. The way data will be written is dependent solely on the lengths of the variables in the output list.

As each output variable is processed, the file access pointer advances beyond the data item to the next

record position in the file. When output ends, the access pointer remains positioned after the last data item written. Subsequent file I/O statements that perform serial access continue reading data from or writing data to that position.

Because file access is controlled by the lengths of the variables in the output list, the simplest data file structure will have records that are the same fixed length, even though an entire record may be written by using several variables of different lengths. For a file structure using variable-length records, careful selection of variable lengths in different PUT # or PUT # statements will move the file access pointer to different parts of a data file. In addition, the SYPT statement can be used to move the pointer to the desired position.

Output continues until all output variables have been written, or until output ends because of one of the conditions described in the next table.

**Behavior When PUT # Ends**

| Ending Condition | Channel 0 Behavior | Channels 1-4 Behavior | Channels 5-15 Behavior |
|---|---|---|---|
| Number of characters defined by the size of the output variable written | Characters in the output variable are displayed. | Characters in the output variable are sent to the device. | Characters in the output variable are written to the file. |
| EOF encountered | N/A. | N/A. | If the data file was created with a size increment greater than zero, the file will automatically expand as long as there is room in memory (see SYAL). |
| Timeout (error 118) | Output aborted. | Output aborted. | N/A. |
| Power switch pressed (error 119) | Output aborted. | Output aborted. | N/A. |
| Low battery (error 200) | Output aborted. | Output aborted. | N/A |
| Port errors 201-208 | N/A. | Output aborted. | N/A. |
| Lost connection while transmitting (error 218) | N/A. | Output aborted. | N/A. |

Note: N/A means the ending condition will not occur for those channels.

**Output Aborted.** In the above table, "output aborted" means that the output operation has been interrupted. When output is aborted, the output operation is ended. Subsequent variables in the output list are not output.

When output is aborted for PUT #, program execution continues on the next line of the program (not on the next statement, if PUT # is in a multistatement line).

When output is aborted because of a numeric error, the I/O length reported by SYIN is set to the number of bytes actually sent up to that point, since that data has already been written to the device or file.

**Bar-Code Data.** Both PUT # and PRINT # can be use to send data to a bar-code reader attached to the serial port. PRINT # is preferred because it allows string constants to be specified as part of the statement. The PRINT # keyword includes a detailed discussion on writing data to bar-code readers.

**Differences Between Development System and Handheld.** The development system does not produce errors 118, 119, 200, or 218.

---

**NOTE** The bar-code port is a read-only port, so writing data to that port will generate an error. To write data to a bar-code reader attached to the handheld, it must be attached to the serial port. The only exception to this rule is the port configuration escape sequence (see below for details).

---

## Related Keywords

GET #, INPUT #, INPUT$, PRINT #, PRINT #...USING, PRINT$, SYAL, SYIN, SYPT

# RAD

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The RAD function interprets the numeric argument as an angle measured in degrees, and returns the value of the angle in radians.


| Item | Description | Range |
|---|---|---|
| numeric argument | numeric expression | — |

## Examples

```
Radians=RAD(Degrees)
PRINT RAD(90)
```

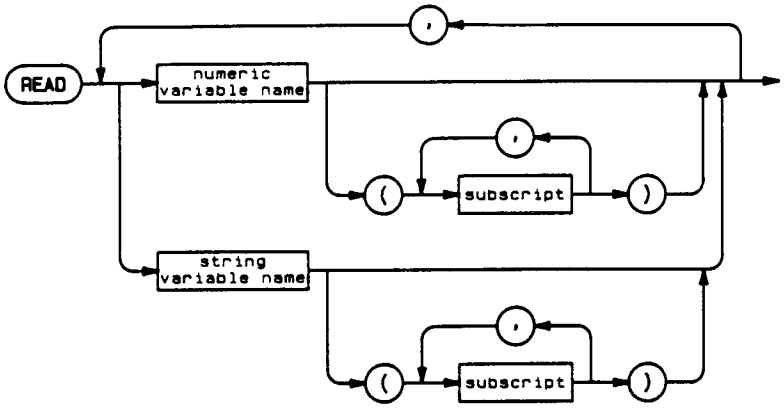
## Related Keywords

ACS, ADS, ARD, ASN, ATN, COS, DMS, PI, SIN, TAN

# READ

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The READ statement reads numeric and/or string constants from one or more DATA statements and assigns those values to program variables.




| Item | Description | Range |
|---|---|---|
| `numeric variable name` | name of a simple numeric variable | any valid name |
| `string variable name` | name of a simple string variable | any valid name |
| `subscript` | numeric expression, truncated to an integer | 0 through 32,767 |

## Examples

```
READ Variable1,Variable2$
READ A(1,2),B,C$,E$(4)
```

## Description

READ uses a data pointer to indicate the data item to be read. When program execution begins, the data pointer is positioned at the first item in the lowest-numbered DATA statement. When the data list in a particular DATA statement is exhausted, the pointer moves to the next-higher numbered DATA statement.

When a READ statement is successful (there is a DATA statement containing a data item of the proper type), execution proceeds to the next *statement* after the READ (on the same line, if the READ is in a multistatement line). When a READ statement attempts to read past the last data item in the program, execution proceeds to the next *line* after the READ, and the previous value of the variable being read into remains unchanged. This is similar to the input aborted condition that occurs for the INPUT statement.

When the READ statement is assigning a value to a string variable, the DATA statement can contain a numeric value, an unquoted string, or a quoted string; a numeric value is interpreted as an unquoted string containing digits. READ will treat anything after a DATA statement in a multistatement line as unquoted string data.

The order in which DATA statements are used can be changed using the RESTORE statement.

Each subprogram has its own data pointer, and can use only its own DATA statements. When a subprogram is called, its first READ statement uses the first DATA statement in that subprogram. When execution returns to a calling program, the calling program resumes use of its own data pointer starting from the pointer's last position.

An interrupt-processing routine defined by SYLB or SYSW has a separate READ data pointer than the one used in the non-interrupt portion of the program. All DATA statements are treated identically, regardless of where they appear in the program (interrupt routine or non-interrupt routine). When the READ statement is executed in an interrupt-processing routine, it will start reading at the first DATA statement in the program, regardless how many data items had been read before the interrupt routine was executed. When the interrupt routine ends, subsequent READ statements in the non-interrupt portion of the program will continue reading DATA statements as if the interrupt routine had not been executed. That is, reading will start after the last data item read before the interrupt routine was executed.

In an interrupt routine, RESTORE will affect the interrupt routine's data pointer independently of the non-interrupt routine's data pointer.

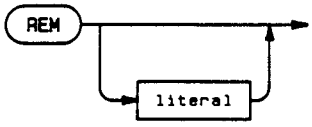
## Related Keywords

DATA, RESTORE, SYLB, SYSW

# REM

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The REM statement allows comments in a program.




| Item | Description | Range |
|---|---|---|
| `literal` | string constant | — |

## Examples

```
REM Written 11/17/86
REM
PRINT "Select menu item" : REM User must choose from menu
```

## Description

The REM statement can be used anywhere after the line number or after a `:` in a multistatement line; all characters following REM are considered to be part of the comment.

A REM statement in a multistatement line should be the last statement in the line, since subsequent characters will not be executed even if they look like legitimate BASIC statements.

## Related Keywords

None.

# RESTORE

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The RESTORE statement specifies that the first DATA statement will be accessed by the next READ operation.


## Examples

```
RESTORE
```

## Description

After RESTORE is executed, the next READ statement will read starting at the first item in the lowest-numbered DATA statement located in the same program or subprogram. When that data statement has been used, the data pointer moves to the next-higher numbered DATA statement. If there are no DATA statements in the program or subprogram, RESTORE has no effect.

An interrupt-processing routine has a separate READ data pointer than the one used in the non-interrupt portion of the program. All DATA statements are treated identically, regardless of where they appear in the program (interrupt routine or non-interrupt routine). When the READ statement is executed in an interrupt-processing routine, it will start reading at the first DATA statement in the program, regardless how many data items had been read before the interrupt routine was executed. When the interrupt routine ends, subsequent READ statements in the non-interrupt portion of the program will continue reading DATA statements as if the interrupt routine had not been executed. That is, reading will start after the last data item read before the interrupt routine was executed.

In an interrupt routine, RESTORE will affect the interrupt routine's data pointer independently of the non-interrupt routine's data pointer.

## Related Keywords

DATA, READ, SYLB, SYSW

# RETURN

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The RETURN statement is used within a subroutine to cause branching to the statement following the invoking GOSUB.


## Examples

```
RETURN
IF A>360 THEN A=360 : RETURN
```

## Description

When an invoking GOSUB (or `ON...GOSUB`) is embedded in a multistatement line, RETURN returns program execution to the statement immediately following the GOSUB (or `ON...GOSUB`).

For interrupt routines, `%CALL SYRT` performs a function similar to RETURN. The two keywords are *not* interchangeable, however.

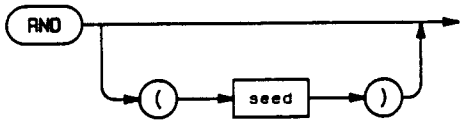
## Related Keywords

GOSUB, `ON...GOSUB`, SYRT

# RND

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The RND function returns a pseudorandom number as a decimal fraction greater than or equal to 0 and less than 1.




| Item | Description | Range |
|---|---|---|
| seed | numeric expression (default=0) | — |

## Examples

```
IF RND>.5 THEN PRINT "Heads"
Seed=RND(0)
```

## Description

The sequence of pseudorandom numbers returned depends on the seed. Using the same seed causes RND to generate the same series of numbers. A seed of 0 (RND(0)) produces random numbers based on the value of the real-time clock. You should execute RND once with a seed to establish the starting seed value (either one you specify or one based on the real-time clock). Subsequent uses of RND without a seed will return random numbers based on that starting seed.

The seed is global, and is passed between the main program and any subprogram(s).

## Related Keywords

None.

# SGN

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The SGN function returns 1 if the numeric argument is positive, -1 if the argument is negative, and 0 if the argument is 0.


| Item | Description | Range |
|---|---|---|
| `numeric argument` | numeric expression | — |

## Examples

```
IF SGN(Y)=1 THEN GOSUB 400
Root=SGN(X)*SQR(ABS(X))
```

## Related Keywords

ABS

# SIN

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The SIN function interprets the numeric argument as an angle measured in degrees, and returns the sine of the angle.


| Item | Description | Range |
|---|---|---|
| numeric argument | numeric expression | — |

## Examples

```
SineX = SIN(X)
If SIN(Theta)=1 THEN PRINT "Theta equals 90 degrees"
```

## Related Keywords

ACS, ADS, ARD, ASN, ATN, COS, DMS, PI, RAD, TAN

# SIZE

- ■ Exists in Development System
- □ Works Same in Development System
- ■ Allowed in `IF...THEN`

The `SIZE` function returns the amount of available memory in bytes.


## Examples

```
PRINT SIZE
IF SIZE+subusage<1000 THEN PRINT "Not enough room"
```

## Description

The available memory in the handheld is a dynamic quantity. When BASIC-language subprograms are called, local variable space is allocated; that memory is released when the subprograms end. Data files can expand or be deleted. Device handlers (such as for the serial port or the bar-code port) may allocate some memory when they are opened and release it when they are closed. Assembly-language subprograms may allocate some memory on a temporary or permanent basis when they are executed.

Consequently, the value returned by `SIZE` will only be valid as long as there are no operations occurring that allocate or release memory. `SIZE` should be used immediately before the amount of available memory is required in a calculation, and known worst case memory usage should be accounted for to ensure that predictions based on available memory are accurate.

**Differences Between Development System and Handheld.** On the development system, `SIZE` returns the amount of available memory in the BASIC workspace. When subprograms are called, they are loaded into the workspace, but they are not deleted when the subprogram ends. Data files can expand on the development system disc, but this expansion will not affect the available memory in the workspace. Because of these differences, `SIZE` will only provide meaningful information to the program when the program is running on the handheld.

## Related Keywords

None.

# SQR

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The SQR function returns the square root of the numeric argument. Negative arguments return an error.


| Item | Description | Range |
|---|---|---|
| numeric argument | numeric expression | ≥0 |

## Examples

```
PRINT SQR(X)
C=SQR(A**2+B**2)
```

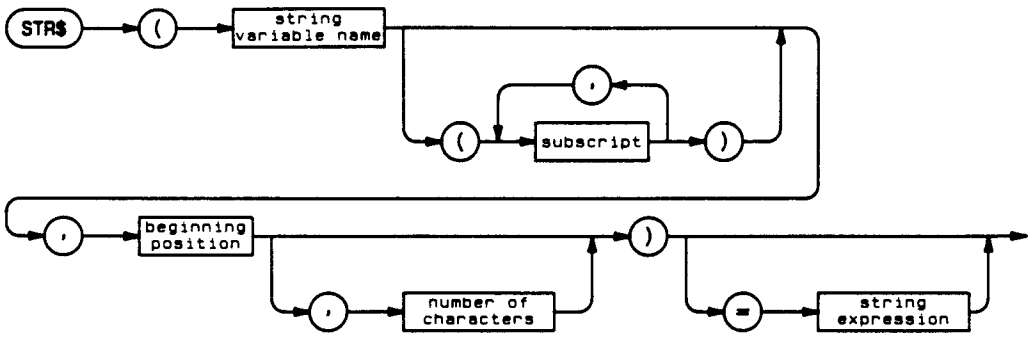
## Related Keywords

None.

# STR$

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The STR$ statement extracts substrings or assigns values to substrings.




| Item | Description | Range |
|---|---|---|
| `string variable name` | name of a simple string variable | any valid name |
| `subscript` | numeric expression, truncated to an integer | 0 through 32,767 |
| `beginning position` | numeric expression, truncated to an integer | -32,768 through +32,767 |
| `number of characters` | numeric expression, truncated to an integer (default = to end of string) | -32,768 through +32,767 |
| `string expression` | (see glossary) | — |

## Examples

```
A$=STR$(B$,2)
STR$(choice$(i),8,3)="ABC"
STR$(Name$,IDX(Name$," ")+1)=STR$(blank$,1,LEN(valid$))
```

## Description

When STR$ is used as a function in a string expression, it returns a substring of the string variable from the beginning position for the number of characters specified. If the number of characters is not specified, the number of characters from the beginning position to the end of the string is used. When on the left side of an assignment statement, STR$ assigns the string value (or result of the string expression) on the right side of the statement to the variable specified in the STR$ function, from the beginning position for the number of characters specified (or to the end of the string).

The following rules apply to string assignments:

- When a string expression is assigned to a string variable, excess characters are *truncated* to the dimensioned size of the variable. For example, if A$ is dimensioned to 5, `A$="abcdefgh"` assigns `abcde` to A$.
- When the assigned expression is shorter than the dimensioned size, the remainder of the string is filled with nulls (ASCII 00h).

The following rules apply to substring assignments:

- When a string expression is assigned to a substring, excess characters are *truncated* to the number of characters in the substring. For example, `STR$(A$,n,2)="abcde"` assigns `ab` to positions *n* and *n+1* of A$.
- When the assigned expression is shorter than the substring size, the assignment only changes the number of characters in the expression. The remainder of the characters in the substring before the assignment was performed are unchanged. For example, if A$ contains "`hello there`", `STR$(A$,3,6)="ab"` changes A$ to "`heabo there`".
- When a substring reference contains only the beginning position, characters are entered into the string starting at that position and continue to be entered until all characters are assigned or string is full. For example, `STR$(A$,n)="qrs"` assigns `qrs` to character positions *n*, *n+1*, and *n+2*.
- If the beginning position is greater than the length of the string variable (but still within the dimensioned size), spaces (ASCII 20h) are filled from the end of the string to the beginning position of the substring. For example, if A$ is dimensioned to 20 and contains "`hello there`", `STR$(A$,15,2)="ab"` changes A$ to "`hello there  ab`".
- If the beginning position is greater than the dimensioned size of the string variable, or if the number of characters is zero, no assignment is performed.

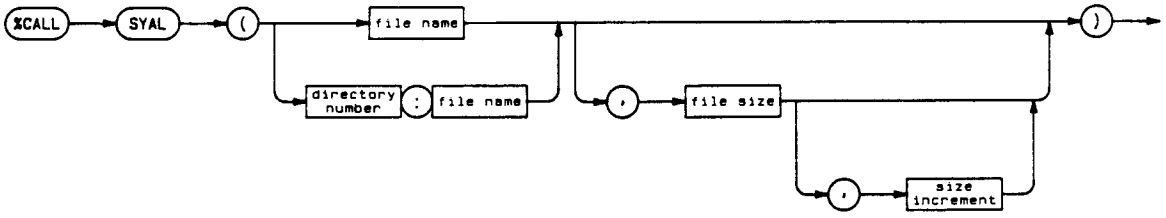
## Related Keywords

IDX, LET

# SYAL

- ☐ Exists in Development System
- ☐ Works Same in Development System
- ■ Allowed in `IF...THEN`

The `%CALL SYAL` (*AL*locate data file) statement creates a data file and optionally specifies the beginning file size and size increment.




| Item | Description | Range |
|---|---|---|
| `file name` | string expression | — |
| `directory number` | non-negative integer | 0 through 1 |
| `file size` | numeric expression, truncated to an integer | 0 through 32,767 |
| `size increment` | numeric expression, truncated to an integer | 0 through 32,767 |

## Examples

```
%CALL SYAL("PDAT")
%CALL SYAL(filename, 100, 10)
%CALL SYAL("0:MLPB", 200)
```

## Description

The file size and size increment values are in *paragraphs*, or blocks of 16 bytes. The file size indicates the size of the file when it is first created. The size increment indicates the increment used to increase the file size when a write operation attempts to write past the end of the file (that is, when the current file size is exceeded). For example, a size increment of three (3) means that when the file size is exceeded, the file will expand by as many three-paragraph increments (48 bytes) as are needed to accommodate the data being added to the file.

If the file size is not specified, it defaults to 0. If the size increment is not specified, it defaults to 1. A newly-created data file is automatically initialized to all nulls. New space added to the file (as defined by the size increment) is also initialized to all nulls.

%CALL SYAL only creates a data file. To be accessed, a data file must be opened with OPEN #.

**Differences Between Development System and Handheld.** SYAL is not implemented on the development system.

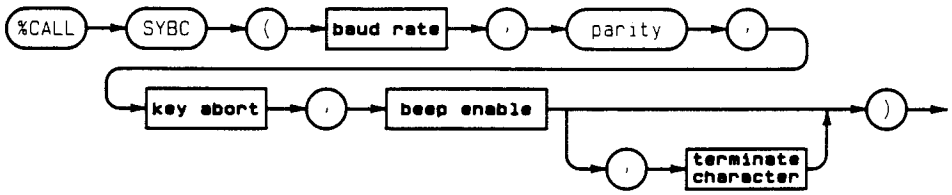
## Related Keywords

CLOSE #, OPEN #

# SYBC

- ☐ Exists in Development System
- ☐ Works Same in Development System
- ■ Allowed in `IF...THEN`

The `%CALL SYBC` (set *b*ar-code port configuration) statement sets the data communications configuration used by the bar-code port (channel 2).




| Item | Description | Range |
|---|---|---|
| `baud rate specifier` | numeric expression, truncated to an integer | 1 through 7 |
| `parity specifier` | numeric expression, truncated to an integer | 0 through 3 |
| `key-abort specifier` | numeric expression, truncated to an integer | 0 through 1 |
| `beep-enable specifier` | numeric expression, truncated to an integer | 0 through 1 |
| `terminate character` | string expression consisting of one character | — |

## Examples

```
%CALL SYBC(7,2,1,0)
%CALL SYBC(baud,parity,keyabort,beepenable,terminator$)
```

## Description

`%CALL SYBC` sets data communications configuration of the bar-code port by specifying the baud rate, parity, key abort option, beep enable, and terminate character with specifiers defined in the tables on this and the following pages. If the data communications configuration is not defined by a `%CALL SYBC` statement, it defaults to 9600 baud, 0 parity, good-read beep enabled, key abort enabled, and no

terminate character (equivalent to %CALL SYBC(1,0,1,1,"")).

The data communications configuration used by the bar-code port is the one in effect before the bar-code port is opened. Therefore, %CALL SYBC should be executed before OPEN #2.

The default data communications configuration is independent of the configuration set by the B (baud rate) operating system command. If you have used the B command, it will have no effect on the behavior of the bar-code port when running a BASIC program.

**Baud Rate Specifier.** The baud rate specifiers for the bar-code port follow the same convention as those for the serial port. Options are:

| Specifier | Baud Rate |
|---|---|
| 1 (default) | 9600 |
| 2 | 4800 |
| 3 | 2400 |
| 4 | 1200 |
| 5 | 600 |
| 6 | 300 |
| 7 | 150 |

**Parity Specifier.** Four parity options are available:

| Specifier | Parity |
|---|---|
| 0 (default) | 0 parity |
| 1 | 1 parity |
| 2 | even parity |
| 3 | odd parity |

**Key-Abort Specifier.** The key-abort feature is designed to simplify handling of unreadable bar-codes. It enables the application program to allow for keyboard entry of what is usually bar-code information.

If key abort is enabled, pressing a key will end input from the bar-code port and return control to the application program. The character for the key pressed will be in the key buffer. The value of the input variable differs whether the GET # or INPUT # statement was used. With GET #, the string variable is loaded with the null string. With INPUT #, the variable's contents remains what it was before the INPUT # statement was executed.

| Specifier | Key Abort |
|---|---|
| 0 | Key abort disabled |
| 1 (default) | Key abort enabled |

The five-step procedure outlined below provides a way for the application program to check whether a key abort has occurred. It works for both `GET #` and `INPUT #`.

1. Invoke error trapping using the `%CALL SYER` statement.
2. Set the error-number variable to 0.
3. Set the input variable to the null string. This is required for `INPUT #`, but optional for `GET #`.
4. Read the data using `GET #` or `INPUT #`.
5. Check the input and error-number variables. If the input variable is null and the error-number variable is 0, then bar-code entry was aborted from the keyboard.

**Beep-Enable Specifier.** If beep is enabled, the HP-94 will sound when it receives data from the bar code port.

| Specifier | Good-Read Beep |
|---|---|
| 0 | good-read beep disabled |
| 1 (default) | good-read beep enabled |

**Terminate Character.** The optional terminate character specifies what character will signal the end of incoming data. The terminate character can be any character except null (ASCII 00h); specifying the null character or a null string is equivalent to having no terminate character. If no terminate character is specified, a delay of 104 milliseconds after receipt of a character ends the input operation.

---

**NOTE** The command SYBC and its associated handler HNBC are not always resident in the handheld. Be sure to down-load these files from the development system to the handheld if you are using them. See the *Utilities Reference Manual* chapter 3 "HXC File Conversion Utility" and chapter 6 "HXCOPY File Copy Utility" for details.

The bar-code handler for the bar-code port (HNBC) will not work with so-called "dumb" bar-code devices (devices that return only a pulse train of light and dark transitions).

HNBC will return error 205 if it is waiting for bar-code data and the bar-code device becomes disconnected.

---

**Differences Between Development System and Handheld.** SYBC is not implemented on the development system.

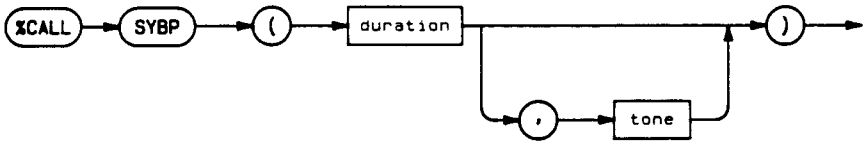
## Related Keywords

CLOSE #, OPEN #, SYSP, SYWN

# SYBP

- ☐ Exists in Development System
- ☐ Works Same in Development System
- ■ Allowed in `IF...THEN`

The `%CALL SYBP` statement produces an audible tone.




| Item | Description | Range |
|---|---|---|
| `duration` | numeric expression, truncated to an integer and modulo 256 to evaluate within the range 0 through 255 | -32,768 through +32,767 |
| `tone` | numeric expression, truncated to an integer (default = 0) | 0 through 1 |

## Examples

```
%CALL SYBP(15)
%CALL SYBP(5,1)
```

## Description

The duration is in tenths of seconds, allowing a range of 0.1 through 25.5 seconds. There are two tones: 0 specifies the low tone (approximately 600 Hz), and 1 specifies the high tone (approximately 1200 Hz).

Tones for 0.5 seconds can also be generated by including display control codes in strings sent to the display with the `PRINT`, `PRINT USING`, `PRINT # 0 USING`, `INPUT MSG`, and `INPUT #0, MSG` statements. Sending an ASCII 07h to the display (with `ASC$(7)` or `&07`) produces a low tone, and sending an ASCII 06h to the display (with `ASC$(6)` or `&06`) produces a high tone.

As soon as `%CALL SYBP` starts the beeper, the BASIC program then continues to run; that is, the program does not wait for the beep to finish before resuming execution. Consequently, `%CALL SYBP` can be executed again while the beeper is beeping. If the tone specified is different than the tone in progress, beeping will continue at the high tone and duration. The high tone and its duration will always take precedence, regardless of the order in which the tones are specified. If the new tone is the same as the tone in progress, beeping will continue at either the remaining duration or the new duration, whichever is longer.

**Differences Between Development System and Handheld.** SYBP is not implemented on the development system.

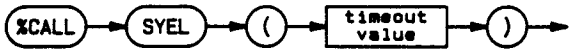
## Related Keywords

INPUT, INPUT #, PRINT, PRINT #, PRINT USING, PRINT #...USING

# SYEL

- ☐ Exists in Development System
- ☐ Works Same in Development System
- ■ Allowed in `IF...THEN`

The `%CALL SYEL` (*E*lectro*L*uminescent backlight timeout) statement sets the time period after which the electroluminescent display backlight will be turned off automatically.




| Item | Description | Range |
|---|---|---|
| `timeout value` | numeric expression, truncated to an integer (default = 120). | 0 through 1,800 |

## Examples

```
%CALL SYEL(15)
%CALL SYEL(0)
```

## Description

The timeout value is defined to be the time period after which the electroluminescent backlight will be turned off automatically. The timeout value is in seconds, allowing a range of 1 second to 30 minutes. A timeout value of 0 specifies that the backlight will never turn off. If the backlight timeout is not specified by `%CALL SYEL`, the timeout value defaults to 120 seconds (two minutes).

`SYEL` does not turn the backlight on or off – it only sets the duration of the automatic turn off of the backlight. To turn on the backlight, use the display control character 1Eh (e.g., `PRINT "&1E";`), or hold down the SHIFT key for one second. To turn off the backlight, use the display control character 1Fh (e.g., `PRINT "&1F";`), or let the backlight turn off automatically.

**CAUTION** Leaving the backlight on continuously or for long periods of time (greater than 5 minutes) will reduce the life of the backlight.

If the backlight is on when `SYEL` is executed, the backlight must be turned off (or turn itself off automatically after the previous timeout expires) before the new timeout will be in effect.

**Differences Between Development System and Handheld.** SYEL is not implemented on the development system.

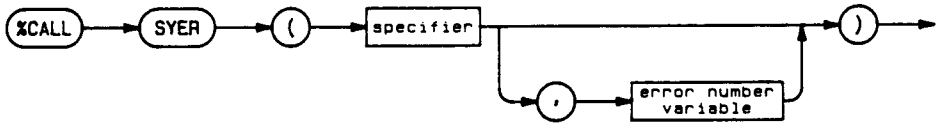
## Related Keywords

SYTO

# SYER

- ☐ Exists in Development System
- ☐ Works Same in Development System
- ■ Allowed in `IF...THEN`

The `%CALL SYER` (*ER*ror number) statement enables error trapping for numeric errors or restores default error processing.




| Item | Description | Range |
|---|---|---|
| `specifier` | numeric expression, truncated to an integer. | 0 through 1 |
| `error-number variable` | numeric variable | any valid name |

## Examples

```
%CALL SYER(0, error)
%CALL SYER(1)
```

## Description

The specifier can have two values, defined in the table below.

SYER **Specifiers**

| Specifier | Meaning |
|---|---|
| 0 | Causes numeric errors to be ignored, and assigns the error number to the error-number variable |
| 1 | Causes numeric errors to be processed normally. |

The handheld can produce two types of errors: numeric and non-numeric. Numeric errors are errors identified by a three-digit number, and non-numeric errors by two alphabetic characters. The most important distinction between numeric and non-numeric errors is that numeric errors can be trapped under program control, whereas non-numeric errors cannot be trapped.

**Numeric Errors.** When no alternate behavior has been specified for error processing, numeric errors cause the BASIC program to halt. Control returns to the operating system with the message `Error NNN LLLLL PPPP`, where NNN is the error number, LLLLL is the BASIC program line number, and PPPP is the name of the BASIC program or subprogram in which the error occurred. The SYER statement can specify that these errors be ignored, so that program execution continues uninterrupted. Some numeric errors (for example, error 200 low battery) need additional setup to avoid halting the program. When a numeric error does occur after `%CALL SYER` with specifier 0 has been executed, the error number is assigned to the error-number variable, allowing programmatic error handling or error trapping. The program name, line number, channel number, and I/O length of the most recent error can be determined with the SYIN statement.

Before executing a statement that may cause a numeric error, set the error number variable to 0. After the statement is executed, if the variable is still 0, no error occurred. If the variable is not 0, its contents will be the error number.

The error-number variable will always contain the value of the most recent numeric error. If a statement causes an error, and the program does not check for the error, the variable may be misinterpreted later (i.e., a subsequent statement does not produce an error when one is expected, but the error number variable still contains the error from the previous error-causing statement). Be sure to set the error-number variable to 0 before the statement of interest, and check the variable immediately afterwards to avoid losing its information. If the program traps errors, but fails to check if they occurred, errors could be missed.

The error-processing behavior defined by SYER is local to the program in which it is specified. `%CALL SYER` with specifier 0 must be called in each subprogram in order to enable error trapping for that subprogram.

**Non-Numeric Errors.** SYER has no effect on non-numeric errors. These errors always cause the program to halt, and return control to the operating system with the message `Error MM LLLLL PPPP`, where MM is the error message, LLLLL is the BASIC program line number, and PPPP is the name of the BASIC program or subprogram in which the error occurred.

Refer to the error appendix for a list of all numeric and non-numeric errors.

**Error Trapping and Interrupt Routines.** `%CALL SYLB` and `%CALL SYSW` specify the location of interrupt routines that will be executed when low battery, power switch, or timeout interrupts occur. These routines will only be executed if `%CALL SYER` with specifier 0 has been executed *in the program or subprogram which defines the interrupt routine* (contains the defining `%CALL SYLB` or `%CALL SYSW`). For error trapping to be enabled within the interrupt routine itself, `%CALL SYER` must also be executed in the interrupt routine.

The interrupt routines may be in a different program than the one executing when the interrupt occurs. To allow examining the error number in such an interrupt routine, the error-number variable should be passed (by reference) to different subprograms as a parameter when the subprogram is CALLed. That way any changes to the error-number variable in a subprogram will be indicated in any calling program that passed the variable.

Refer to the keyword descriptions of SYSW and SYLB for more information on interrupt routines.

# ...SYER

**Differences Between Development System and Handheld.** SYER is not implemented on the development system.

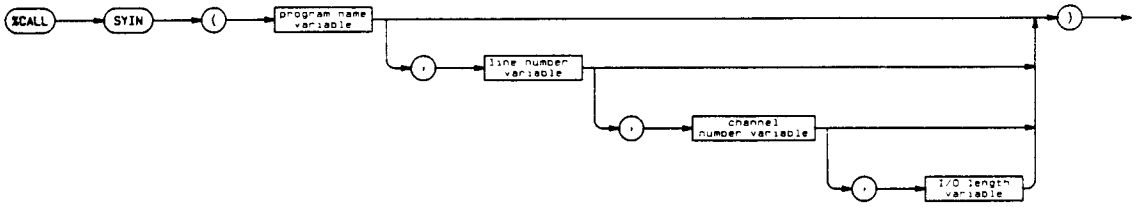
## Related Keywords

SYIN, SYLB, SYSW, SYTO

# SYIN

Exists in Development System ☐
Works Same in Development System ☐
Allowed in IF...THEN ■

The %CALL SYIN (error *IN*formation) statement returns, through the reference parameters, information about the most recent numeric error.




| Item | Description | Range |
|---|---|---|
| program name variable | string variable | any valid name |
| line number variable | numeric variable | any valid name |
| channel number variable | numeric variable | any valid name |
| I/O length variable | numeric variable | any valid name |

## Examples

```
%CALL SYIN(badprog$)
%CALL SYIN(badprog$, badline, badchannel)
%CALL SYIN(interprog$, interline, interchannel, interlength)
```

# ...SYIN

## Description

The %CALL SYIN statement assigns to the program name variable the file name of the BASIC program or subprogram that was executing when the most recent numeric error occurred. %CALL SYIN assigns to the optional line number variable the BASIC line number where the error occurred.

The optional channel number variable receives the channel number that was being used when the error occurred. If the error occurred while an I/O statement was executing for channels 0-4 (GET #, INPUT, INPUT #, INPUT$, PRINT #, PRINT #...USING, or PUT #), the channel number saved will be for the appropriate device or file. If no I/O was occurring when the error occurred, the channel number variable will be set to 0.

If the error occurred during the GET #, PRINT #, PRINT #...USING, or PUT # statements for channels 0-4, %CALL SYIN assigns to the optional I/O length variable the number of bytes of data that had been input or output up to that point. By the time the error occurs for these statements, those bytes will have been input (placed in the input variable) or output. If the error occurred for any other I/O statement, the I/O length will be 0 because the data received for that variable will have been discarded. In all cases, the I/O length will accurately reflect either the number of bytes input (placed in the input variable) or the number of bytes output when the error occurred.

The I/O length is accurate only for the variable that was being processed when the error occurred. If an I/O statement has more than one variable in its variable list, there is no way to tell which variable was being processed when the error occurred.

If no error has occurred before %CALL SYIN is executed, the values assigned default to zero and the null string.

For details on how the different I/O statements behave when they end because of an error, refer to the keyword description for each keyword.

**Differences Between Development System and Handheld.** SYIN is not implemented on the development system.

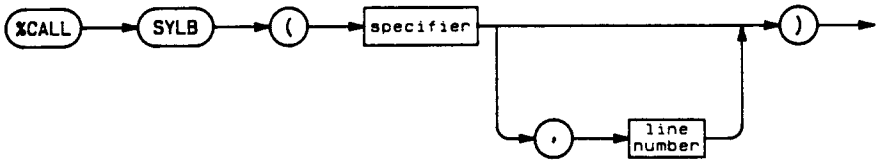
## Related Keywords

GET #, INPUT, INPUT #, INPUT$, PRINT #, PRINT #...USING, PUT #, SYER, SYLB, SYSW, SYTO

# SYLB

Exists in Development System ☐
Works Same in Development System ☐
Allowed in IF...THEN ■

The %CALL SYLB (*L*ow *B*attery behavior) statement specifies program behavior when a low battery condition occurs and error-trapping is enabled.




| Item | Description | Range |
|---|---|---|
| specifier | numeric expression, truncated to an integer | 0 through 1 |
| line number | numeric expression, truncated to an integer | 0 through 32,767 |

## Examples

```
%CALL SYLB(0, 1000)
%CALL SYLB(1)
%CALL SYLB(0, lineno)
```

## Description

The default behavior when a low battery condition occurs is for the handheld to stop what it is doing as quickly as possible. It halts any running program, shuts off the power to I/O devices such as the serial and bar code ports, turns off the electroluminescent display backlight, stops blinking the cursor and scanning the keyboard, and displays the message Error 200. The handheld will then wait for the power switch to be pressed to turn the machine off. The next time the power switch is pressed to turn the handheld on, it will cold start (refer to SYPO for details).

%CALL SYLB specifies alternate behavior for a low battery condition by identifying the location of a routine to process interrupts caused by a low battery. This routine will be executed when the handheld is already on and the main (NiCd) battery voltage drops below a certain level. (If the main battery is below that voltage level while the handheld is off, the machine will not turn back on until the battery has been charged enough to bring its voltage above that level.)

# ...SYLB

When the low battery condition occurs, it causes an error and, if error-trapping is enabled by %CALL SYER, causes program execution to branch to the specified interrupt-processing routine. The SYLB specifier determines the behavior during program execution when the low battery occurs. The specifier can have two values, defined in the table below. Unlike the power switch and timeout, processing of low battery cannot be disabled. It is always enabled (although it may only take the default action).

SYLB **Specifiers**

| Specifier | Meaning |
|---|---|
| 0 | Defines an interrupt-processing routine starting at the specified line number. When the low battery occurs, the program will complete the current BASIC statement (not line), and transfer control to this interrupt-processing routine. If I/O is occurring for channels 0-4, I/O will be aborted when the interrupt occurs. If I/O is occurring for channels 5-15, the current I/O statement will complete its operation (unless the battery is totally exhausted). |
| 1 | Cancels the specified interrupt-processing routine. Restores low battery behavior to either the last behavior defined by the a previous calling program or to the default behavior. |

The I/O statements that can be interrupted during I/O to channels 0-4 are INPUT, INPUT #, INPUT$, GET #, PRINT #, PRINT #...USING, and PUT #.

When the low battery interrupt routine begins to execute, nothing will have changed (except that I/O to channels 0-4 will have been aborted). That is, the devices and files will still be open, the display backlight will still be on, the cursor will still be blinking, and the keyboard will still be scanned.

**Error Trapping and Interrupt Routines.** %CALL SYLB defines the location of interrupt routine that will be executed when low battery occurs. This routine will be executed only if %CALL SYER with specifier 0 has been executed *in the program or subprogram which defines the interrupt routine* (contains the defining %CALL SYLB). The following table shows the behavior of low battery depending on whether SYLB and SYER are used together.

SYLB **and** SYER **Interaction**

| Usage | Low Battery Behavior |
|---|---|
| Neither SYLB or SYER | Program halts with Error 200. |
| SYER only | Program halts with Error 200. |
| SYLB only | Program halts with Error 200. |
| Both SYLB and SYER | Program execution transfers to the interrupt-processing routine. If I/O to channels 0-4 was interrupted, error number variable set to 200, and SYIN variables updated. |

For error trapping to be enabled within the interrupt routine itself, %CALL SYER with specifier 0 must also be executed in the interrupt routine.

The error number variable is always local to the currently executing program or subprogram (in this case, the interrupted program). To allow examining the error number in an interrupt routine that is in a different program than the interrupted program (see next topic), the error number variable should be passed by reference to different subprograms as a parameter when the subprogram is CALLed. That way any changes to the error variable in a subprogram will be indicated in any calling program that passed the variable.

**Global and Local Control.** SYLB provides both global and local control of the behavior of the power switch and timeout. Global control occurs when an interrupt-processing routine is defined in the main program. This routine will be in effect whenever the main program or any subprogram it calls is executing. Local control occurs when a subprogram defines a new interrupt-processing routine. The new routine will be in effect for that subprogram and for any subprograms it calls, until the subprogram cancels the interrupt routine (with specifier 1) or the subprogram ends. Then the low battery behavior will revert back to either the last behavior defined by a previous calling program (which may be the global behavior), or to the default behavior if no previous caller has defined an interrupt routine.

This local-versus-global control is true no matter how deep subprograms are nested. That is, an interrupt-processing routine in a calling program is no longer in effect when the called subprogram defines its own interrupt-processing routine, but is reactivated when the subprogram cancels its routine or ends.

**Automatic Power Off After Low Battery.** The low battery condition only occurs once, when the main (NiCd) battery voltage drops below a certain level. At that point, the program has two to five (2-5) minutes left before the battery voltage drops so low that the handheld turns itself off automatically without warning. (The low battery condition will not occur again until the handheld has been turned off and back on after the battery has been recharged enough to raise its voltage above the low battery level.)

The actual amount of time available depends on what is happening when the low battery condition occurs. For example, the display backlight takes more power, as does the HP 82470A RS-232C Level Converter (if one is connected to the serial port), so less operating time will be available if these are on. The time also depends on how much the battery was charged during its last charging cycle, the ambient temperature, and many other factors. Because the remaining operating time is variable, the program should respond to the low battery interrupt as rapidly as possible by ending its activity gracefully (complete file updates that were in progress, etc.), notifying the user that it is necessary to recharge the main battery, and turning the power off with %CALL SYPO.

If the program continues operating until the handheld turns itself off automatically, the effect is as if the reset switch was pressed. No data in data files will be lost, since the memory backup batteries will keep memory intact, but the handheld will cold start the next time it is turned on. This means that any data in program variables that did not get saved in a data file will be lost.

**Distinguishing Between Low Battery and Power Switch.** An interrupt-processing routine can determine which event started its execution by looking at the error variable. However, the variable will be set to the low battery error, 200, only when I/O to channels 0-4 is interrupted. If low battery occurs during execution of non-I/O BASIC statements, it is not considered an error condition, so the error variable will not be changed from its current value. Consequently, the value in the error variable may not give a true indication of whether or not low battery occurred.

Note: the power switch interrupt has the same characteristics as the low battery interrupt. It can occur at any time in a program, and will only be considered an error (and thus set the error variable to the power switch error, 119) if I/O to channels 0-4 is interrupted. It is recommended that you define the low battery interrupt routine to be the different than the power switch interrupt routine if you need to distinguish which event caused the interrupt.

**Data File Integrity.** When a low battery interrupt occurs during execution of non-I/O BASIC statements, the current statement will finish executing before control is transferred to the interrupt-processing routine. Only BASIC statements performing I/O to channels 0-4 will be interrupted, and will not complete their I/O, because of low battery interrupts (aborted I/O is discussed in the keyword descriptions for the I/O statements). If data is being written into a file, the write operation will be completed before the interrupt routine begins to execute. You do not have to worry about files being corrupted because of a partially-completed (interrupted) file write operation.

However, you do have to worry about files being properly updated if the program executes several file write statements, and the interrupt occurs before all the statements have been completed. In that situation, if the program turns off the machine and specifies a subsequent cold start ( %CALL SYPO(0) ), the file update will be incomplete. For this reason, you may want to write your application so that it completes the file update operation before turning off the power. (At a subsequent warm start, the program will continue execution, and can then complete the file update process with no loss of file integrity.)

**Behavior During Interrupt-Processing Routines.** The global control of the low battery means that program execution will transfer to the interrupt-processing routine *even if the interrupt occurs in a different subprogram than the one containing the interrupt routine.* Because this is effectively a subroutine call that can cross program boundaries, RETURN cannot be used to return from the interrupt routine to the program that was executing when the interrupt occurred. Instead, %CALL SYRT must be used to return from the interrupt routine.

The CALL statement will give an error if it is used within an interrupt-processing routine. If the END statement is executed in an interrupt-processing routine, the program or subprogram will end and return control back to the operating system (not to the calling (sub)program).

With the previous exceptions, any BASIC statement can be executed in an interrupt routine, including %CALL SYLB (and %CALL SYSW ) with any specifier. Once the interrupt routine ends, any subsequent interrupt will be processed according to conditions defined by the most recently executed SYLB statement.

New interrupts can still occur while an interrupt-processing routine is executing. However, they do not cause that or any other interrupt routine (such as power switch) to be executed. During execution of non-I/O BASIC statements in an interrupt routine, new interrupts are not processed until the interrupt routine ends. Because of this, interrupt routines should be as short as possible.

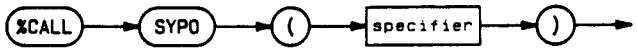
During I/O to channels 0-4 in an interrupt routine, new interrupts still cause I/O to be aborted and the error number variable to be changed, but the interrupt routine itself is not reexecuted. (Recall that for this to occur, %CALL SYER with specifier 0 must also be executed in the interrupt routine.)

An interrupt-processing routine has a separate READ data pointer than the one used in the non-interrupt portion of the program. All DATA statements are treated identically, regardless of where they appear in the program (interrupt routine or non-interrupt routine). When the READ statement is executed in an interrupt-processing routine, it will start reading at the first DATA statement in the program, regardless how many data items had been read before the interrupt routine was executed. When

# SYPO

- ☐ Exists in Development System
- ☐ Works Same in Development System
- ■ Allowed in `IF...THEN`

The `%CALL SYPO` (*P*ower *O*ff) statement programmatically turns off the handheld.




| Item | Description | Range |
|---|---|---|
| `specifier` | numeric expression, truncated to an integer | 0 through 1 |

## Examples

```
%CALL SYPO(warm)
%CALL SYPO(0)
```

## Description

The specifier indicates the manner in which the current program should be started the next time the power switch is pressed to turn the handheld on. Specifier 0 causes the handheld to *cold start* the program called `MAIN` from the beginning; specifier 1 causes the handheld to *warm start* the current program from the statement following the `%CALL SYPO` statement. The meanings of the terms cold start and warm start are defined in the following table:

**Handheld Restart Behavior**

| Cold Start (default) | Warm Start |
|---|---|
| Display cleared | Display cleared (same as cold start) |
| I/O halted | I/O halted (same as cold start) |
| Key buffer and serial port buffer cleared | Key buffer and serial port buffer cleared (same as cold start) |
| Program named MAIN restarts from beginning | Current program restarts from statement after %CALL SYPO |
| All data files closed | Previously open data files remain open |
| Serial, bar-code and expansion ports closed | Previously open serial, bar-code and expansion ports remain open |
| Electroluminescent backlight turned off | Electroluminescent backlight remains on |
| BASIC variable contents lost | BASIC variable contents preserved |
| Allocated scratch space for assembly language programs reclaimed | Allocated scratch space for assembly language programs preserved |
| Cursor turned on as blinking underline (_) | Cursor status unchanged |
| Keyboard set to numeric mode (unshifted) | Keyboard status unchanged |
| Electroluminescent backlight timeout value set to 120 seconds | Electroluminescent backlight timeout value unchanged |
| Power switch behavior set to default | Power switch behavior unchanged |
| Handheld timeout value set to 120 seconds | Handheld timeout value unchanged |
| Low battery behavior set to default | Low battery behavior unchanged |

Cold start behavior is provided to completely restart an application from the beginning. Warm start behavior is provided to continue an application from where it was interrupted. For most applications, the only aspect that must be restored after a warm start is the contents of the display and continuation of I/O, if appropriate.

If the restart behavior is not specified by a %CALL SYPO statement, it defaults to cold start behavior.

**Differences Between Development System and Handheld.** SYPO is not implemented on the development system.

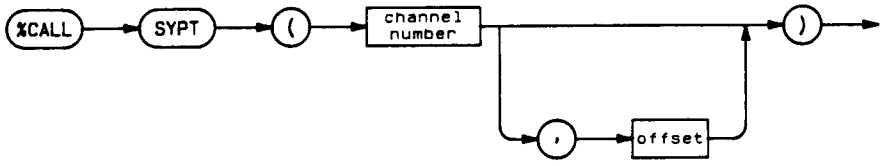
## Related Keywords

SYLB, SYSW

# SYPT

- ☐ Exists in Development System
- ☐ Works Same in Development System
- ■ Allowed in `IF...THEN`

The `%CALL SYPT` (set *PoinTer*) statement sets the data file pointer of the specified channel.




| Item | Description | Range |
|---|---|---|
| `channel number` | numeric expression, truncated to an integer | 5 through 15 |
| `offset` | numeric expression | 0 through $2^{31} - 1$ |

## Examples

```
%CALL SYPT(channel)
%CALL SYPT(15, position)
```

## Description

The data file pointer affected is the one associated with the data file specified by the channel number. The data file must have been OPEN ed before SYPT can move the pointer.

If no offset is included, the data file pointer is set to the end of the data in the file (EOD), which may also be the end of the file itself (EOF). If an offset is included, the data file pointer is set to the number of bytes specified by the offset from the beginning of the file (offset is relative to 0).

`%CALL SYPT` will give an error if the specified offset is beyond EOD.

**Differences Between Development System and Handheld.** SYPT is not implemented on the development system.

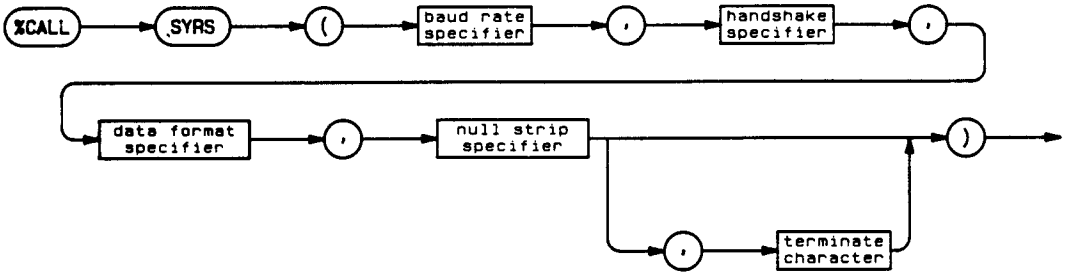
## Related Keywords

EOF, GET #, INPUT #, INPUT$, OPEN #, PRINT #, PRINT #...USING, PUT #

# SYRS

Exists in Development System ☐
Works Same in Development System ☐
Allowed in IF...THEN ■

The %CALL SYRS (set *RS*-232 configuration) statement sets the data communications configuration used by the built-in serial-port handler.




| Item | Description | Range |
|---|---|---|
| baud rate specifier | numeric expression, truncated to an integer | 1 through 7 |
| handshake specifier | numeric expression, truncated to an integer | 0 through 1 |
| data format specifier | numeric expression, truncated to an integer | 0 through 15 |
| null strip specifier | numeric expression, truncated to an integer | 0 through 1 |
| terminate character | string expression consisting of one character | — |

## Examples

```
%CALL SYRS(4,1,11,0)
%CALL SYRS(baud,handshake,dataformat,nullstrip,terminator$)
```

## Description

%CALL SYRS sets serial data communications configuration by specifying the baud rate, handshake, data format, and null strip behavior with specifiers defined in the tables on this and the following pages. If the data communications configuration is not defined by a %CALL SYRS statement, it defaults to 9600 baud, XON/XOFF enabled, seven data bits, one stop bit, even parity, null strip

disabled, and no terminate character (equivalent to %CALL SYRS(1,1,6,0)).

The data communications configuration used by the serial port is the one in effect when the serial port is opened. Therefore, %CALL SYRS should be executed before OPEN #1.

The default data communications configuration is independent of the configuration set by the B (baud rate) operating system command. If you have used the B command, it will have no effect on the behavior of the serial port when running a BASIC program.

**NOTE** The %CALL SYRS statement is for use only with the built-in serial port handler. If your OPEN # statement specifies a different serial port handler, use an assembly-language configuration subprogram for that handler. If you are using the bar-code serial port handler, use %CALL SYSP for configuring the serial port.

**Baud Rate Specifier**

| Specifier | Baud Rate |
|---|---|
| 1 (default) | 9600 |
| 2 | 4800 |
| 3 | 2400 |
| 4 | 1200 |
| 5 | 600 |
| 6 | 300 |
| 7 | 150 |

**Handshake Specifier**

| Specifier | Handshake |
|---|---|
| 0 | XON/XOFF disabled |
| 1 (default) | XON/XOFF enabled |

**Data Format Specifier**

| Specifier | Word Length (bits) | Stop Bits | Parity |
|---|---|---|---|
| 0 | 7 | 1 | none |
| 1 | 8 | 1 | none |
| 2 | 7 | 1 | odd |
| 3 | 8 | 1 | odd |
| 4 | 7 | 1 | none |
| 5 | 8 | 1 | none |
| 6 (default) | 7 | 1 | even |
| 7 | 8 | 1 | even |
| 8 | 7 | 2 | none |
| 9 | 8 | 2 | none |
| 10 | 7 | 2 | odd |
| 11 | 8 | 2 | odd |
| 12 | 7 | 2 | none |
| 13 | 8 | 2 | none |
| 14 | 7 | 2 | even |
| 15 | 8 | 2 | even |

Note: the duplicates in the table are correct – specifier 0 matches specifier 4, 1 matches 5, 8 matches 12, and 9 matches 13.

For 7-bit data only, specifiers 1 and 5 are equivalent to a word length of 7, 1 stop bit, and 0's parity. For 7-bit data only, specifiers 8 and 12 are equivalent to a word length of 7, 1 stop bit, and 1's parity.

**Null Strip Specifier**

| Specifier | Null Strip |
|---|---|
| 0 (default) | null strip disabled |
| 1 | null strip enabled |

When the null strip specifier is 1 (enable), null characters (ASCII 00h) are removed from the received data stream.

The optional terminate character specifies what character will signal the end of incoming data, thereby allowing variable length data input. The terminate character can be any character except null (ASCII 00h); specifying the null character or a null string is equivalent to having no terminate character.

The terminate character is also appended to each item output by `PRINT #` and `PRINT #...USING`. The optional end-of-line sequence sent by `PRINT #` and `PRINT #...USING` (carriage return-line feed) is considered a separate item, so a terminate character is sent after that

sequence as well.

**Serial Port Operation.** If XON/XOFF handshaking is enabled, the handheld will take the following actions: A single XON character (DC1, 11h) will be transmitted when the serial port is opened (with `OPEN #1,""`). One XOFF (DC3, 13h) will be sent for every character received in the 64-byte receive buffer, starting when 48 bytes have been received. A single XON will be sent when the handheld has read all the received data from the buffer. `HXBASIC` does not perform XON/XOFF handshaking or receive-data buffering.

The hardware lines used by the default serial port device handler are RTS (request to send), DTR (data terminal ready), and CTS (clear to send). When the serial port is opened, RTS and DTR are raised. In addition, Vcc is supplied to power the HP 82470A RS-232C Level Converter. When the serial port is closed (with `CLOSE #1`), RTS and DTR are lowered, and Vcc is no longer supplied.

The handheld will not transmit data unless CTS has been raised by the external device. `Error 218` will be reported if CTS remains low while attempting to transmit to indicate no device connected to the handheld. `Error 218` will also be reported if CTS drops in the middle of a transmission to indicate a lost connection.

The built-in serial port handler uses full-duplex operation. Because of the way RTS is controlled, half-duplex is not available with the built-in handler. Half-duplex operation is possible with an alternate device handler that uses RTS to control the communication direction (see below).

The hardware in the handheld has the ability to control RTS and DTR, and to monitor CTS, DSR (data set ready), and DCD (data carrier detect). The default serial port software does not take advantage of this hardware to monitor DSR or DCD, nor does it provide control for RTS and DTR other than described above. An alternate serial port handler could be written in assembly language that provides the ability to observe or ignore any or all of these lines, using whatever convention is required by the devices connected to the serial port. Refer to the *Technical Reference Manual* for details on writing device handlers and controlling and monitoring the serial port control lines.

**Differences Between Development System and Handheld.** `SYRS` is not implemented on the development system.

---

**NOTE** Do not use `SYRS` for bar-code readers attached to the serial port. Use `SYSP` instead.

---

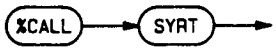
## Related Keywords

`CLOSE #, OPEN #`

# SYRT

Exists in Development System ☐
Works Same in Development System ☐
Allowed in IF...THEN ■

The %CALL SYRT (interrupt *Re*Turn) statement resumes execution of an interrupted program at the statement (or line) after the last one completed before the interrupt occurred.




## Examples

```
%CALL SYRT
```

## Description

%CALL SYRT is used to end an interrupt routine that specifies behavior for low battery, power switch, or timeout. When one of these conditions occurs, the interrupt-processing routine defined by the %CALL SYLB or %CALL SYSW statements will be executed. When %CALL SYRT is executed to end the interrupt routine, control will transfer to the *statement* after the last one completed before the interrupt occurred (on the same line, if the interrupt occurred in a multistatement line). However, if one of these events occurred during I/O to channels 0-4, the I/O is aborted. %CALL SYRT will then transfer control to the *line* (not statement) after the one which was interrupted. Refer to the keyword descriptions for the I/O statements (INPUT, INPUT #, INPUT$, GET #, PRINT #, PRINT #...USING, and PUT #) for a discussion of aborted I/O.

Because the low battery or power switch interrupts can occur anywhere in a program (i.e., during I/O or not), control will transfer to either the next statement or the next line after the interrupted line. Since timeouts can only occur during I/O to channels 0-4, control will always transfer to the next line after the interrupted line.

**Differences Between Development System and Handheld.** SYRT is not implemented on the development system.

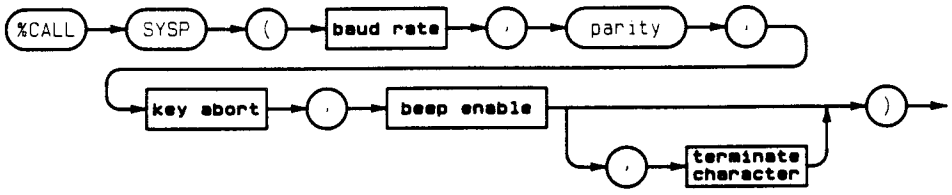
## Related Keywords

RETURN, SYLB, SYSW

# SYSP

- ☐ Exists in Development System
- ☐ Works Same in Development System
- ■ Allowed in `IF...THEN`

The `%CALL SYSP` (set *serial port* configuration) statement sets the data communications configuration used by the serial port (channel 1) for use with a bar-code reader.




| Item | Description | Range |
|---|---|---|
| `baud rate specifier` | numeric expression, truncated to an integer | 1 through 7 |
| `parity specifier` | numeric expression, truncated to an integer | 0 through 3 |
| `key-abort specifier` | numeric expression, truncated to an integer | 0 through 1 |
| `beep-enable specifier` | numeric expression, truncated to an integer | 0 through 1 |
| `terminate character` | string expression consisting of one character | — |

## Examples

```
%CALL SYSP(7,2,1,0)
%CALL SYSP(baud,parity,keyabort,beepenable,terminator$)
```

## Description

`%CALL SYSP` sets data communications configuration of the serial port by specifying the baud rate, parity, key abort option, beep enable, and terminate character with specifiers defined in the tables on this and the following pages. If the data communications configuration is not defined by a `%CALL SYSP` statement, it defaults to 9600 baud, 0 parity, good-read beep enabled, key abort enabled, and no

terminate character (equivalent to %CALL SYSP(1,0,1,1,"")). The handler allows operation with devices configured for either one or two stop bits.

**NOTE** The %CALL SYSP statement is for use only with the HNSP serial port handler. If your OPEN # statement specifies a different serial port handler, use an assembly-language configuration subprogram for that handler. If you are using the default serial port handler, use %CALL SYRS instead.

The data communications configuration used by the serial port is the one in effect before the serial port is opened. Therefore, %CALL SYSP should be executed before OPEN #1.

The default data communications configuration is independent of the configuration set by the B (baud rate) operating system command. If you have used the B command, it will have no effect on the behavior of the serial port when running a BASIC program.

**Baud Rate Specifier.** The baud rate specifiers for the serial port follow the same convention as those for the serial port. Options are:

| Specifier | Baud Rate |
|---|---|
| 1 (default) | 9600 |
| 2 | 4800 |
| 3 | 2400 |
| 4 | 1200 |
| 5 | 600 |
| 6 | 300 |
| 7 | 150 |

**Parity Specifier.** Four parity options are available:

| Specifier | Parity |
|---|---|
| 0 (default) | 0 parity |
| 1 | 1 parity |
| 2 | even parity |
| 3 | odd parity |

**Key-Abort Specifier.** The key-abort feature is designed to simplify handling of unreadable bar-codes. It enables the application program to allow for keyboard entry of what is usually bar-code information.

| Specifier | Key Abort |
|---|---|
| 0 | Key abort disabled |
| 1 (default) | Key abort enabled |

If key abort is enabled, pressing a key causes input from the bar-code port to end and control to return to the application program. The character for the key pressed will be in the key buffer. The value of the variable input differs whether the GET # or INPUT # statement was used. With GET #, the string variable is loaded with the null string. With INPUT #, the variable's contents remains what it was before the INPUT # statement was executed.

The five-step procedure outlined below provides a way for the application program to check whether a key abort has occurred. It works for both GET # and INPUT #.

1. Invoke error trapping using the %CALL SYER statement.
2. Set the error-number variable to 0.
3. Set the input variable to the null string. This is required for INPUT #, but optional for GET #.
4. Read the data using GET # or INPUT #.
5. Check the input and error-number variables. If the input variable is null and the error-number variable is 0, then bar-code entry was aborted from the keyboard.

The handheld will not transmit data unless CTS has been raised by the external device. Error 218 will be reported if CTS remains low while attempting to transmit to indicate no device connected to the handheld. Error 218 will also be reported if CTS drops in the middle of a transmission to indicate a lost connection.

**Beep-Enable Specifier.** If beep is enabled, the HP-94 will sound when it receives data from the serial port.

| Specifier | Good-Read Beep |
|---|---|
| 0 | good-read beep disabled |
| 1 (default) | good-read beep enabled |

**Terminate Character.** The optional terminate character specifies what character will signal the end of incoming data. The terminate character can be any character except null (ASCII 00h); specifying the null character or a null string is equivalent to having no terminate character. If no terminate character is specified, a delay of 104 milliseconds after receipt of a character ends the input operation.

The terminate character is also appended to each item output by PRINT # and PRINT #...USING. The optional end-of-line sequence sent by PRINT # and PRINT #...USING (CR LF) is considered a separate item, so a terminate character is sent after that sequence as well.

**NOTE** Unlike SYRS and the built-in serial port handler, the command SYSP and its associated handler HNSP are not resident in the handheld. Be sure to down-load these files from the development system to the handheld. See the *Utilities Reference Manual* chapter 3 "HXC File Conversion Utility" and chapter 6 "HXCOPY File Copy Utility" for details.

The bar-code handler for the serial port, HNSP will not work with so-called "dumb" bar-code devices (devices that return only a pulse train of light and dark transitions).

**Differences Between Development System and Handheld.** SYSP is not implemented on the development system.

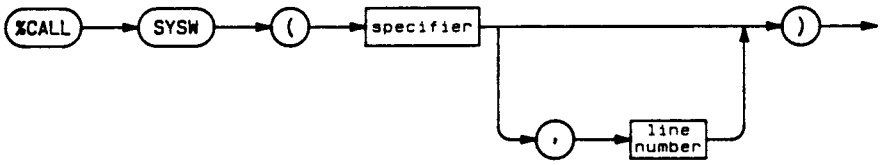
## Related Keywords

CLOSE #, OPEN #, SYBC, SYWN

# SYSW

- ☐ Exists in Development System
- ☐ Works Same in Development System
- ■ Allowed in `IF...THEN`

The `%CALL SYSW` (power *SW*itch behavior) statement specifies program behavior when the ON/OFF key is pressed or a timeout occurs.




| Item | Description | Range |
|---|---|---|
| `specifier` | numeric expression, truncated to an integer. | 0 through 3 |
| `line number` | numeric expression, truncated to an integer. | 0 through 32,767 |

## Examples

```
%CALL SYSW(0, 1000)
%CALL SYSW(behavior)
%CALL SYSW(0, lineno)
```

## Description

The default behavior when the power switch ( ON/OFF key) is pressed or when timeout occurs is for the handheld to turn off. The next time the power switch is pressed to turn on the handheld, it will cold start (refer to `SYPO` for details). `%CALL SYSW` specifies alternate behavior for the power switch and timeout. This alternate behavior will take effect when the handheld is already on and the power switch is pressed or the timeout occurs. (The effect of the power switch when the handheld is already off is either to cold start or warm start the machine, depending on how the handheld was turned off.)

When `%CALL SYSW` is used to specify a routine to process interrupts caused by the power switch or timeout, either event causes an error and, if error-trapping is enabled by `%CALL SYER`, causes program execution to branch to the specified interrupt-processing routine. The `SYSW` specifier determines the behavior during program execution when the power switch is pressed or a timeout occurs. The specifier can have four values, defined in the table below.

SYSW **Specifiers**

| Specifier | Meaning |
|---|---|
| 0 | Defines an interrupt-processing routine starting at the specified line number. When the power switch is pressed (unless disabled) or a timeout occurs (unless disabled), the program will complete the current BASIC statement (not line), and transfer control to this interrupt-processing routine. If I/O is occurring for channels 0-4, I/O will be aborted when the interrupt occurs. |
| 1 | Cancels the specified interrupt-processing routine. Restores power switch and timeout behavior to either the last behavior defined by a previous calling program or to the default behavior (unless disabled). |
| 2 | Disables the power switch. The power switch is ignored when pressed, but a timeout (unless disabled) is still processed by the specified interrupt-processing routine (or the default behavior). |
| 3 | Enables the power switch. Restores its behavior as defined by the specified interrupt-processing routine (or the default behavior). |

The I/O statements that can be interrupted during I/O to channels 0-4 are `INPUT`, `INPUT #`, `INPUT$`, `GET #`, `PRINT #`, `PRINT #...USING`, and `PUT #`.

**Error Trapping and Interrupt Routines.** `%CALL SYSW` defines the location of an interrupt routine that will be executed when the power switch is pressed or timeout occurs. This routine will only be executed if `%CALL SYER` with specifier 0 has been executed *in the program or subprogram which defines the interrupt routine* (contains the defining `%CALL SYSW`). The following table shows the behavior of the power switch and timeout depending on whether `SYSW` and `SYER` are used together.

SYSW **and** SYER **Interaction**

| Usage | Power Switch Behavior | Timeout Behavior |
|---|---|---|
| Neither `SYSW` nor `SYER` | Handheld turns off. | Handheld turns off. |
| `SYER` only | Handheld turns off. | Handheld turns off. |
| `SYSW` only | Program halts with `Error 119`. | Program halts with `Error 118`. |
| Both `SYSW` and `SYER` | Program execution transfers to the interrupt-processing routine. If I/O to channels 0-4 was interrupted, error number variable set to 119, and `SYIN` variables updated. | Program execution transfers to the interrupt-processing routine. Error number variable always set to 118, and `SYIN` variables always updated. |

For error trapping to be enabled within the interrupt routine itself, %CALL SYER with specifier 0 must also be executed in the interrupt routine. If it is not, the behavior of the power switch and timeout during the routine will be identical to the "SYSW only" behavior in the previous table.

The error number variable is always local to the currently executing program or subprogram (in this case, the interrupted program). To allow examining the error number in an interrupt routine that is in a different program than the interrupted program (see next topic), the error number variable should be passed (by reference) to different subprograms as a parameter when the subprogram is CALLed. That way any changes to the error variable in a subprogram will be indicated in any calling program that passed the variable.

**Global and Local Control.** SYSW provides both global and local control of the behavior of the power switch and timeout. Global control occurs when an interrupt-processing routine is defined in the main program. This routine will be in effect whenever the main program or any subprogram it calls is executing. Local control occurs when a subprogram defines a new interrupt-processing routine. The new routine will be in effect for that subprogram and for any subprograms it calls. When the subprogram ends, the power-switch and time-out behaviors revert to those in effect before the subprogram was called.

This local-versus-global control is true no matter how deep subprograms are nested. That is, an interrupt-processing routine in a calling program is no longer in effect when the called subprogram defines its own interrupt-processing routine, but is reactivated when the subprogram cancels its routine or ends.

**Interaction Between Power Switch and Timeout.** The power switch and timeout interrupts both cause the same interrupt routine to be executed. This is because frequently the desired behavior will be the same for both situations. If the user tries to turn the handheld off (power switch), or if the handheld tries to turn itself off (timeout), the program can treat both events identically. For example, when waiting for keyboard input, either event could cause the program to save certain data or status, and use %CALL SYPO to turn the machine off, perhaps specifying a subsequent warm start to allow the user to return to the previous activities.

**Distinguishing Between Power Switch and Timeout.** An interrupt-processing routine can determine which event started its execution by looking at the error variable. However, the variable will be set to the power switch error, 119, only when I/O to channels 0-4 is interrupted. If the power switch is pressed during execution of non-I/O BASIC statements, it is not considered an error condition, so the error variable will not be changed from its current value. Consequently, the value in the error variable may not give a true indication of whether or not the power switch was pressed.

Similarly, the error variable will only be set to the timeout error, 118, if a timeout interrupted I/O to channels 0-4. Unlike the power switch, which can be pressed at any point in a BASIC program and still transfer control to the interrupt routine, I/O to channels 0-4 is the only condition in which a timeout can occur. Therefore, the best way to distinguish which event invoked the interrupt routine is to check if the error variable is set to the timeout error, 118. If so, timeout occurred. If not, the power switch was pressed, since that is the only other event that can transfer control to the interrupt routine.

Note: the low battery interrupt has the same characteristics as the power switch interrupt. It can occur at any time in a program, and will only be considered an error (and thus set the error variable to the low battery error, 200) if I/O to channels 0-4 is interrupted. It is recommended that you define the low battery interrupt routine to be different than the power switch interrupt routine if you need to distinguish which event caused the interrupt.

**Disabling the Power Switch or Timeout.** There are situations in which the power switch and timeout need to be enabled and disabled independently of each other. For example, the program may want to ensure that certain critical operations (such as updating important files or data communications) do not get interrupted by the user pressing the power switch. To allow this additional power switch control, specifier 2 disables the power switch, and specifier 3 enables the switch again.

The timeout will still be in effect while the power switch is disabled. This can be used in situations (particularly data communications) where there is no keyboard input, but other I/O is occurring, such as to the serial port. The program can use the timeout to monitor long I/O operations with a local interrupt-processing routine, and the power switch will not have any effect.

If it is desired to prevent the handheld from turning itself off, the timeout can be disabled with `%CALL SYTO(0)`. The power switch will still be in effect while the timeout is disabled, allowing the user the option of turning off the machine (depending on the whether or not there is an interrupt-processing routine defined for the power switch). The timeout can be enabled by providing a timeout value. The default timeout is 120 seconds (two minutes), set by `%CALL SYTO(120)`.

Both the power switch and timeout can be disabled, so the machine will neither respond to the power switch or turn itself off. While either or both events are disabled, you can still define or cancel the interrupt-processing routine with specifiers 0 or 1. The last behavior of the power switch or timeout will take effect when either the power switch or timeout are enabled again, even if the behavior was changed while the events were disabled.

When the power switch is disabled, particularly during program development and debugging, you may not be able to turn the handheld off. Just use a pencil or paper clip to push the reset switch (the small hole next to the power switch). The next time the handheld turns on after being turned off by the reset switch, it will cold start.

**Data File Integrity.** When a power switch or timeout interrupt occurs during execution of non-I/O BASIC statements, the current statement will finish executing before control is transferred to the interrupt-processing routine. Only BASIC statements performing I/O to channels 0-4 will be interrupted, and will not complete their I/O, because of power switch or timeout interrupts (aborted I/O is discussed in the keyword descriptions for the I/O statements). If data is being written into a file, the write operation will be completed before the interrupt routine begins to execute. You do not have to worry about files being corrupted because of a partially-completed (interrupted) file write operation.

However, you do have to worry about files being properly updated if the program executes several file write statements, and the interrupt occurs before all the statements have been completed. In that situation, if the program turns off the machine and specifies a subsequent cold start (`%CALL SYPO(0)`), the file update will be incomplete. For this reason, you may want to write your application so that it completes the file update operation before turning off the power, or to disable the power switch during critical file updates. (At a subsequent warm start, the program will continue execution, and can then complete the file update process with no loss of file integrity.)

**Behavior During Interrupt-Processing Routines.** The global control of the power switch and timeout means that program execution will transfer to the interrupt-processing routine *even if the interrupt occurs in a different subprogram than the one containing the interrupt routine.* Because this is effectively a subroutine call that can cross program boundaries, RETURN cannot be used to return from the interrupt routine to the program that was executing when the interrupt occurred. Instead, `%CALL SYRT` must be used to return from the interrupt routine.

The CALL statement will give an error if it is used within an interrupt-processing routine. If the END statement is executed in an interrupt-processing routine, the program or subprogram will end and return control back to the operating system (not to the calling (sub)program).

With the previous exceptions, any BASIC statement can be executed in an interrupt routine, including %CALL SYSW (and %CALL SYLB ) with any specifier. Once the interrupt routine ends, any subsequent interrupt will be processed according to conditions defined by the most recently executed SYSW statement. New interrupts can still occur while an interrupt-processing routine is executing. However, they do not cause that or any other interrupt routine (such as power switch) to be executed. During execution of non-I/O BASIC statements in an interrupt routine, new interrupts are not processed until the interrupt routine ends. Because of this, interrupt routines should be as short as possible.

During execution of I/O statements to channels 0-4 in an interrupt routine, new interrupts still cause I/O to be aborted and the error number variable to be changed, but the interrupt routine itself is not reexecuted. (Recall that for this to occur, %CALL SYER with specifier 0 must also be executed in the interrupt routine.)

An interrupt-processing routine has a separate READ data pointer than the one used in the non-interrupt portion of the program. All DATA statements are treated identically, regardless of where they appear in the program (interrupt routine or non-interrupt routine). When the READ statement is executed in an interrupt-processing routine, it will start reading at the first DATA statement in the program, regardless of how many data items had been read before the interrupt routine was executed. When the interrupt routine ends, subsequent READ statements in the non-interrupt portion of the program will continue reading DATA statements as if the interrupt routine had not been executed. That is, reading will start after the last data item read before the interrupt routine was executed.

In an interrupt routine, RESTORE will affect the interrupt routine's data pointer independently of the non-interrupt routine's data pointer.

---

**CAUTION** The line number specified by %CALL SYSW with specifier 0 will *not* be renumbered by the HXBASIC R (renumber) command. If you renumber your program, make sure you update the SYSW line number (or the value placed in the variable used for the line number).

---

**Differences Between Development System and Handheld.** SYSW is not implemented on the development system.

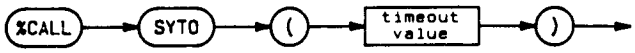
## Related Keywords

DATA, GET #, INPUT, INPUT #, INPUT$, PRINT #, PRINT #...USING, PUT #, READ, SYER, SYIN, SYLB, SYPO, SYRT, SYTO

# SYTO

Exists in Development System ☐
Works Same in Development System ☐
Allowed in IF...THEN ■

The %CALL SYTO (*TimeOut*) statement sets the time period of inactivity after which the handheld is turned off.




| Item | Description | Range |
|---|---|---|
| timeout value | numeric expression, truncated to an integer. | 0 through 1,800 |

## Examples

```
%CALL SYTO(0)
%CALL SYTO(60)
```

## Description

For the purpose of measuring the timeout period, *inactivity* is defined as the state where the handheld is waiting for keyboard or bar-code port input or serial port input or output. The timeout value is in seconds, allowing a range of 1 second to 30 minutes. A timeout value of 0 specifies that timeout is disabled; the handheld will wait indefinitely for input. If no other timeout value is specified by a %CALL SYTO statement, the timeout value defaults to 120 seconds (two minutes).

When no alternate timeout behavior has been specified by the %CALL SYSW statement, timeout causes the handheld to turn off, then to cold start when the power switch is pressed. %CALL SYSW specifies alternate behavior for the power switch and, unless timeout has been disabled, for timeout also. Refer to the keyword description for SYSW for complete details.

**Differences Between Development System and Handheld.** SYTO is not implemented on the development system.

## Related Keywords

SYEL, SYSW

# SYWN

- ☐ Exists in Development System
- ☐ Works Same in Development System
- ■ Allowed in `IF...THEN`

The `%CALL SYWN` (set *WaNd* configuration) statement sets the configuration used for the HP Smart Wand.


| Item | Description | Range |
|---|---|---|
| `escape specifier` | numeric expression, truncated to an integer | 0 through 1 |
| `channel number` | numeric expression, truncated to an integer | 1 through 2 |

## Examples

```
%CALL SYWN(1,2)
%CALL SYWN(escape_yes,channel)
```

## Description

`%CALL SYWN` sets options to the high-level handler for the HP Smart Wand. This handler works only if the bar-code device contains HP Smart Wand firmware version 12.3 or later and the handheld's operating system is version 1.03 or later. It will not work with other smart bar-code devices nor will it work with "dumb" wands (wands that return only a pulse train of light and dark transitions). The high-level smart wand handler can utilize either the bar-code port or the serial port.

The primary purpose of the high-level handler for the HP Smart Wand is to provide special processing of escape sequences sent by the Smart Wand to the handheld.

The options used by the high-level handler are the ones in effect when the port is opened. Therefore, %CALL SYWN should be executed before OPEN #.

**Escape Specifier.** The escape specifier controls whether escape sequences sent by the Smart Wand to the handheld will be passed to the calling program. If escape is turned off, the handheld will ignore all strings that begin with "$^{E}c\$". There is no beep and the string sequence is not passed to the application program. This mode may be used to prevent configuration messages from getting into the handheld's data files.

| Specifier | Escape Sequences |
|---|---|
| 0 (default) | escape sequences disabled |
| 1 | escape sequences enabled |

If escape sequences are enabled, strings beginning with "$^{E}c\$" are transmitted to the calling program. The following HP Smart Wand escape sequences are supported.

| Escape Sequence | Meaning | Beep Sound |
|---|---|---|
| Configuration complete ($^{E}c*$) | Smart Wand has completed the configuration operation specified by the bar code. | 4 high-pitched beeps |
| Configuration partially complete ($^{E}c\+$). | Smart Wand has completed a portion of the configuration operation. This is sent for intermediate steps in configuration operations requiring more than one scan. | 2 high-pitched beeps |
| Syntax error ($^{E}c\-$) | Configuration menu was out of context. This may be caused by scanning configuration bar codes in the wrong order, that are of the wrong type, or that are numerically out of range. | 4 low-pitched beeps |
| Configuration dump ($^{E}c*^{C}R^{L}F^{E}c$ [...] $^{C}R^{L}F$) | Status information about the Smart Wand. | |
| Hard-Reset Message (ready *w.v*) | The configuration bar code specifying a hard reset has been scanned. | |
| No-read message (user-defined, default is $^{C}R^{L}F$) | The Smart Wand is unable to decode the bar code and no-read message is enabled. | |

These beeps will sound whether or not beeps are enabled using SYBC or SYSP.

**Channel Number.** This numeric expression specifies to which port the bar-code reader is connected.

| Specifier | Channel |
|---|---|
| 1 | serial port |
| 2 | bar-code port |

**NOTE** Unlike SYRS and the built-in serial port handler, the command SYSP and its associated handler HNSP are not resident in the handheld. Be sure to down-load these files from the development system to the handheld. See the *Utilities Reference Manual* chapter 3 "HXC File Conversion Utility" and chapter 6 "HXCOPY File Copy Utility" for details.

The bar-code handler for the serial port, HNSP will not work with so-called "dumb" bar-code devices (devices that return only a pulse train of light and dark transitions).

**Differences Between Development System and Handheld.** SYWN is not implemented on the development system.

## Related Keywords

CLOSE #, OPEN #, SYBC, SYSP

# TAN

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The TAN function interprets the numeric argument as an angle measured in degrees, and returns the tangent of the angle.


| Item | Description | Range |
|---|---|---|
| numeric argument | numeric expression | — |

## Examples

```
Tangent=TAN(Theta)
Vertical=Horizontal*TAN(x)
```

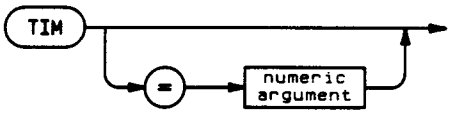
## Related Keywords

ACS, ADS, ARD, ASN, ATN, COS, DMS, PI, RAD, SIN

# TIM

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The `TIM` function returns the current time or sets the time for the real-time clock.




| Item | Description | Range |
|---|---|---|
| `numeric argument` | numeric expression. | See description below. |

## Examples

```
C=ADS(TIM)
IF ADS(TIM)<ADS(start)+ADS(.0005) GOTO 100
```

## Description

Invoking `TIM` without an argument returns the current time from the real-time clock.

Passing an argument to `TIM` will cause the function to set the time.

The format of the argument and value returned is a floating point number in the format *HH.MMSS* as shown in the table below.

| Item | Description | Range |
|---|---|---|
| *HH* | Hours in 24-hour clock format. | 0 through 24 |
| *MM* | Minutes. | 00 through 59 |
| *SS* | Seconds. | 00 through 59 |

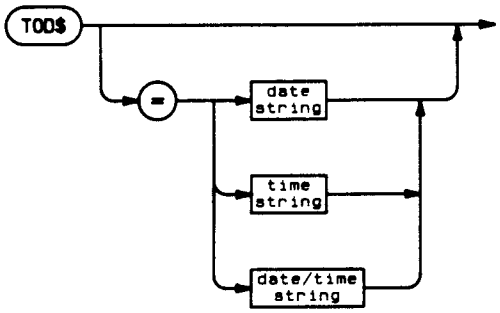
## Related Keywords

ADS, DMS, TOD$

# TOD$

Exists in Development System ■
Works Same in Development System □
Allowed in IF...THEN ■

The TOD$ function returns the current time and date from the real-time clock, or sets the time and date.




| Item | Description | Range |
|---|---|---|
| date string | string expression in the form *MM/DD/YY* (month/day/year) | *MM*: 01 through 12; *DD*: 01 through 31; *YY*: 00 through 99 |
| time string | string expression in the form *HH:MM:SS* (hour:minute:second) | *HH*: 00 through 23; *MM*: 00 through 59; *SS*: 00 through 59 |
| date/time string | string expression in the form *MM/DD/YY*,HH:MM:SS | same as for date string and time string |

## Examples

```
TOD$="11/17/85" : REM Set the date
TOD$="15:25:30" : REM Set the time
TOD$="11/17/85,15:25:30" : REM Set the date and time
Today$=TOD$ : REM Read the date and time
```

# ...TOD$

## Description

TOD$ either sets or reads the real-time clock. To set the clock, either a date string, time string, or date/time string must be supplied as the argument. To read the clock, TOD$ returns a 17-character date/time string. Either the TOD$ function or the string returned by it can be used like any string variable: assigned to another string variable, supplied to functions or statements that accept string arguments, etc.

**Differences Between Development System and Handheld.** On the development system, the clock is set the same way as on the handheld. When the clock is read, however, the date/time string is 22 characters long instead of 17, and includes the day of the week in the following format: *MM/DD/YY(DOW),HH:MM:SS*, where *DOW* is a three-character abbreviation for the day of the week (SUN, MON, TUE, WED, THU, FRI, or SAT).

**NOTE** The difference in date/time string lengths is a useful debugging tool that allows a program to determine which machine it is running on. Programs running on the development system can branch around statements that the development system does not support, or execute statements that perform similar functions.

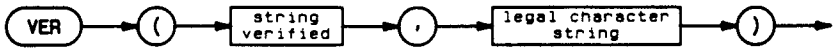
## Related Keywords

TIM

# VER

Exists in Development System ■
Works Same in Development System ■
Allowed in IF...THEN ■

The VER function verifies that a string contains only specific characters.




| Item | Description | Range |
|---|---|---|
| string verified | string expression | — |
| legal character string | string expression | — |

## Examples

```
Badposition=VER(A$,"1234567890")
IF VER(A$,"ABCDEFGH") THEN PRINT "Illegal input"
```

## Description

If the string verified contains only characters that are in the legal character string, VER returns 0. If the string verified contains characters that are not in the legal character string, VER returns the position of the first such character. VER(**""**,A$) always returns 0; VER(A$,**""**) returns 1 (0 if A$ is the null string).

## Related Keywords

STR$

# XOR

- Exists in Development System
- Works Same in Development System
- Allowed in `IF...THEN`

The XOR operator returns the bit-by-bit exclusive-OR of the binary representation of the operands.


| Item | Description | Range |
|---|---|---|
| operand | numeric expression | -32,768 through +32,767 |

## Examples

```
IF S<>0 XOR P<>0 THEN GOSUB 400
S=J(1) XOR J(2)
```

## Description

The operands are truncated to integers represented as two's-complement. The results of each bit-by-bit XOR are used to construct the integer result. Each bit is computed according the following truth table.

**Bit-by-Bit XOR**

| Operand 1 | Operand 2 | Result |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Relational operators (=, <, >, <=, >=, <>) always return -1 for true and 0 for false. The bit-by-bit XOR of these results will always be 0 or -1.

## Related Keywords

AND, NOT, OR

# A

# BASIC Keyword Summary

| Keyword | Summary | Page |
|---|---|---|
| ABS | Absolute value. | 2-3 |
| ACS | Arccosine (1st or 2nd quadrant). | 2-4 |
| ADS | Converts a value from degrees, minutes and seconds into decimal degrees. | 2-5 |
| AND | Bit-by-bit AND of two values. | 2-6 |
| ARD | Converts an angle from radians to degrees. | 2-8 |
| ASC$ | Interprets a numeric value as a character code and returns the character. | 2-9 |
| ASN | Arcsine (1st or 4th quadrant). | 2-11 |
| ATN | Arctangent (1st or 4th quadrant.) | 2-12 |
| CALL | Calls a subprogram and optionally passes parameters. | 2-13 |
| %CALL | Calls an assembly language program and optionally passes parameters. | 2-15 |
| CHR$ | Returns the string equivalent of a value. | 2-18 |
| CLOSE # | Closes a device or file. | 2-19 |
| COD | Returns decimal code of first character in string. | 2-21 |
| COS | Cosine. | 2-22 |
| DATA | Specifies data items for READ. | 2-23 |
| DEF FN | Defines a user-defined function. | 2-25 |
| DIM | Reserves memory for arrays and strings. | 2-27 |
| DMS | Converts a value from decimal degrees into degrees, minutes, and seconds. | 2-31 |
| END | Returns program execution to the calling (sub)program, or halts main program execution. | 2-32 |
| EOF | Detects the end-of-file. | 2-33 |
| EXP | $e^x$ | 2-34 |
| FIX0 | Rounds a fraction down. | 2-35 |
| FIX5 | Rounds a fraction off. | 2-36 |
| FIX9 | Rounds a fraction up. | 2-38 |

**Keyword Summary (continued)**

| Keyword | Summary | Page |
|---|---|---|
| FIXE | Rounds off the mantissa of a number in scientific notation. | 2-39 |
| FN | User-defined function call. | 2-40 |
| FOR...NEXT | Defines a FOR...NEXT loop. | 2-42 |
| FORMAT | Provides formats for PRINT USING and PRINT #...USING. | 2-45 |
| FRC | Fractional part of a number. | 2-48 |
| GET # | Inputs data from a device or file into program variables. | 2-49 |
| GOSUB | Causes branching to a subroutine. | 2-53 |
| GOTO | Causes branching to a statement. | 2-54 |
| HEX$ | Converts decimal value to a two-character string containing its hexadecimal representation. | 2-55 |
| IDX | Position of a character in a string. | 2-56 |
| IF...THEN | Causes conditional branching. | 2-58 |
| INPUT | Inputs data from the keyboard into program variables. | 2-60 |
| INPUT # | Inputs data from a device or file into program variables. | 2-62 |
| INPUT$ | Inputs data from a device or file into program variables. | 2-65 |
| INT | Integer part of a number. | 2-68 |
| INTEGER | Declares variables and arrays to be integers. | 2-69 |
| KEY | Number of bytes in the key buffer or serial port buffer. | 2-71 |
| LEN | Length of a string. | 2-73 |
| LET | Variable assignment. | 2-74 |
| LGT | Log to the base 10. | 2-76 |
| LOG | Log to the base e. | 2-77 |
| MAX | Larger of a group of values. | 2-78 |
| MIN | Smaller of a group of values. | 2-79 |
| MOD | Modulo operator; remainder of division. | 2-80 |
| NOT | Bit-by-bit NOT of a value. | 2-81 |
| NUM | Numeric equivalent of a string. | 2-82 |
| ON...GOSUB/GOTO | Computed GOSUB and GOTO. | 2-84 |
| OPEN # | Opens a device or file for reading or writing. | 2-86 |
| OPTION BASE | Declares lower bound of 0 for array variables. | 2-89 |
| OR | Bit-by-bit OR of two values. | 2-90 |
| PARAM | First statement of a subprogram; defines the formal parameters. | 2-92 |
| PI | $\pi$ | 2-94 |
| PRINT | Displays items on the display. | 2-95 |

## Keyword Summary (continued)

| Keyword | Summary | Page |
|---|---|---|
| PRINT USING | Displays items on the display in user-defined format | 2-99 |
| PRINT # | Outputs items to a device or file. | 2-101 |
| PRINT #...USING | Outputs items to a device or file in user-defined format. | 2-107 |
| PUT # | Outputs items to a device or file. | 2-111 |
| RAD | Converts angle from degrees into radians. | 2-115 |
| READ | Reads items from DATA statements. | 2-116 |
| REM | Program comment. | 2-118 |
| RESTORE | Resets pointer for DATA statements. | 2-119 |
| RETURN | Transfers control from a subroutine to the statement following the calling GOSUB. | 2-120 |
| RND | Random number. | 2-121 |
| SGN | Sign of a number. | 2-122 |
| SIN | Sine. | 2-123 |
| SIZE | Amount of available memory. | 2-124 |
| SQR | Square root. | 2-125 |
| STR$ | Extracts substring. | 2-126 |
| SYAL | Creates a data file. | 2-128 |
| SYBC | Sets bar-code port configuration for HNBC. | 2-130 |
| SYBP | Produces an audible tone. | 2-134 |
| SYEL | Sets timeout value for electroluminescent display backlight. | 2-136 |
| SYER | Establishes the variable where error numbers will be placed. | 2-138 |
| SYIN | Program name, line number, channel number, and I/O length of most recent error. | 2-141 |
| SYLB | Establishes location of low battery interrupt routine. | 2-143 |
| SYPO | Powers off the computer. | 2-148 |
| SYPT | Moves the data pointer within a data file. | 2-150 |
| SYRS | Sets serial port configuration for default handler. | 2-151 |
| SYRT | Transfers control from an interrupt routine to the statement following the statement where the interrupt occurred. | 2-155 |
| SYSP | Sets serial port configuration for HNSP. | 2-156 |
| SYSW | Establishes location of power switch and timeout interrupt routines. | 2-160 |
| SYTO | Sets timeout value. | 2-165 |
| SYWN | Sets configuration for HP Smart Wand handler HNWN. | 2-166 |
| TAN | Tangent. | 2-169 |

## Keyword Summary (continued)

| Keyword | Summary | Page |
|---|---|---|
| TIM | Sets or returns the current time in HH.MMSS format. | 2-170 |
| TOD$ | Sets or returns the time and date in MM/DD/YY,HH:MM:SS format. | 2-171 |
| VER | Verifies that a string contains only specific characters. | 2-173 |
| XOR | Bit-by-bit exclusive-OR of two values. | 2-174 |

# B

# Numeric and Non-Numeric Errors

The handheld can produce two types of errors: numeric and non-numeric. Numeric errors are errors identified by a three-digit number, and non-numeric errors by two alphabetic characters. The most important distinction between numeric and non-numeric errors is that numeric errors can be trapped under program control, whereas non-numeric errors cannot be trapped.

**Numeric Errors.** When no alternate behavior has been specified for error processing, numeric errors cause the BASIC program to halt. Control returns to the operating system with the message `Error` *NNN LLLLL PPPP*, where *NNN* is the error number, *LLLLL* is the BASIC program line number and PPPP is the name of the BASIC program or subprogram in which the error occurred. The `SYER` statement can specify that these errors be ignored, so that program execution continues uninterrupted. After `CALL SYER` with specifier 0 has been executed, when a numeric error does occur, its error number is assigned to the error number variable, allowing programmatic error handling or error trapping. The program name, line number, channel number, and I/O length of the most recent error can be determined with the `SYIN` statement.

**Non-Numeric Errors.** `SYER` has no effect on non-numeric errors. These errors always cause the program to halt, and return control to the operating system with the message `Error` *MM LLLLL PPPP*, where *MM* is the error message, *LLLLL* is the BASIC program line number, and *PPPP* is the name of the BASIC program or subprogram in which the error occurred. The tables on the subsequent pages list all the numeric and non-numeric errors.

## Numeric Errors (from Operating System)

| Number | Meaning |
|---|---|
| 100* | BASIC interpreter not found. |
| 101 | Illegal parameter. |
| 102 | Directory does not exist. |
| 103 | File not found. |
| 104 | Too many files. |
| 105 | Channel not open. |
| 106 | Channel already open. |
| 107 | File already open. |
| 108 | File already exists. |
| 109 | Read-only access. |
| 110 | Access restricted. |
| 111 | No room for file. |
| 112 | No room to expand file. |
| 113 | No room for scratch area. |
| 114 | Scratch area does not exist. |
| 115† | Short record detected. |
| 116† | Terminate character detected. |
| 117† | End of data. |
| 118 | Timeout. |
| 119 | Power switch pressed. |
| 200 | Low battery. |
| 201 | Receive buffer overflow. |
| 202 | Parity error. |
| 203 | Overrun error. |
| 204 | Parity and overrun error. |
| 205 | Framing error. |
| 206 | Framing and parity error. |
| 207 | Framing and overrun error. |
| 208 | Framing, overrun, and parity error. |
| 209† | Invalid MDS file received. |
| 210* | Low backup battery – main memory. |
| 211* | Low backup battery – 128K memory board or 40K RAM card. |
| 212* | Checksum error – main memory directory table. |
| 213* | Checksum error – 40K RAM or ROM/EPROM card directory table. |
| 214* | Checksum error – reserved scratch space. |
| 215* | Checksum error – main memory free space. |
| 216* | Checksum error – main memory file. |
| 217* | Checksum error – 40K RAM or ROM/EPROM card file. |
| 218 | Lost connection while transmitting. |
| 219† | Illegal use of operating system stack. |

* Only reported when handheld is being turned on.

† Never reported by built-in BASIC keywords.

**Non-Numeric Errors (from BASIC Interpreter)**

| Message | Meaning |
|---|---|
| AR | Array subscript error |
| BM | BASIC interpreter malfunction |
| BR | Branch destination error |
| CN | Data conversion error |
| CO | Conversion overflow |
| DO | Decimal overflow |
| DT | Data error |
| EP | Missing END statement |
| FN | Illegal DEF FN statement |
| IL | Illegal argument |
| IR | Insufficient RAM |
| IS | Illegal statement |
| LN | Nonexistent line |
| MO | Memory overflow |
| NF | Program not found |
| RT | RETURN or SYRT error |
| SY | Syntax error |
| TY | Data type mismatch |
| UM | Unmatched number of arguments |

# C

# Keyboard Layout

The handheld allows the font for 32 characters to be redefined: character codes 128-159 (or 80h-9Fh), the control codes for the upper 128 characters of the character set. Refer to the *HP-94 Technical Reference Manual* for details on creating user-defined characters.

The handheld also allows 16 keys to be redefined: character codes 128-143 (or 80h-8Fh), the first 16 control codes for the upper 128 characters of the character set. Normally, when one of these keys is pressed, the handheld displays a blank character. If there is a user-defined character for the code corresponding to that key, the handheld will display the user-defined character instead. The handheld will echo user-defined characters when the corresponding keys are pressed only during a running program. It will echo a blank character when those keys are pressed while using the operating system commands.

## ASCII Characters Associated With Each Key

| Shifted Key (orange) | Shifted Character | Unshifted Key (white) | Unshifted Character |
|---|---|---|---|
| A | A (41h) | (unmarked) | user-def. (80h) |
| B | B (42h) | (unmarked) | user-def. (81h) |
| C | C (43h) | (unmarked) | user-def. (82h) |
| D | D (44h) | (unmarked) | user-def. (83h) |
| E | E (45h) | (unmarked) | user-def. (84h) |
| F | F (46h) | (unmarked) | user-def. (85h) |
| G | G (47h) | (unmarked) | user-def. (86h) |
| H | H (48h) | 7 | 7 (37h) |
| I | I (49h) | 8 | 8 (38h) |
| J | J (4Ah) | 9 | 9 (39h) |
| K | K (4Bh) | (unmarked) | user-def. (87h) |
| L | L (4Ch) | (unmarked) | user-def. (88h) |
| M | M (4Dh) | (unmarked) | user-def. (89h) |
| N | N (4Eh) | 4 | 4 (34h) |
| O | O (4Fh) | 5 | 5 (35h) |
| P | P (50h) | 6 | 6 (36h) |
| Q | Q (51h) | (unmarked) | user-def. (8Ah) |
| R | R (52h) | (unmarked) | user-def. (8Bh) |
| S | S (53h) | (unmarked) | user-def. (8Ch) |
| T | T (54h) | 1 | 1 (31h) |
| U | U (55h) | 2 | 2 (32h) |
| V | V (56h) | 3 | 3 (33h) |
| W | W (57h) | (unmarked) | user-def. (8Dh) |
| W | X (58h) | (unmarked) | user-def. (8Eh) |
| Y | Y (59h) | (unmarked) | user-def. (8Fh) |
| Z | Z (5Ah) | 0 | 0 (30h) |
| ← | (delete) (7Fh) | ← | (delete) (7Fh) |
| ENTER | (car. ret.) (0Dh) | ENTER | (car. ret.) (0Dh) |
| CLEAR | (control-X) (18h) | CLEAR | (control-X) (18h) |
| SPACE | (space) (20h) | 00 | (dbl 0) (30h 30h) |
| * | * (2Ah) | # | # (23h) |
| - | - (2Dh) | - | - (2Dh) |
| . | . (2Eh) | . | . (2Eh) |

# D

# Roman-8 Character Set

The Roman-8 character set consists of the standard U.S. ASCII character set and the Roman Extension character set. Each character in the set is assigned a character code with a decimal value from 0 through 255. The system uses these codes for identifying characters. Characters are normally produced from the keyboard by pressing the corresponding keys. (Refer to appendix C for the keyboard layout.)

The first half of the Roman-8 character set (decimal values 0 through 127, the U.S. ASCII character set) is identical to the standard character set used on many other computer systems. The second half (decimal values 128 through 255) contains special characters, including those used by other languages. The Roman-8 character set is shown in the tables on the following pages.

The handheld allows the font for 32 characters to be redefined (character codes 128 through 159-the control codes for the upper 128 characters of the character set). The development system does not support user-defined characters. When character codes in the user-defined range are used, the development system will always map them to blanks, and the handheld will map them to user-defined characters.

The handheld uses the Roman-8 character set. The development system may use either the Roman-8 or the IBM-compatible character sets as set by the HXCHRSET utility described in chapter 5 of the *Utilities Reference Manual*. The difference between the two character sets occurs in the control codes (`ASC$(0)` through `ASC$(31)`) and in the upper half of the character set (`ASC$(128)` through `ASC$(255)`).

# US ASCII Character Set

| ASCII Char. | Character Code | | | |
|---|---|---|---|---|
| | Dec | Binary | Oct | Hex |
| NUL | 0 | 00000000 | 000 | 00 |
| SOH | 1 | 00000001 | 001 | 01 |
| STX | 2 | 00000010 | 002 | 02 |
| ETX | 3 | 00000011 | 003 | 03 |
| EOT | 4 | 00000100 | 004 | 04 |
| ENQ | 5 | 00000101 | 005 | 05 |
| ACK | 6 | 00000110 | 006 | 06 |
| BEL | 7 | 00000111 | 007 | 07 |
| BS | 8 | 00001000 | 010 | 08 |
| HT | 9 | 00001001 | 011 | 09 |
| LF | 10 | 00001010 | 012 | 0A |
| VT | 11 | 00001011 | 013 | 0B |
| FF | 12 | 00001100 | 014 | 0C |
| CR | 13 | 00001101 | 015 | 0D |
| SO | 14 | 00001110 | 016 | 0E |
| SI | 15 | 00001111 | 017 | 0F |
| DLE | 16 | 00010000 | 020 | 10 |
| DC1 | 17 | 00010001 | 021 | 11 |
| DC2 | 18 | 00010010 | 022 | 12 |
| DC3 | 19 | 00010011 | 023 | 13 |
| DC4 | 20 | 00010100 | 024 | 14 |
| NAK | 21 | 00010101 | 025 | 15 |
| SYNC | 22 | 00010110 | 026 | 16 |
| ETB | 23 | 00010111 | 027 | 17 |
| CAN | 24 | 00011000 | 030 | 18 |
| EM | 25 | 00011001 | 031 | 19 |
| SUB | 26 | 00011010 | 032 | 1A |
| ESC | 27 | 00011011 | 033 | 1B |
| FS | 28 | 00011100 | 034 | 1C |
| GS | 29 | 00011101 | 035 | 1D |
| RS | 30 | 00011110 | 036 | 1E |
| US | 31 | 00011111 | 037 | 1F |

| ASCII Char. | Character Code | | | |
|---|---|---|---|---|
| | Dec | Binary | Oct | Hex |
| space | 32 | 00100000 | 040 | 20 |
| ! | 33 | 00100001 | 041 | 21 |
| " | 34 | 00100010 | 042 | 22 |
| # | 35 | 00100011 | 043 | 23 |
| $ | 36 | 00100100 | 044 | 24 |
| % | 37 | 00100101 | 045 | 25 |
| & | 38 | 00100110 | 046 | 26 |
| ' | 39 | 00100111 | 047 | 27 |
| ( | 40 | 00101000 | 050 | 28 |
| ) | 41 | 00101001 | 051 | 29 |
| * | 42 | 00101010 | 052 | 2A |
| + | 43 | 00101011 | 053 | 2B |
| , | 44 | 00101100 | 054 | 2C |
| - | 45 | 00101101 | 055 | 2D |
| . | 46 | 00101110 | 056 | 2E |
| / | 47 | 00101111 | 057 | 2F |
| 0 | 48 | 00110000 | 060 | 30 |
| 1 | 49 | 00110001 | 061 | 31 |
| 2 | 50 | 00110010 | 062 | 32 |
| 3 | 51 | 00110011 | 063 | 33 |
| 4 | 52 | 00110100 | 064 | 34 |
| 5 | 53 | 00110101 | 065 | 35 |
| 6 | 54 | 00110110 | 066 | 36 |
| 7 | 55 | 00110111 | 067 | 37 |
| 8 | 56 | 00111000 | 070 | 38 |
| 9 | 57 | 00111001 | 071 | 39 |
| : | 58 | 00111010 | 072 | 3A |
| ; | 59 | 00111011 | 073 | 3B |
| < | 60 | 00111100 | 074 | 3C |
| = | 61 | 00111101 | 075 | 3D |
| > | 62 | 00111110 | 076 | 3E |
| ? | 63 | 00111111 | 077 | 3F |

# US ASCII Character Set (Continued)

| ASCII Char. | Character Code | | | |
|---|---|---|---|---|
| | Dec | Binary | Oct | Hex |
| @ | 64 | 01000000 | 100 | 40 |
| A | 65 | 01000001 | 101 | 41 |
| B | 66 | 01000010 | 102 | 42 |
| C | 67 | 01000011 | 103 | 43 |
| D | 68 | 01000100 | 104 | 44 |
| E | 69 | 01000101 | 105 | 45 |
| F | 70 | 01000110 | 106 | 46 |
| G | 71 | 01000111 | 107 | 47 |
| H | 72 | 01001000 | 110 | 48 |
| I | 73 | 01001001 | 111 | 49 |
| J | 74 | 01001010 | 112 | 4A |
| K | 75 | 01001011 | 113 | 4B |
| L | 76 | 01001100 | 114 | 4C |
| M | 77 | 01001101 | 115 | 4D |
| N | 78 | 01001110 | 116 | 4E |
| O | 79 | 01001111 | 117 | 4F |
| P | 80 | 01010000 | 120 | 50 |
| Q | 81 | 01010001 | 121 | 51 |
| R | 82 | 01010010 | 122 | 52 |
| S | 83 | 01010011 | 123 | 53 |
| T | 84 | 01010100 | 124 | 54 |
| U | 85 | 01010101 | 125 | 55 |
| V | 86 | 01010110 | 126 | 56 |
| W | 87 | 01010111 | 127 | 57 |
| X | 88 | 01011000 | 130 | 58 |
| Y | 89 | 01011001 | 131 | 59 |
| Z | 90 | 01011010 | 132 | 5A |
| [ | 91 | 01011011 | 133 | 5B |
| \ | 92 | 01011100 | 134 | 5C |
| ] | 93 | 01011101 | 135 | 5D |
| ^ | 94 | 01011110 | 136 | 5E |
| _ | 95 | 01011111 | 137 | 5F |

| ASCII Char. | Character Code | | | |
|---|---|---|---|---|
| | Dec | Binary | Oct | Hex |
| ` | 96 | 01100000 | 140 | 60 |
| a | 97 | 01100001 | 141 | 61 |
| b | 98 | 01100010 | 142 | 62 |
| c | 99 | 01100011 | 143 | 63 |
| d | 100 | 01100100 | 144 | 64 |
| e | 101 | 01100101 | 145 | 65 |
| f | 102 | 01100110 | 146 | 66 |
| g | 103 | 01100111 | 147 | 67 |
| h | 104 | 01101000 | 150 | 68 |
| i | 105 | 01101001 | 151 | 69 |
| j | 106 | 01101010 | 152 | 6A |
| k | 107 | 01101011 | 153 | 6B |
| l | 108 | 01101100 | 154 | 6C |
| m | 109 | 01101101 | 155 | 6D |
| n | 110 | 01101110 | 156 | 6E |
| o | 111 | 01101111 | 157 | 6F |
| p | 112 | 01110000 | 160 | 70 |
| q | 113 | 01110001 | 161 | 71 |
| r | 114 | 01110010 | 162 | 72 |
| s | 115 | 01110011 | 163 | 73 |
| t | 116 | 01110100 | 164 | 74 |
| u | 117 | 01110101 | 165 | 75 |
| v | 118 | 01110110 | 166 | 76 |
| w | 119 | 01110111 | 167 | 77 |
| x | 120 | 01111000 | 170 | 78 |
| y | 121 | 01111001 | 171 | 79 |
| z | 122 | 01111010 | 172 | 7A |
| { | 123 | 01111011 | 173 | 7B |
| \| | 124 | 01111100 | 174 | 7C |
| } | 125 | 01111101 | 175 | 7D |
| ~ | 126 | 01111110 | 176 | 7E |
| DEL | 127 | 01111111 | 177 | 7F |

# Roman Extension Character Set

| Char. | Character Code | | | |
|---|---|---|---|---|
| | Dec | Binary | Oct | Hex |
| | 128 | 10000000 | 200 | 80 |
| | 129 | 10000001 | 201 | 81 |
| | 130 | 10000010 | 202 | 82 |
| | 131 | 10000011 | 203 | 83 |
| | 132 | 10000100 | 204 | 84 |
| | 133 | 10000101 | 205 | 85 |
| | 134 | 10000110 | 206 | 86 |
| | 135 | 10000111 | 207 | 87 |
| | 136 | 10001000 | 210 | 88 |
| | 137 | 10001001 | 211 | 89 |
| | 138 | 10001010 | 212 | 8A |
| | 139 | 10001011 | 213 | 8B |
| | 140 | 10001100 | 214 | 8C |
| | 141 | 10001101 | 215 | 8D |
| | 142 | 10001110 | 216 | 8E |
| | 143 | 10001111 | 217 | 8F |
| | 144 | 10010000 | 220 | 90 |
| | 145 | 10010001 | 221 | 91 |
| | 146 | 10010010 | 222 | 92 |
| | 147 | 10010011 | 223 | 93 |
| | 148 | 10010100 | 224 | 94 |
| | 149 | 10010101 | 225 | 95 |
| | 150 | 10010110 | 226 | 96 |
| | 151 | 10010111 | 227 | 97 |
| | 152 | 10011000 | 230 | 98 |
| | 153 | 10011001 | 231 | 99 |
| | 154 | 10011010 | 232 | 9A |
| | 155 | 10011011 | 233 | 9B |
| | 156 | 10011100 | 234 | 9C |
| | 157 | 10011101 | 235 | 9D |
| | 158 | 10011110 | 236 | 9E |
| | 159 | 10011111 | 237 | 9F |

| Char. | Character Code | | | |
|---|---|---|---|---|
| | Dec | Binary | Oct | Hex |
| space | 160 | 10100000 | 240 | A0 |
| À | 161 | 10100001 | 241 | A1 |
| Â | 162 | 10100010 | 242 | A2 |
| È | 163 | 10100011 | 243 | A3 |
| Ê | 164 | 10100100 | 244 | A4 |
| Ë | 165 | 10100101 | 245 | A5 |
| Î | 166 | 10100110 | 246 | A6 |
| Ï | 167 | 10100111 | 247 | A7 |
| ´ | 168 | 10101000 | 250 | A8 |
| ` | 169 | 10101001 | 251 | A9 |
| ^ | 170 | 10101010 | 252 | AA |
| ¨ | 171 | 10101011 | 253 | AB |
| ~ | 172 | 10101100 | 254 | AC |
| Ù | 173 | 10101101 | 255 | AD |
| Û | 174 | 10101110 | 256 | AE |
| £ | 175 | 10101111 | 257 | AF |
| ¯ | 176 | 10110000 | 260 | B0 |
| Ý | 177 | 10110001 | 261 | B1 |
| ý | 178 | 10110010 | 262 | B2 |
| ° | 179 | 10110011 | 263 | B3 |
| Ç | 180 | 10110100 | 264 | B4 |
| ç | 181 | 10110101 | 265 | B5 |
| Ñ | 182 | 10110110 | 266 | B6 |
| ñ | 183 | 10110111 | 267 | B7 |
| ¡ | 184 | 10111000 | 270 | B8 |
| ¿ | 185 | 10111001 | 271 | B9 |
| ¤ | 186 | 10111010 | 272 | BA |
| £ | 187 | 10111011 | 273 | BB |
| ¥ | 188 | 10111100 | 274 | BC |
| § | 189 | 10111101 | 275 | BD |
| ƒ | 190 | 10111110 | 276 | BE |
| ¢ | 191 | 10111111 | 277 | BF |

# Roman Extension Character Set (Continued)

| Char. | Character Code | | | |
|---|---|---|---|---|
| | Dec | Binary | Oct | Hex |
| â | 192 | 11000000 | 300 | C0 |
| ê | 193 | 11000001 | 301 | C1 |
| ô | 194 | 11000010 | 302 | C2 |
| û | 195 | 11000011 | 303 | C3 |
| á | 196 | 11000100 | 304 | C4 |
| é | 197 | 11000101 | 305 | C5 |
| ó | 198 | 11000110 | 306 | C6 |
| ú | 199 | 11000111 | 307 | C7 |
| à | 200 | 11001000 | 310 | C8 |
| è | 201 | 11001001 | 311 | C9 |
| ò | 202 | 11001010 | 312 | CA |
| ù | 203 | 11001011 | 313 | CB |
| ä | 204 | 11001100 | 314 | CC |
| ë | 205 | 11001101 | 315 | CD |
| ö | 206 | 11001110 | 316 | CE |
| ü | 207 | 11001111 | 317 | CF |
| À | 208 | 11010000 | 320 | D0 |
| Î | 209 | 11010001 | 321 | D1 |
| Ø | 210 | 11010010 | 322 | D2 |
| Æ | 211 | 11010011 | 323 | D3 |
| å | 212 | 11010100 | 324 | D4 |
| í | 213 | 11010101 | 325 | D5 |
| ø | 214 | 11010110 | 326 | D6 |
| æ | 215 | 11010111 | 327 | D7 |
| Ä | 216 | 11011000 | 330 | D8 |
| ì | 217 | 11011001 | 331 | D9 |
| Ö | 218 | 11011010 | 332 | DA |
| Ü | 219 | 11011011 | 333 | DB |
| É | 220 | 11011100 | 334 | DC |
| ï | 221 | 11011101 | 335 | DD |
| ß | 222 | 11011110 | 336 | DE |
| Ô | 223 | 11011111 | 337 | DF |

| Char. | Character Code | | | |
|---|---|---|---|---|
| | Dec | Binary | Oct | Hex |
| Á | 224 | 11100000 | 340 | E0 |
| Ã | 225 | 11100001 | 341 | E1 |
| ã | 226 | 11100010 | 342 | E2 |
| Ð | 227 | 11100011 | 343 | E3 |
| ð | 228 | 11100100 | 344 | E4 |
| Í | 229 | 11100101 | 345 | E5 |
| Ì | 230 | 11100110 | 346 | E6 |
| Ó | 231 | 11100111 | 347 | E7 |
| Ò | 232 | 11101000 | 350 | E8 |
| Õ | 233 | 11101001 | 351 | E9 |
| õ | 234 | 11101010 | 352 | EA |
| Š | 235 | 11101011 | 353 | EB |
| š | 236 | 11101100 | 354 | EC |
| Ú | 237 | 11101101 | 355 | ED |
| Ÿ | 238 | 11101110 | 356 | EE |
| ÿ | 239 | 11101111 | 357 | EF |
| Þ | 240 | 11110000 | 360 | F0 |
| þ | 241 | 11110001 | 361 | F1 |
| · | 242 | 11110010 | 362 | F2 |
| µ | 243 | 11110011 | 363 | F3 |
| ¶ | 244 | 11110100 | 364 | F4 |
| ¾ | 245 | 11110101 | 365 | F5 |
| ‾ | 246 | 11110110 | 366 | F6 |
| ¼ | 247 | 11110111 | 367 | F7 |
| ½ | 248 | 11111000 | 370 | F8 |
| ª | 249 | 11111001 | 371 | F9 |
| º | 250 | 11111010 | 372 | FA |
| « | 251 | 11111011 | 373 | FB |
| ■ | 252 | 11111100 | 374 | FC |
| » | 253 | 11111101 | 375 | FD |
| ± | 254 | 11111110 | 376 | FE |
| ▒ | 255 | 11111111 | 377 | FF |

# E

# Display Control Characters

The table below describes each display control character used by the handheld. Note that these characters, as well as any character in the handheld's character set, can be specified with & and its two-digit hexadecimal ASCII code in a literal (& is specified by &&).

**Display Control Characters**

| Hex Value | Meaning |
|---|---|
| 01 (SOH) | Turn on cursor. |
| 02 (STX) | Turn off cursor. |
| 06 (ACK) | High tone beep for 0.5 second. |
| 07 (BEL) | Low tone beep for 0.5 second. |
| 08 (BS) | Move cursor left one column. When the cursor reaches the left end of the line, it will back up to the right end of the previous line. When the cursor reaches the top left corner, backspace will have no effect. |
| 0A (LF) | Move cursor down one line. If the cursor is on the bottom line, the display contents will scroll up one line. |
| 0B (VT) | Clear every character from the cursor position to the end of the current line. |
| 0C (FF) | Move cursor to top left corner and clear the display. |
| 0D (CR) | Move cursor to left end of current line. |
| 0E (SO) | Change keyboard to numeric mode (underline cursor). |
| 0F (SI) | Change keyboard to alpha mode (block cursor). |
| 1E (RS) | Turn on electroluminescent backlight. |
| 1F (US) | Turn off electroluminescent backlight. |


Printed in U.S.A. 12/86